

pitón para Análisis de los datos

LITIGIOS DE DATOS CON PANDAS,
NUMPY Y IPYTHON



energizado por



wes mckinney

Python for Data Analysis

Get complete instructions for manipulating, processing, cleaning, and crunching datasets in Python. Updated for Python 3.6, the second edition of this hands-on guide is packed with practical case studies that show you how to solve a broad set of data analysis problems effectively. You'll learn the latest versions of pandas, NumPy, IPython, and Jupyter in the process.

Written by Wes McKinney, the creator of the Python pandas project, this book is a practical, modern introduction to data science tools in Python. It's ideal for analysts new to Python and for Python programmers new to data science and scientific computing. Data files and related material are available on GitHub.

- Use the IPython shell and Jupyter Notebook for exploratory computing
- Learn basic and advanced features in NumPy (Numerical Python)
- Get started with data analysis tools in the pandas library
- Use flexible tools to load, clean, transform, merge, and reshape data
- Create informative visualizations with matplotlib
- Apply the pandas groupby facility to slice, dice, and summarize datasets
- Analyze and manipulate regular and irregular time series data
- Learn how to solve real-world data analysis problems with thorough, detailed examples

“Already a classic of the Python data ecosystem, this new edition is updated in key areas that enhance its unique value, from Python 3.6 to the latest features in pandas. By explaining the why and how of Python's data tools, this book helps the reader learn to use them effectively in new and creative ways. It is an essential part of any modern library of data-intensive computing.”

—Fernando Pérez

Assistant Professor of Statistics,
UC Berkeley, IPython creator and
cofounder of Project Jupyter

Wes McKinney is the creator of pandas, the popular open source Python library for data analysis. He is an active public speaker and open source Python and C++ developer in the Python data science community and the Apache Software Foundation. He works as a software architect in New York City.

US \$49.99

CAN \$65.99

ISBN: 978-1-491-95766-0



9



Twitter: @oreillymedia
facebook.com/oreilly

SEGUNDA EDICION

Python para análisis de datos

Gestión de datos con Pandas, NumPy e IPython

wes mckinney

Pekín Boston Farnham Sebastopol Tokio

O'REILLY®

Python para análisis de
datos por Wes McKinney

Derechos de autor © 2018 William McKinney. Reservados todos los derechos.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Los libros de O'Reilly se pueden comprar con fines educativos, comerciales o de promoción de ventas. Las ediciones en línea también están disponibles para la mayoría de los títulos (<http://oreilly.com/safari>). Para obtener más información, comuníquese con nuestro departamento de ventas institucionales/corporativas: 800-998-9938 o corporate@oreilly.com.

Montaje: Marie Beaugureau
Editora de producción: Kristen Brown
Correctora: Jasmine Kwityn
Correctora: Rachel Monaghan

Indexador: Lucie Haskins
Diseñador de interiores: David Futato
Diseño de portada: Karen Montgomery
Ilustrador: Rebecca Demarest

Octubre 2012: Primera edición
Octubre 2017: Segunda edición

Historial de revisiones de la segunda edición

2017-09-25: Primer lanzamiento

Consulte <http://oreilly.com/catalog/errata.csp?isbn=9781491957660> para obtener detalles de la versión.

El logotipo de O'Reilly es una marca comercial registrada de O'Reilly Media, Inc. Python para análisis de datos, la imagen de portada y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Si bien el editor y el autor se han esforzado de buena fe para garantizar que la información y las instrucciones contenidas en este trabajo sean precisas, el editor y el autor renuncian a toda responsabilidad por errores u omisiones, incluida, entre otras, la responsabilidad por daños que resulten del uso de o confianza en este trabajo. El uso de la información e instrucciones contenidas en este trabajo es bajo su propio riesgo. Si alguna muestra de código u otra tecnología que este trabajo contiene o describe está sujeta a licencias de código abierto o a los derechos de propiedad intelectual de otros, es su responsabilidad asegurarse de que su uso cumpla con dichas licencias y/o derechos.

978-1-491-95766-0

[LSI]

Tabla de contenido

Prefacio. xi

1. Preliminares. 1

1.1 ¿De qué trata este libro? 1

 ¿Qué tipo de datos? 1

1.2 ¿Por qué Python para el análisis de datos? 2

 Python como 2

 pegamento Resolviendo el problema de los "dos 3

 idiomas" ¿Por qué no Python? 3

1.3 Bibliotecas esenciales de Python 4

 NumPy 4

 pandas 4

 matplotlib 5

 IPython y Jupyter SciPy 6

 scikit-learn 7

 statsmodels 8

1.4 Instalación y configuración 8

 Windows 9

 Apple (OS X, macOS) 9

 GNU/Linux 9

 Instalación o actualización de paquetes Python 10

 Python 2 y Python 3 Entornos 11

 de desarrollo integrados (IDE) y editores de texto 1.5 Comunidad y conferencias 1.6 11

Navegación por este libro Ejemplos de 12

 código Datos para ejemplos 12

 13

 13

Convenciones de importación	14
Jerga	14
2. Conceptos básicos del lenguaje Python, IPython y Jupyter Notebooks.	15
2.1 El intérprete de Python 2.2	15
Conceptos básicos de	17
IPython Ejecución de IPython Shell	17
Ejecución de Jupyter Notebook	18
Finalización de	21
pestañas	23
Introspección El comando	25
%run Ejecución de código desde el portapapeles	26
Atajos de teclado de terminal Acerca	27
de los comandos mágicos	28
Integración con Matplotlib	29
2.3 Conceptos básicos del lenguaje	30
Python Semántica del	30
lenguaje Tipos	38
escalares Flujo de control	46
3. Estructuras de datos, funciones y archivos integrados.	51
3.1 Estructuras de datos y secuencias	51
Lista	51
de	54
tuplas Funciones de secuencia	59
incorporadas dict	61
colocar	67
Comprensiones de listas, conjuntos y dictados	67
3.2 Funciones	69
Espacios de nombres, ámbito y funciones locales	70
que devuelven valores múltiples	71
Las funciones son objetos	72
Funciones anónimas (Lambda) Currying:	73
Argumentos parciales Generadores de aplicaciones	74
Manejo de	75
errores y excepciones	77
3.3 Archivos y el	80
sistema operativo Bytes y Unicode con	83
archivos	84
3.4 Conclusión	84
4. Conceptos básicos de NumPy: arreglos y computación vectorizada.	87
85 4.1 NumPy ndarray: un objeto de matriz multidimensional	87

Creación de ndarrays	88
Tipos de datos para ndarrays	90
Aritmética con NumPy Arrays	93
Indexación y corte básicos	94
Indexación booleana	99
Indexación elegante	102
Transposición de matrices y ejes de intercambio	103
4.2 Funciones universales: funciones de matriz rápidas	105
basadas en elementos 4.3 Programación orientada a matrices con matrices	108
Expresión de la lógica condicional como operaciones con matrices	109
Métodos matemáticos y estadísticos	111
Métodos para matrices booleanas	113
Clasificación de conjuntos únicos	114
y otros lógicos 4.4 Entrada y salida de archivos	115
con matrices 4.5	116
Álgebra lineal 4.6 Generación de números pseudoaleatorios	118
4.7 Ejemplo:	119
Paseos aleatorios Simulación de muchos paseos aleatorios a la vez	121
4.8 Conclusión	122
5. Introducción a los pandas.	123 124
5.1 Introducción a las estructuras de datos pandas	
Series	124
DataFrame	128
Index Objects	134
5.2 Funcionalidad esencial	136
Reindexación	136
Eliminación de entradas de un eje	138
Indexación, selección y filtrado de índices enteros	140
Aritmética	145
y función de alineación de datos	146
Aplicación y mapeo	151
Clasificación y clasificación de índices	153
de eje con etiquetas duplicadas 5.3	157
Resumen y cálculo de estadísticas descriptivas	158
Valores únicos de correlación y covarianza, conteo de valores y membresía	160
5.4 Conclusión	162
	165
6. Carga de datos, almacenamiento y formatos de archivo.	167
6.1 Leer y escribir datos en formato de texto	167

Lectura de archivos de texto por partes	173
Escritura de datos en formato de texto	175
Trabajo con formatos delimitados	176
Datos JSON XML y HTML: Web	178
Scraping 6.2 Formatos de datos binarios	180
Uso del formato HDF5	184
Lectura de archivos de Microsoft Excel 6.3	186
Interacción con API web 6.4	187
Interacción con bases de datos 6.5	188
Conclusión	190
7. Limpieza y preparación de datos	191
7.1 Manejo de datos faltantes	191
Filtrar datos faltantes	193
Completar datos faltantes 7.2	195
Transformación de datos	197
Eliminación de duplicados	197
Transformación de datos	198
Uso de una función o mapeo	200
Reemplazo de valores	201
Cambio de nombre de índices de eje	203
Discretización y clasificación en intervalos	205
Detección y filtrado de valores atípicos	206
Permutación y muestreo aleatorio	208
Cómputo de indicadores/variables ficticias 7.3	211
Manipulación de cadenas	211
Métodos de objetos de cadena	213
Expresiones regulares	216
Funciones de cadena vectorizadas en pandas 7.4	219
Conclusión	219
8. Gestión de datos: unir, combinar y remodelar	221
8.1 Indexación jerárquica	221
Reordenación y clasificación de niveles	224
Estadísticas resumidas por nivel	225
Indexación con columnas de DataFrame	225
8.2 Combinación y fusión de conjuntos de datos	227
Uniones de marcos de datos de estilo de base de datos	227
Fusión en el índice	232
Concatenación a lo largo de un eje	236
Combinación de datos con superposición	241
8.3 Reformar y pivotar	242

Remodelación con indexación jerárquica	243
Rotación de formato “largo” a “ancho”	246
Rotación de formato “ancho” a “largo”	249
8.4 Conclusión	251
9. Trazado y Visualización.	253
9.1 Breve	
introducción a la API de matplotlib Figuras	253
y subgráficos Colores,	255
marcadores y estilos de línea Marcas,	259
etiquetas y leyendas Anotaciones	261
y dibujos en un subgráfico Guardar gráficos en	265
un archivo Matplotlib	267
Configuración 9.2 Representación	268
gráfica con pandas y seaborn Gráficos de	268
líneas	269
Gráficos	272
de barras Histogramas y gráficos de	277
densidad Dispersión o	280
Gráficos de puntos Cuadrículas de	283
facetas y datos categóricos 9.3 Otras	285
herramientas de visualización de Python 9.4 Conclusión	286
10. Agregación de Datos y Operaciones de Grupo.	287
10.1	
Mecánica de GroupBy Iteración	288
sobre grupos Selección de	291
una columna o subconjunto de columnas Agrupación	293
con dictados y series Agrupación con	294
funciones Agrupación por	295
niveles de índice 10.2 Agregación	295
de datos	296
Aplicación por columnas y de función múltiple Devolución	298
de datos agregados sin índices de fila 10.3 Aplicar: División	301
general-aplicación-combinación Supresión de	302
las claves de grupo Análisis	304
cuantílico y de cubo Ejemplo:	305
Relleno de valores faltantes con valores específicos del grupo Ejemplo:	306
Muestreo aleatorio y permutación Ejemplo: Grupo	308
ponderado Ejemplo de promedio y correlación: regresión	310
lineal por grupos	312
10.4 Tablas dinámicas y tabulación cruzada	313
Tabulaciones cruzadas: Tabulación	315
cruzada 10.5 Conclusión	316

11. Serie temporal.	317
11.1 Herramientas y tipos de datos de fecha y hora	318
Conversión entre cadena y fecha y hora	319
11.2 Conceptos básicos de series temporales	322
Indexación, selección, creación de	323
subconjuntos de series temporales con índices	326
11.3 Rangos de fechas, frecuencias y desplazamientos	327
Generar intervalos de fechas	328
Frecuencias y compensaciones de	330
fechas Cambio de datos (adelantados y atrasados)	332
11.4 Manejo de zonas horarias	335
Operaciones de localización y conversión de zonas	335
horarias con objetos de marcas de tiempo con reconocimiento de zona	338
horaria Operaciones entre diferentes zonas horarias	339
11.5 Períodos y aritmética de períodos	339
Conversión de frecuencia de períodos	340
Frecuencias de períodos trimestrales	342
Conversión de marcas de tiempo	344
Períodos (y viceversa) Creación de un índice de período a partir de	345
11.6 Remuestreo y conversión de frecuencia	348
Remuestreo inferior	349
Remuestreo superior e interpolación	352
Remuestreo con períodos	353
11.7 Funciones de ventana móvil	354
Funciones ponderadas exponencialmente	358
Funciones binarias de ventana móvil	359
Funciones de ventana móvil definidas por el usuario	361
11.8 Conclusión	362
12. Pandas avanzados.	363
12.1 Antecedentes de	363
datos categóricos y motivación	363
Tipo categórico en pandas	365
Cálculos con categóricos	367
Métodos categóricos	370
12.2 GroupBy avanzado	373
Use transformaciones de grupo y GroupBys "desenvueltos"	373
Remuestreo de tiempo	377
12.3 Técnicas para el agrupado	378
encadenamiento de métodos	380
12.4 Conclusión El método de tubería	381

13. Introducción a las bibliotecas de modelado en Python.	
383 13.1 Interfaz entre pandas y el código del modelo	383
13.2 Creación de descripciones de modelos con	386
Patsy Transformaciones de datos en Patsy	389
Fórmulas Datos categóricos y	390
Patsy 13.3 Introducción a statsmodels	393
Estimación de modelos	393
lineales Estimación de procesos de series temporales	396
13.4 Introducción a scikit-learn	397
13.5 Continuación de su educación	401
14. Ejemplos de análisis de datos.	
403 14.1 1.USA.gov Datos de Bitly	403
Conteo de zonas horarias en Pure Python	404
Conteo de zonas horarias con pandas	406
14.2 Conjunto de datos MovieLens 1M	413
Desacuerdo de calificación de medición	418
14.3 Nombres de bebés de EE. UU. 1880–	419
2010 Análisis de tendencias de nombres	425
14.4 Base de datos de alimentos del	434
USDA 14.5 Base de datos de la Comisión Federal de Elecciones de 2012	440
Estadísticas de donaciones por ocupación y agrupación de	442
empleadores Cantidades de	445
donaciones Estadísticas de	447
donaciones por estado 14.6 Conclusión	448
A. NumPy avanzado.	449
A.1 Jerarquía NumPy dtype de	449
elementos internos del objeto	450
ndarray A.2 Manipulación avanzada de matrices	451
Remodelación de	452
arreglos C versus orden	454
Fortran Concatenación y división de arreglos	454
Elementos repetidos: mosaico y repetición	457
Equivalentes de indexación sofisticados: tomar y	459
poner A.5 Matrices	460
estructuradas y de registro	462
	465
	466
	466
	468
	469

Tipos anidados y campos multidimensionales ¿Por qué utilizar matrices estructuradas?	469
A.6 Más acerca de la clasificación	471
de clasificaciones indirectas: argsort y lexsort	472
Algoritmos de clasificación alternativos	474
Clasificación parcial de matrices	474
numpy.searchsorted: búsqueda de elementos en una matriz ordenada	475
A.7 Escribir funciones NumPy rápidas con Numba	476
Creación de objetos numpy.ufunc personalizados con Numba A.8	478
Entrada y salida de matrices avanzadas Archivos	478
asignados a la memoria HDF5	478
y otras opciones de almacenamiento de matrices A.9	480
Consejos de rendimiento La	480
importancia de la memoria contigua	480
 B. Más información sobre el sistema IPython.	 483
483 B.1 Uso del historial de comandos	483
Búsqueda y reutilización de las variables de entrada y salida del historial de comandos	484
B.2 Interacción con el sistema operativo Shell	485
Comandos y alias Directorio	486
Sistema de marcadores B.3	487
Herramientas de desarrollo de software	487
Depurador interactivo	488
Código de temporización: %time y %timeit Básico Generación de perfiles:	492
%prun y %run -p Generación de perfiles de una función línea por línea	494
B.4 Sugerencias para el desarrollo de código productivo usando IPython	496
Recarga de módulos Dependencias	498
Sugerencias de diseño	499
de código B.5 Funciones avanzadas de	500
IPython Creación de sus propias clases Perfiles y configuración compatibles con	500
IPython B.6 Conclusión	501
Índice.	503
	505

Prefacio

Nuevo para la Segunda Edición

La primera edición de este libro se publicó en 2012, durante una época en la que las bibliotecas de análisis de datos de código abierto para Python (como pandas) eran muy nuevas y se desarrollaban rápidamente. En esta segunda edición actualizada y ampliada, revisé los capítulos para tener en cuenta tanto los cambios incompatibles y las obsolescencias como las nuevas características que se produjeron en los últimos cinco años. También agregué contenido nuevo para presentar herramientas que no existían en 2012 o que no habían madurado lo suficiente como para hacer el primer corte.

Finalmente, he tratado de evitar escribir sobre proyectos de código abierto nuevos o de vanguardia que pueden no haber tenido la oportunidad de madurar. Me gustaría que los lectores de esta edición descubran que el contenido sigue siendo casi tan relevante en 2020 o 2021 como lo es en 2017.

Las principales actualizaciones de esta segunda edición incluyen:

- Todo el código, incluido el tutorial de Python, actualizado para Python 3.6 (la primera edición usó Python 2.7)
- Instrucciones de instalación de Python actualizadas para Anaconda Python Distribution y otros paquetes de Python necesarios
- Actualizaciones para las últimas versiones de la biblioteca pandas en 2017 • Un nuevo capítulo sobre algunas herramientas pandas más avanzadas y algunos otros consejos de uso • Una breve introducción al uso de statsmodels y scikit-learn

También reorganicé una parte importante del contenido de la primera edición para que el libro fuera más accesible para los recién llegados.

Las convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

Cursiva

Indica nuevos términos, URL, direcciones de correo electrónico, nombres de archivo y extensiones de archivo.

Ancho constante Se

utiliza para listas de programas, así como dentro de párrafos para referirse a elementos de programas como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

Negrita de ancho constante

Muestra comandos u otro texto que el usuario debe escribir literalmente.

Cursiva de ancho constante

Muestra texto que debe ser reemplazado con valores proporcionados por el usuario o por valores determinados por el contexto.



Este elemento significa un consejo o sugerencia.



Este elemento significa una nota general.



Este elemento indica una advertencia o precaución.

Uso de ejemplos de código

Puede encontrar archivos de datos y material relacionado para cada capítulo en el repositorio de GitHub de este libro en <http://github.com/wesm/pydata-book>.

Este libro está aquí para ayudarle a hacer su trabajo. En general, si se ofrece un código de ejemplo con este libro, puede usarlo en sus programas y documentación. No es necesario que se comunique con nosotros para obtener permiso a menos que esté reproduciendo una parte significativa del código. Por ejemplo, escribir un programa que use varios fragmentos de código de este

libro no requiere permiso. Vender o distribuir un CD-ROM de ejemplos de libros de O'Reilly requiere permiso. Responder una pregunta citando este libro y citando código de ejemplo no requiere permiso. La incorporación de una cantidad significativa de código de ejemplo de este libro en la documentación de su producto requiere permiso.

Apreciamos, pero no requerimos, atribución. Una atribución suele incluir el título, el autor, el editor y el ISBN. Por ejemplo: "Python para análisis de datos por Wes McKinney (O'Reilly). Derechos de autor 2017 Wes McKinney, 978-1-491-95766-0."

Si cree que su uso de los ejemplos de código está fuera del uso justo o del permiso otorgado anteriormente, no dude en contactarnos en permisos@oreilly.com.

Safari O'Reilly



Safari (anteriormente Safari Books Online) es una plataforma de referencia y capacitación basada en membresía para empresas, gobiernos, educadores e individuos.

Los miembros tienen acceso a miles de libros, videos de capacitación, rutas de aprendizaje, tutoriales interactivos y listas de reproducción seleccionadas de más de 250 editores, incluidos O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett y Course Technology, entre otros.

Para obtener más información, visite <http://oreilly.com/safari>.

Cómo contactarnos

Dirija sus comentarios y preguntas sobre este libro a la editorial:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (en los Estados Unidos o Canadá)
707-829-0515 (internacional o local)
707-829-0104 (fax)

Tenemos una página web para este libro, donde enumeramos las erratas, ejemplos y cualquier información adicional. Puede acceder a esta página en http://bit.ly/python_data_analysis_2e.

Para comentar o hacer preguntas técnicas sobre este libro, envíe un correo electrónico a bookquestions@oreilly.com.

Para obtener más información sobre nuestros libros, cursos, conferencias y noticias, visite nuestro sitio web en <http://www.oreilly.com>.

Encuéntrenos en Facebook: <http://facebook.com/oreilly>

Síguenos en Twitter: <http://twitter.com/oreillymedia>

Míranos en YouTube: <http://www.youtube.com/oreillymedia>

Expresiones de gratitud

Este trabajo es el producto de muchos años de debates fructíferos, colaboraciones y asistencia con y de muchas personas en todo el mundo. Me gustaría agradecer a algunos de ellos.

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

Nuestro querido amigo y colega John D. Hunter falleció después de una batalla contra el cáncer de colon el 28 de agosto de 2012. Esto fue poco tiempo después de haber completado el manuscrito final de la primera edición de este libro.

Sería difícil exagerar el impacto y el legado de John en las comunidades científicas y de datos de Python. Además de desarrollar matplotlib a principios de la década de 2000 (una época en la que Python no era tan popular), ayudó a dar forma a la cultura de una generación crítica de desarrolladores de código abierto que se han convertido en pilares del ecosistema de Python que ahora solemos tomar por sentado.

Tuve la suerte de conectarme con John al principio de mi carrera de código abierto en enero de 2010, justo después de lanzar pandas 0.1. Su inspiración y tutoría me ayudaron a seguir adelante, incluso en los momentos más oscuros, con mi visión de pandas y Python como un lenguaje de análisis de datos de primera clase.

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

Agradecimientos a la Segunda Edición (2017)

Han pasado casi cinco años desde que completé el manuscrito de la primera edición de este libro en julio de 2012. Mucho ha cambiado. La comunidad de Python ha crecido enormemente y el ecosistema de software de código abierto que la rodea ha florecido.

Esta nueva edición del libro no existiría si no fuera por los esfuerzos incansables de los desarrolladores principales de pandas, que han convertido el proyecto y su comunidad de usuarios en una de las piedras angulares del ecosistema de ciencia de datos de Python. Estos incluyen, entre otros, Tom Augspurger, Joris van den Bossche, Chris Bartak, Phillip Cloud, gyoung, Andy Hayden, Masaaki Horikoshi, Stephan Hoyer, Adam Klein, Wouter Overmeire, Jeff Reback, Chang She, Skipper Seabold, Jeff Tratner, y sí.

En cuanto a la redacción de esta segunda edición, me gustaría agradecer al personal de O'Reilly que me ayudó pacientemente con el proceso de redacción. Esto incluye a Marie Beaugureau, Ben Lorica y Colleen Toporek. Nuevamente tuve destacados revisores técnicos con la contribución de Tom Augpurger, Paul Barry, Hugh Brown, Jonathan Coe y Andreas Müller. Gracias.

La primera edición de este libro ha sido traducida a muchos idiomas extranjeros, incluidos chino, francés, alemán, japonés, coreano y ruso. Traducir todo este contenido y ponerlo a disposición de un público más amplio es un esfuerzo enorme y, a menudo, ingrato. Gracias por ayudar a más personas en el mundo a aprender a programar y utilizar herramientas de análisis de datos.

También tengo la suerte de haber contado con el apoyo de Cloudera y Two Sigma Investments para mis continuos esfuerzos de desarrollo de código abierto durante los últimos años. Con los proyectos de software de código abierto con menos recursos que nunca en relación con el tamaño de las bases de usuarios, se está volviendo cada vez más importante para las empresas brindar apoyo para el desarrollo de proyectos clave de código abierto. Es la cosa justa que hacer.

Agradecimientos a la Primera Edición (2012)

Habría sido difícil para mí escribir este libro sin el apoyo de un gran número de personas.

En el equipo de O'Reilly, estoy muy agradecido por mis editores, Meghan Blanchette y Julie Steele, quienes me guiaron a través del proceso. Mike Loukides también trabajó conmigo en las etapas de la propuesta y me ayudó a hacer realidad el libro.

Recibí una gran cantidad de revisión técnica de un gran elenco de personajes. En particular, Martin Blais y Hugh Brown fueron increíblemente útiles para mejorar los ejemplos, la claridad y la organización del libro de principio a fin. James Long, Drew Conway, Fernando Pérez, Brian Granger, Thomas Kluyver, Adam Klein, Josh Klein, Chang She y Stéfan van der Walt revisaron cada uno uno o más capítulos, brindando comentarios puntuales desde muchas perspectivas diferentes.

Recibí muchas ideas geniales para ejemplos y conjuntos de datos de amigos y colegas en 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 en deuda con los muchos líderes de la comunidad científica de código abierto de Python que sentaron las bases para mi trabajo de desarrollo y me alentaron mientras escribía este libro: el equipo central 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 muchos otros para mencionarlos. Varias otras personas brindaron una gran cantidad de apoyo, ideas y aliento a lo largo del camino: Drew Conway, Sean Taylor, Giuseppe Paleologo, Jared Lander, David Epstein, John Krowas, Joshua Bloom, Den Pilsworth, John Myles-White , y muchos otros que he olvidado.

También me gustaría agradecer a varias personas de mis años de formación. Primero, mis antiguos colegas de AQR que me animaron en mi trabajo con pandas a lo largo de los años: Alex Reyfman, Michael Wong, Tim Sargen, Oktay Kurbanov, Matthew Tschantz, Roni Israelov, Michael Katz, Chris Uga, Prasad Ramanan, Ted Square y Hoon Kim. Por último, mis asesores académicos Haynes Miller (MIT) y Mike West (Duke).

Recibí una ayuda significativa de Phillip Cloud y Joris Van den Bossche en 2014 para actualizar los ejemplos de código del libro y corregir algunas otras imprecisiones debido a cambios en pandas.

En el aspecto personal, Casey me brindó un apoyo invaluable en el día a día durante el proceso de escritura, tolerando mis altibajos mientras preparaba el borrador final además de un programa ya sobrecargado. Por último, mis padres, Bill y Kim, me enseñaron a seguir siempre mis sueños y nunca conformarme con menos.

Preliminares

1.1 ¿De qué trata este libro?

Este libro se ocupa de los aspectos básicos de la manipulación, el procesamiento, la limpieza y el procesamiento de datos en Python. Mi objetivo es ofrecer una guía de las partes del lenguaje de programación Python y su ecosistema de biblioteca orientada a datos y herramientas que lo equiparán para convertirse en un analista de datos efectivo. Si bien el "análisis de datos" se encuentra en el título del libro, el enfoque se centra específicamente en la programación, las bibliotecas y las herramientas de Python en lugar de la metodología de análisis de datos. Esta es la programación de Python que necesita para el análisis de datos.

¿Qué tipo de datos?

Cuando digo "datos", ¿a qué me refiero exactamente? El enfoque principal está en los datos estructurados, un término deliberadamente vago que abarca muchas formas comunes diferentes de datos, como:

- Datos tabulares o similares a una hoja de cálculo en los que cada columna puede ser de un tipo diferente (cadena, numérico, fecha u otro). Esto incluye la mayoría de los tipos de datos comúnmente almacenados en bases de datos relacionales o archivos de texto delimitados por tabulaciones o comas.
- Arrays multidimensionales (matrices).
- Múltiples tablas de datos interrelacionados por columnas clave (lo que sería principal o claves foráneas para un usuario de SQL).
- Series de tiempo uniforme o desigualmente espaciadas.

Esto no es de ninguna manera una lista completa. Aunque no siempre sea obvio, un gran porcentaje de conjuntos de datos se puede transformar en una forma estructurada que sea más adecuada para el análisis y el modelado. De lo contrario, es posible extraer características de un conjunto de datos.

en una forma estructurada. Como ejemplo, una colección de artículos de noticias podría procesarse en una tabla de frecuencia de palabras, que luego podría usarse para realizar un análisis de sentimiento.

La mayoría de los usuarios de programas de hojas de cálculo como Microsoft Excel, quizás la herramienta de análisis de datos más utilizada en el mundo, no serán ajenos a este tipo de datos.

1.2 ¿Por qué Python para el análisis de datos?

Para muchas personas, 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 interpretados más populares, junto con Perl, Ruby y otros. Python y Ruby se han vuelto especialmente populares desde 2005 aproximadamente para crear sitios web utilizando sus numerosos marcos web, como Rails (Ruby) y Django (Python). Dichos lenguajes a menudo se denominan lenguajes de secuencias de comandos, ya que se pueden usar para escribir rápidamente pequeños programas o secuencias de comandos para automatizar otras tareas. No me gusta el término "lenguaje de secuencias de comandos", ya que conlleva la connotación de que no se pueden usar para crear software serio. Entre los lenguajes interpretados, por varias razones históricas y culturales, Python ha desarrollado una comunidad de análisis de datos y computación científica grande y activa. En los últimos 10 años, Python ha pasado de ser un lenguaje informático científico de vanguardia o "bajo su propio riesgo" a uno de los lenguajes más importantes para la ciencia de datos, el aprendizaje automático y el desarrollo de software en general en la academia y la industria.

Para el análisis de datos y la computación interactiva y la visualización de datos, Python inevitablemente establecerá comparaciones con otros lenguajes de programación comercial y de código abierto y herramientas de amplio uso, como R, MATLAB, SAS, Stata y otros. En los últimos años, el soporte mejorado de Python para bibliotecas (como pandas y scikit-learn) lo ha convertido en una opción popular para las tareas de análisis de datos. Combinado con la fortaleza general de Python para la ingeniería de software de propósito general, es una excelente opción como lenguaje principal para crear aplicaciones de datos.

Python como

pegamento Parte del éxito de Python en la computación científica es la facilidad de integrar código C, C++ y FORTRAN. La mayoría de los entornos informáticos modernos comparten un conjunto similar de bibliotecas FORTRAN y C heredadas para realizar álgebra lineal, optimización, integración, transformadas rápidas de Fourier y otros algoritmos similares. La misma historia ha sido válida para muchas empresas y laboratorios nacionales que han utilizado Python para unir décadas de software heredado.

Muchos programas consisten en pequeñas porciones de código en las que se gasta la mayor parte del tiempo, con grandes cantidades de "código adhesivo" que no se ejecuta con frecuencia. En muchos casos, el tiempo de ejecución del código adhesivo es insignificante; el esfuerzo se invierte más fructíferamente en optimizar

los cuellos de botella computacionales, a veces moviendo el código a un lenguaje de nivel inferior como C.

Resolviendo el problema de los "dos idiomas" En muchas

organizaciones, es común investigar, crear prototipos y probar nuevas ideas usando un lenguaje de computación más especializado como SAS o R y luego trasladar esas ideas para que formen parte de un sistema de producción más grande escrito en , digamos, Java, C# o C++. Lo que la gente encuentra cada vez más es que Python es un lenguaje adecuado no solo para investigar y crear prototipos, sino también para construir los sistemas de producción. ¿Por qué mantener dos entornos de desarrollo cuando uno es suficiente? Creo que cada vez más empresas seguirán este camino, ya que a menudo hay beneficios organizacionales significativos al tener tanto a investigadores como a ingenieros de software usando el mismo conjunto de herramientas de programación.

¿Por qué no Python?

Si bien Python es un entorno excelente para crear muchos tipos de aplicaciones analíticas y sistemas de uso general, hay una serie de usos para los que Python puede ser menos adecuado.

Como Python es un lenguaje de programación interpretado, en general, la mayoría del código de Python se ejecutará considerablemente más lento que el código escrito en un lenguaje compilado como Java o C++. Como el tiempo del programador suele ser más valioso que el tiempo de la CPU, muchos están felices de hacer este intercambio. Sin embargo, en una aplicación con una latencia muy baja o requisitos exigentes de utilización de recursos (por ejemplo, un sistema comercial de alta frecuencia), el tiempo dedicado a programar en un lenguaje de nivel inferior (pero también de menor productividad) como C++ para lograr el máximo rendimiento posible podría ser tiempo bien invertido.

Python puede ser un lenguaje desafiante para crear aplicaciones de subprocesos múltiples altamente concurrentes, particularmente aplicaciones con muchos subprocesos vinculados a la CPU. La razón de esto es que tiene lo que se conoce como bloqueo global del intérprete (GIL), un mecanismo que evita que el intérprete ejecute más de una instrucción de Python a la vez.

Las razones técnicas de por qué existe el GIL están más allá del alcance de este libro. Si bien es cierto que en muchas aplicaciones de procesamiento de big data, se puede requerir un grupo de computadoras para procesar un conjunto de datos en una cantidad de tiempo razonable, todavía hay situaciones en las que es deseable un sistema de proceso único y multiproceso.

Esto no quiere decir que Python no pueda ejecutar código paralelo verdaderamente multiproceso. Las extensiones de Python C que usan subprocesos múltiples nativos (en C o C++) pueden ejecutar código en paralelo sin verse afectadas por la GIL, siempre que no necesiten interactuar regularmente con objetos de Python.

1.3 Bibliotecas esenciales de Python

Para aquellos que están menos familiarizados con el ecosistema de datos de Python y las bibliotecas utilizadas a lo largo del libro, daré una breve descripción general de algunas de ellas.

NumPy

NumPy, abreviatura de Numerical Python, ha sido durante mucho tiempo la piedra angular de la computación numérica en Python. Proporciona las estructuras de datos, los algoritmos y el pegamento de biblioteca necesarios para la mayoría de las aplicaciones científicas que involucran datos numéricos en Python. NumPy contiene, entre otras cosas:

- Un objeto de matriz multidimensional rápido y eficiente ndarray • Funciones para realizar cálculos de elementos con matrices o matemáticas operaciones cal entre arreglos
- Herramientas para leer y escribir conjuntos de datos basados en matrices en el disco • Operaciones de álgebra lineal, transformada de Fourier y generación de números aleatorios • Una API de C madura para permitir el acceso a extensiones de Python y código C o C++ nativo Estructuras de datos y facilidades computacionales de NumPy

Más allá de las capacidades de procesamiento rápido de arreglos que NumPy agrega a Python, uno de sus usos principales en el análisis de datos es como un contenedor para que los datos pasen entre algoritmos y bibliotecas. Para datos numéricos, las matrices NumPy son más eficientes para almacenar y manipular datos que las otras estructuras de datos integradas de Python. Además, las bibliotecas escritas en un lenguaje de nivel inferior, como C o Fortran, pueden operar con los datos almacenados en una matriz NumPy sin copiar los datos en alguna otra representación de memoria.

Por lo tanto, muchas herramientas de computación numérica para Python asumen arreglos NumPy como una estructura de datos principal o apuntan a una interoperabilidad perfecta con NumPy.

pandas

pandas proporciona funciones y estructuras de datos de alto nivel diseñadas para hacer que trabajar con datos estructurados o tabulares sea rápido, fácil y expresivo. Desde su aparición en 2010, ha ayudado a que Python sea un entorno de análisis de datos poderoso y productivo. Los objetos principales en pandas que se utilizarán en este libro son DataFrame, una estructura de datos tabular orientada a columnas con etiquetas de fila y columna, y Series, un objeto de matriz etiquetado unidimensional. pandas combina las ideas de computación de matriz de alto rendimiento de

NumPy con las capacidades flexibles de manipulación de datos de las hojas de cálculo y las bases de datos relacionales (como SQL). Proporciona una funcionalidad de indexación sofisticada para facilitar la remodelación, el corte y el corte, la realización de agregaciones y la selección de subconjuntos de datos. Desde la manipulación de datos,

la preparación y la limpieza son una habilidad tan importante en el análisis de datos que los pandas son uno de los enfoques principales de este libro.

Como antecedentes, comencé a construir pandas a principios de 2008 durante mi mandato en AQR Capital Management, una empresa de gestión de inversiones cuantitativas. En ese momento, tenía un conjunto distinto de requisitos que no estaban bien abordados por ninguna herramienta única a mi disposición:

- Estructuras de datos con ejes etiquetados que admiten la alineación automática o explícita de datos: esto evita errores comunes que resultan de datos desalineados y trabajar con datos indexados de manera diferente que provienen de diferentes fuentes.

- Funcionalidad integrada de series de tiempo •

Las mismas estructuras de datos manejan datos de series de tiempo y datos que no son de series de tiempo

- Operaciones aritméticas y reducciones que preservan metadatos • Manejo flexible

de datos faltantes • Fusión y otras operaciones

relacionales que se encuentran en bases de datos populares (basadas en SQL,

Por ejemplo)

Quería poder hacer todas estas cosas en un solo lugar, preferiblemente en un lenguaje adecuado para el desarrollo de software de propósito general. Python era un buen lenguaje candidato para esto, pero en ese momento no había un conjunto integrado de estructuras de datos y herramientas que proporcionaran esta funcionalidad. Como resultado de haber sido construido inicialmente para resolver problemas financieros y de análisis de negocios, pandas presenta una funcionalidad de serie de tiempo especialmente profunda y herramientas muy adecuadas para trabajar con datos indexados en el tiempo generados por procesos de negocios.

Para los usuarios del lenguaje R para computación estadística, el nombre de DataFrame les resultará familiar, ya que el objeto recibió el nombre del objeto R `data.frame` similar. A diferencia de Python, los marcos de datos están integrados en el lenguaje de programación R y su biblioteca estándar. Como resultado, muchas funciones que se encuentran en pandas suelen ser parte de la implementación central de R o proporcionadas por paquetes complementarios.

El nombre de pandas en sí se deriva de datos de panel, un término econométrico para conjuntos de datos estructurados multidimensionales, y un juego con la frase Análisis de datos de Python en sí.

matplotlib

matplotlib es la biblioteca de Python más popular para producir gráficos y otras visualizaciones de datos bidimensionales. Originalmente fue creado por John D. Hunter y ahora lo mantiene un gran equipo de desarrolladores. Está diseñado para crear tramas adecuadas para su publicación. Si bien existen otras bibliotecas de visualización disponibles para los programadores de Python, matplotlib es la más utilizada y, como tal, generalmente tiene una buena inter

gratificación con el resto del ecosistema. Creo que es una opción segura como herramienta de visualización predeterminada.

IPython y Jupyter El **proyecto**

IPython comenzó en 2001 como un proyecto paralelo de Fernando Pérez para crear un mejor intérprete interactivo de Python. En los siguientes 16 años se ha convertido en una de las herramientas más importantes en la pila de datos moderna de Python. Si bien no proporciona ninguna herramienta informática o de análisis de datos por sí mismo, IPython está diseñado desde cero para maximizar su productividad tanto en la informática interactiva como en el desarrollo de software. Fomenta un flujo de trabajo de ejecución y exploración en lugar del típico flujo de trabajo de ejecución de edición y compilación de muchos otros lenguajes de programación. También proporciona fácil acceso al shell y al sistema de archivos de su sistema operativo. Dado que gran parte de la codificación del análisis de datos implica exploración, prueba y error e iteración, IPython puede ayudarlo a realizar el trabajo más rápido.

En 2014, Fernando y el equipo de IPython anunciaron el **proyecto Jupyter**, una iniciativa más amplia para diseñar herramientas informáticas interactivas independientes del lenguaje. El cuaderno web IPython se convirtió en el cuaderno Jupyter, con soporte ahora para más de 40 lenguajes de programación. El sistema IPython ahora se puede usar como kernel (un modo de lenguaje de programación) para usar Python con Jupyter.

IPython se ha convertido en un componente del proyecto de código abierto mucho más amplio Jupyter, que proporciona un entorno productivo para la computación interactiva y exploratoria. Su "modo" más antiguo y simple es como un shell de Python mejorado diseñado para acelerar la escritura, prueba y depuración del código de Python. También puede usar el sistema IPython a través de Jupyter Notebook, un "cuaderno de notas" de código interactivo basado en la web que ofrece soporte para docenas de lenguajes de programación. Los cuadernos IPython shell y Jupyter son especialmente útiles para la exploración y visualización de datos.

El sistema de cuadernos Jupyter también le permite crear contenido en Markdown y HTML, lo que le brinda un medio para crear documentos enriquecidos con código y texto. Otros lenguajes de programación también han implementado kernels para Jupyter para permitirle usar lenguajes distintos a Python en Jupyter.

Para mí, IPython suele estar involucrado con la mayoría de mi trabajo de Python, incluida la ejecución, la depuración y la prueba del código.

En los **materiales del libro adjunto**, encontrará cuadernos Jupyter que contienen todos los ejemplos de código de cada capítulo.

SciPy

SciPy es una colección de paquetes que abordan varios dominios de problemas estándar diferentes en computación científica. Aquí hay una muestra de los paquetes incluidos:

scipy.integrate Rutinas

de integración numérica y solucionadores de ecuaciones diferenciales

scipy.linalg Rutinas

de álgebra lineal y descomposiciones de matrices que se extienden más allá de las proporcionadas en numpy.linalg

scipy.optimize

Optimizadores de funciones (minimizadores) y algoritmos de búsqueda de raíces

scipy.signal

Herramientas de procesamiento de señales

scipy.sparse

Matrices dispersas y solucionadores de sistemas lineales dispersos

scipy.special

Wrapper alrededor de SPECFUN, una biblioteca de Fortran que implementa muchas funciones matemáticas comunes, como la función gamma

scipy.stats

Distribuciones de probabilidad discretas y continuas estándar (funciones de densidad, muestreadores, funciones de distribución continua), varias pruebas estadísticas y estadísticas más descriptivas

Juntos, NumPy y SciPy forman una base computacional razonablemente completa y madura para muchas aplicaciones informáticas científicas tradicionales.

scikit-learn

Desde el inicio del proyecto en 2010, **scikit-learn** se ha convertido en el principal conjunto de herramientas de aprendizaje automático de uso general para programadores de Python. En solo siete años, ha tenido más de 1.500 colaboradores de todo el mundo. Incluye submódulos para modelos como:

- Clasificación: SVM, vecinos más cercanos, bosque aleatorio, regresión logística, etc.
- Regresión: Lasso, regresión de cresta, etc.
- Agrupamiento: k-medias, agrupamiento espectral, etc.
- Reducción de dimensionalidad: PCA, selección de características, factorización de matrices, etc.
- Selección de modelos: búsqueda en grillas, validación cruzada, métricas
- Preprocesamiento: extracción de características, normalización

Junto con pandas, statsmodels e IPython, scikit-learn ha sido fundamental para permitir que Python sea un lenguaje de programación de ciencia de datos productivo. Mientras no lo haré

poder incluir una guía completa de scikit-learn en este libro, daré una breve introducción a algunos de sus modelos y cómo usarlos con las otras herramientas presentadas en el libro.

modelos estadísticos

statsmodels es un paquete de análisis estadístico que fue iniciado por el trabajo del profesor de estadística de la Universidad de Stanford, Jonathan Taylor, quien implementó una serie de modelos de análisis de regresión populares 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 alcanzar una masa crítica de usuarios y colaboradores comprometidos. Nathaniel Smith desarrolló el proyecto Patsy, que proporciona un marco de especificación de fórmulas o modelos para modelos estadísticos inspirados en el sistema de fórmulas de R.

En comparación con scikit-learn, statsmodels contiene algoritmos para estadísticas y econometría clásicas (principalmente frecuentistas). Esto incluye submódulos como:

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

statsmodels se centra más en la inferencia estadística, proporcionando estimaciones de incertidumbre y valores p para los parámetros. scikit-learn, por el contrario, se centra más en la predicción.

Al igual que con scikit-learn, daré una breve introducción a statsmodels y cómo usarlo con NumPy y pandas.

1.4 Instalación y configuración

Dado que todos usan Python para diferentes aplicaciones, no existe una solución única para configurar Python y los paquetes complementarios necesarios. Muchos lectores no tendrán un entorno de desarrollo de Python completo adecuado para seguir este libro, por lo que aquí daré instrucciones detalladas para configurar cada sistema operativo. Recomiendo usar la distribución gratuita de Anaconda. En el momento de escribir este artículo, Anaconda se ofrece en las formas Python 2.7 y 3.6, aunque esto podría cambiar en algún momento en el futuro. Este libro usa Python 3.6, y lo animo a usar Python 3.6 o superior.

ventanas

Para comenzar en Windows, descargue el [instalador de Anaconda](#). Recomiendo seguir las instrucciones de instalación para Windows disponibles en la página de descarga de Anaconda, que pueden haber cambiado entre el momento en que se publicó este libro y el momento en que lo está leyendo.

Ahora, verifiquemos que las cosas estén configuradas correctamente. Para abrir la aplicación Símbolo del sistema (también conocida como cmd.exe), haga clic con el botón derecho en el menú Inicio y seleccione Símbolo del sistema. Intente iniciar el intérprete de Python escribiendo python. Debería ver un mensaje que coincida con la versión de Anaconda que instaló:

```
C:\Usuarios\wesm>python
Python 3.5.2 [Anaconda 4.1.1 (64 bits)] (predeterminado, 5 de julio de 2016, 11:41:13)
[MSC v.1900 de 64 bits (AMD64)] en win32
>>>
```

Para salir del shell, presione Ctrl-D (en Linux o macOS), Ctrl-Z (en Windows), o escriba el comando exit() y presione Entrar.

Apple (OS X, mac OS)

Descargue el instalador de OS X Anaconda, que debería tener un nombre similar a Anaconda3-4.1.0-MacOSX-x86_64.pkg. Haga doble clic en el archivo .pkg para ejecutar el instalador.

Cuando se ejecuta el instalador, agrega automáticamente la ruta del ejecutable de Anaconda a su archivo .bash_profile. Este se encuentra en /Users/\$USER/.bash_profile.

Para verificar que todo funcione, intente iniciar IPython en el shell del sistema (abra la aplicación Terminal para obtener un símbolo del sistema):

```
$ ipython
```

Para salir del shell, presione Ctrl-D o escriba exit() y presione Enter.

GNU/Linux

Los detalles de Linux variarán un poco dependiendo de su versión de Linux, pero aquí doy detalles para distribuciones como Debian, Ubuntu, CentOS y Fedora. La configuración es similar a OS X, con la excepción de cómo se instala Anaconda. El instalador es un script de shell que debe ejecutarse en la terminal. Dependiendo de si tiene un sistema de 32 bits o de 64 bits, deberá instalar el instalador x86 (32 bits) o x86_64 (64 bits). Luego tendrá un archivo con un nombre similar a Anaconda3-4.1.0-Linux-x86_64.sh.

Para instalarlo, ejecute este script con bash:

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```



Algunas distribuciones de Linux tienen versiones de todos los paquetes de Python necesarios en sus administradores de paquetes y se pueden instalar con una herramienta como apt. La configuración que se describe aquí utiliza Anaconda, ya que es fácilmente reproducible en todas las distribuciones y es más sencillo actualizar los paquetes a sus últimas versiones.

Después de aceptar la licencia, se le presentará la opción de dónde colocar los archivos de Anaconda. Recomendando instalar los archivos en la ubicación predeterminada en su directorio de inicio, por ejemplo, /home/\$USUARIO/anaconda (con su nombre de usuario, naturalmente).

El instalador de Anaconda puede preguntarle si desea anteponer su directorio bin/ a su variable \$PATH . Si tiene algún problema después de la instalación, puede hacerlo usted mismo modificando su .bashrc (o .zshrc, si está usando el shell zsh) con algo similar a:

```
exportar RUTA=/inicio/$USUARIO/anaconda/bin:$RUTA
```

Después de hacer esto, puede iniciar un nuevo proceso de terminal o ejecutar su .bashrc nuevamente con source ~/.bashrc.

Instalación o actualización de paquetes de Python En

algún momento durante la lectura, es posible que desee instalar paquetes de Python adicionales que no están incluidos en la distribución de Anaconda. En general, estos se pueden instalar con el siguiente comando:

```
conda install nombre_paquete
```

Si esto no funciona, también puede instalar el paquete usando la herramienta de administración de paquetes pip:

```
pip install nombre_paquete
```

Puede actualizar paquetes usando el comando de actualización conda :

```
actualización de conda nombre_del_paquete
```

pip también admite actualizaciones usando el indicador --upgrade :

```
pip install --upgrade nombre_paquete
```

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



Si bien puede usar conda y pip para instalar paquetes, no debe intentar actualizar los paquetes conda con pip, ya que hacerlo puede generar problemas ambientales. Cuando utilice Anaconda o Miniconda, es mejor intentar primero actualizar con conda.

Python 2 y Python 3 La

primera versión de la línea de intérpretes Python 3.x se lanzó a finales de 2008.

Incluyó una serie de cambios que hicieron que algunos códigos de Python 2.x previamente escritos fueran incompatibles. Debido a que habían pasado 17 años desde el primer lanzamiento de Python en 1991, se consideró que la creación de un lanzamiento "innovador" de Python 3 era un bien mayor dadas las lecciones aprendidas durante ese tiempo.

En 2012, gran parte de la comunidad científica y de análisis de datos todavía usaba Python 2.x porque muchos paquetes no se habían hecho completamente compatibles con Python 3. Por lo tanto, la primera edición de este libro utilizó Python 2.7. Ahora, los usuarios son libres de elegir entre Python 2.x y 3.x y, en general, tienen soporte completo de biblioteca con cualquiera de los dos tipos.

Sin embargo, Python 2.x llegará al final de su vida útil de desarrollo en 2020 (incluidos los parches de seguridad críticos), por lo que ya no es una buena idea comenzar nuevos proyectos en Python 2.7. Por lo tanto, este libro utiliza Python 3.6, una versión estable ampliamente implementada y bien respaldada. Hemos comenzado a llamar a Python 2.x "Legacy Python" y a Python 3.x simplemente "Python". Yo te animo a que hagas lo mismo.

Este libro utiliza Python 3.6 como base. Su versión de Python puede ser más reciente que la 3.6, pero los ejemplos de código deben ser compatibles con versiones anteriores. Algunos ejemplos de código pueden funcionar de manera diferente o no funcionar en absoluto en Python 2.7.

Entornos de desarrollo integrados (IDE) y editores de texto Cuando me preguntan sobre mi entorno de desarrollo estándar, casi siempre digo "IPython más un editor de texto". Por lo general, escribo un programa y pruebo y depuro iterativamente cada parte en cuadernos IPython o Jupyter. También es útil poder jugar con los datos de forma interactiva y verificar visualmente que un conjunto particular de manipulaciones de datos está haciendo lo correcto. Bibliotecas como pandas y NumPy están diseñadas para ser fáciles de usar en el shell.

Sin embargo, al crear software, algunos usuarios pueden preferir usar un IDE con más funciones en lugar de un editor de texto comparativamente primitivo como Emacs o Vim. Aquí hay algunos que puedes explorar:

- PyDev (gratis), un IDE integrado en la plataforma Eclipse • PyCharm de JetBrains (basado en suscripción para usuarios comerciales, gratuito para desarrolladores de código abierto)
- Python Tools para Visual Studio (para usuarios de Windows) • Spyder (gratis), un IDE que actualmente se envía con Anaconda • Komodo IDE (comercial)

Debido a la popularidad de Python, la mayoría de los editores de texto, como Atom y Sublime Text 2, tienen una excelente compatibilidad con Python.

1.5 Comunidad y Conferencias

Fuera de una búsqueda en Internet, las diversas listas de correo de Python científicas y relacionadas con datos son generalmente útiles y responden a las preguntas. Algunos para echar un vistazo incluyen:

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

Deliberadamente no publiqué URL para estos en caso de que cambien. Se pueden localizar fácilmente a través de una búsqueda en Internet.

Cada año se llevan a cabo muchas conferencias en todo el mundo para programadores de Python. Si desea conectarse con otros programadores de Python que comparten sus intereses, lo animo a explorar la posibilidad de asistir a uno, si es posible. Muchas conferencias tienen apoyo financiero disponible para aquellos que no pueden pagar la entrada o el viaje a la conferencia. Aquí hay algunos a considerar:

- PyCon y EuroPython: Las dos principales conferencias generales de Python en North América y Europa, respectivamente
- SciPy y EuroSciPy: conferencias orientadas a la computación científica en América del Norte y Europa, respectivamente
- PyData: una serie mundial de conferencias regionales dirigidas a la ciencia de datos y casos de uso de análisis de datos
- Conferencias internacionales y regionales de PyCon (ver <http://pycon.org> para una lista completa)

1.6 Navegando por este libro

Si nunca antes ha programado en Python, querrá pasar algún tiempo en los Capítulos 2 y 3, donde he colocado un tutorial condensado sobre las características del lenguaje Python y el shell IPython y los cuadernos Jupyter. Estas cosas son un requisito previo

conocimiento para el resto del libro. Si ya tiene experiencia con Python, puede optar por hojear u omitir estos capítulos.

A continuación, doy una breve introducción a las características clave 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 aplicando pandas, NumPy y matplotlib (para visualización). He estructurado el material de la manera más incremental posible, aunque ocasionalmente hay algunos cruces menores entre capítulos, con algunos casos aislados en los que se utilizan conceptos que aún no se han introducido necesariamente.

Si bien los lectores pueden tener muchos objetivos finales diferentes para su trabajo, las tareas requeridas generalmente se dividen en varios grupos amplios diferentes:

Interactuando con el mundo exterior

Leer y escribir con una variedad de formatos de archivo y almacenes de datos

Preparación

Limpiar, mezclar, combinar, normalizar, remodelar, rebanar y trocear, y transformar datos para su análisis

Transformación

Aplicar operaciones matemáticas y estadísticas a grupos de conjuntos de datos para derivar nuevos conjuntos de datos (p. ej., agregar una tabla grande por variables de grupo)

Modelado y computación

Conexión de sus datos a modelos estadísticos, algoritmos de aprendizaje automático u otras herramientas informáticas

Presentación

Crear visualizaciones gráficas estáticas o interactivas o resúmenes textuales

Ejemplos de código

La mayoría de los ejemplos de código en el libro se muestran con entrada y salida tal y como aparecería ejecutado en el shell de IPython o en los cuadernos de Jupyter:

```
En [5]: CÓDIGO EJEMPLO
Salida[5]: SALIDA
```

Cuando vea un ejemplo de código como este, la intención es que escriba el código de ejemplo en el bloque de entrada en su entorno de codificación y lo ejecute presionando la tecla Intro (o Shift-Enter en Jupyter). Debería ver un resultado similar al que se muestra en el bloque Salida .

Datos para ejemplos Los

conjuntos de datos para los ejemplos de cada capítulo están alojados en [un repositorio de GitHub](#). Puede descargar estos datos utilizando el sistema de control de versiones de Git en el comando

línea o descargando un archivo zip del repositorio desde el sitio web. Si tiene problemas, navegue a [mi sitio web](#) para obtener instrucciones actualizadas sobre cómo obtener los materiales del libro.

He hecho todo lo posible para asegurarme de que contenga todo lo necesario para reproducir los ejemplos, pero es posible que haya cometido algunos errores u omisiones. Si es así, envíeme un correo electrónico: book@wesmckinney.com. La mejor manera de informar errores en el libro es en la [página de erratas en el sitio web de O'Reilly](#).

Convenciones de importación

La comunidad de Python ha adoptado una serie de convenciones de nomenclatura para los módulos de uso común:

```
importar numpy como
np importar matplotlib.pyplot como plt
importar pandas como
pd importar seaborn como
sns importar statsmodels como sm
```

Esto significa que cuando ve `np.arange`, esta es una referencia a la función `arange` en NumPy. Esto se hace porque se considera una mala práctica en el desarrollo de software de Python importar todo (desde `numpy import *`) desde un paquete grande como NumPy.

Jerga

Usaré algunos términos comunes tanto a la programación como a la ciencia de datos con los que quizás no esté familiarizado. Por lo tanto, aquí hay algunas definiciones breves:

Munge/munging/disputas

Describe el proceso general de manipulación de datos no estructurados y/o desordenados en una forma estructurada o limpia. La palabra se ha colado en la jerga de muchos hackers de datos modernos. “Munge” rima con “grunge”.

pseudocódigo

Una descripción de un algoritmo o proceso que toma una forma similar a un código aunque probablemente no sea un código fuente válido real.

Azúcar sintáctico

Sintaxis de programación que no agrega nuevas funciones, pero hace que algo sea más conveniente o más fácil de escribir.

Conceptos básicos del lenguaje Python, IPython y Cuadernos Jupyter

Cuando escribí la primera edición de este libro en 2011 y 2012, había menos recursos disponibles para aprender a hacer análisis de datos en Python. Esto fue en parte un problema del huevo y la gallina; muchas bibliotecas que ahora damos por sentadas, como pandas, scikit-learn y statsmodels, eran comparativamente inmaduras en ese entonces. En 2017, ahora hay una creciente literatura sobre ciencia de datos, análisis de datos y aprendizaje automático, que complementa los trabajos anteriores sobre computación científica de propósito general dirigidos a científicos computacionales, físicos y profesionales en otros campos de investigación. También hay excelentes libros sobre cómo aprender el lenguaje de programación Python y convertirse en un ingeniero de software efectivo.

Como este libro pretende ser un texto introductorio para trabajar con datos en Python, creo que es valioso tener una descripción general independiente de algunas de las características más importantes de las estructuras de datos y bibliotecas integradas de Python desde la perspectiva de manipulación de datos. Por lo tanto, en este capítulo y en [el Capítulo 3](#) solo presentaré la información suficiente para permitirle continuar con el resto del libro.

En mi opinión, no es necesario volverse competente en la creación de un buen software en Python para poder realizar análisis de datos de manera productiva. Lo animo a usar el shell IPython y los cuadernos Jupyter para experimentar con los ejemplos de código y explorar la documentación de los diversos tipos, funciones y métodos. Si bien hice todo lo posible para presentar el material del libro de forma incremental, es posible que ocasionalmente encuentre cosas que aún no se han presentado por completo.

Gran parte de este libro se centra en el análisis basado en tablas y las herramientas de preparación de datos para trabajar con grandes conjuntos de datos. Para usar esas herramientas, a menudo primero debe hacer algo de munging para acorralar los datos desordenados en una forma tabular (o estructurada) más agradable. Afortunadamente, Python es un lenguaje ideal para darle forma rápidamente a sus datos. El

cuanto mayor sea su facilidad con el lenguaje Python, más fácil le resultará preparar nuevos conjuntos de datos para el análisis.

Algunas de las herramientas de este libro se exploran mejor desde una sesión en vivo de IPython o Jupyter. Una vez que aprendas a poner en marcha IPython y Jupyter, te recomiendo que sigas los ejemplos para que puedas experimentar y probar cosas diferentes. Al igual que con cualquier entorno similar a una consola controlado por teclado, desarrollar memoria muscular para los comandos comunes también es parte de la curva de aprendizaje.



Hay conceptos introductorios de Python que este capítulo no cubre, como clases y programación orientada a objetos, que pueden resultarle útiles en su incursión en el análisis de datos en Python.

Para profundizar su conocimiento del lenguaje Python, le recomiendo que complemente este capítulo con el [tutorial oficial de Python](#) y, potencialmente, uno de los muchos libros excelentes sobre programación Python de propósito general. Algunas recomendaciones para comenzar incluyen:

- Libro de cocina de Python, tercera edición, de David Beazley y Brian K. Jones
- (O'Reilly) • Python fluido de Luciano Ramalho
- (O'Reilly) • Python eficaz de Brett Slatkin (Pearson)

2.1 El intérprete de Python

Python es un lenguaje interpretado. El intérprete de Python ejecuta un programa ejecutando una declaración a la vez. El intérprete de Python interactivo estándar se puede invocar en la línea de comando con el comando `python` :

```
$ pitón
Pitón 3.6.0 | empaquetado por conda-forge | (predeterminado, 13 de enero de 2017, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] en Linux Escriba
"ayuda", "derechos de autor", "créditos" o "licencia" para obtener más información. >>> un = 5

>>> imprime(a)
5
```

El `>>>` que ves es el indicador donde escribirás expresiones de código. Para salir del intérprete de Python y volver al símbolo del sistema, puede escribir `exit()` o presionar `Ctrl-D`.

Ejecutar programas de Python es tan simple como llamar a Python con un archivo `.py` como primer argumento. Supongamos que hubiéramos creado `hello_world.py` con estos contenidos:

```
imprimir('Hola mundo')
```

Puede ejecutarlo ejecutando el siguiente comando (el archivo `hello_world.py` debe estar en el directorio de su terminal de trabajo actual):

```
$ python hola_mundo.py Hola
mundo
```

Mientras que algunos programadores de Python ejecutan todo su código de Python de esta manera, aquellos que realizan análisis de datos o computación científica utilizan IPython, un intérprete de Python mejorado, o cuadernos de Jupyter, cuadernos de código basados en la web creados originalmente dentro del proyecto IPython. Ofrezco una introducción al uso de IPython y Jupyter en este capítulo y he incluido una mirada más profunda a la funcionalidad de IPython en el [Apéndice A](#). Cuando usa el comando `%run`, IPython ejecuta el código en el archivo especificado en el mismo proceso, lo que le permite explorar los resultados de forma interactiva cuando termina:

```
$ ipython
Python 3.6.0 | empaquetado por conda-forge | (predeterminado, 13 de enero de 2017, 23:17:12)
Escriba "derechos de autor", "créditos" o "licencia" para obtener más información.
```

```
IPython 5.1.0: un Python interactivo mejorado.
?                -> Introducción y descripción general de las características de IPython .
%quickref -> Referencia rápida. ayuda
-> Sistema de ayuda propio de Python . ¿objeto? ->
Detalles sobre 'objeto', use 'objeto?' para detalles adicionales.
```

```
En [1]: %run hello_world.py Hola
mundo
```

```
En [2]:
```

El indicador predeterminado de IPython adopta el estilo numerado `In [2]:` en comparación con el indicador `>>>` estándar .

2.2 Conceptos básicos de IPython

En esta sección, lo pondremos en marcha con el shell IPython y el cuaderno Jupyter, y le presentaremos algunos de los conceptos esenciales.

Ejecución de IPython Shell

Puede iniciar el shell de IPython en la línea de comando al igual que iniciar el regular Intérprete de Python excepto con el comando `ipython` :

```
$ ipython
Python 3.6.0 | empaquetado por conda-forge | (predeterminado, 13 de enero de 2017, 23:17:12)
Escriba "derechos de autor", "créditos" o "licencia" para obtener más información.
```

```
IPython 5.1.0: un Python interactivo mejorado. ?
-> Introducción y descripción general de las características de IPython.
%quickref -> Referencia rápida. ayuda
-> Sistema de ayuda propio de Python.
```

¿objeto? -> Detalles sobre 'objeto', use 'objeto?' para detalles adicionales.

En [1]: a = 5

En [2]: un

Salida[2]: 5

Puede ejecutar declaraciones de Python arbitrarias escribiéndolas y presionando Retorno (o Intro). Cuando escribe solo una variable en IPython, muestra una representación de cadena del objeto:

En [5]: `importar numpy como np`

En [6]: `datos = {i : np.random.randn() para i en el rango (7)}`

Entrada [7]:

salida

de datos [7]: {0:

-0.20470765948471295, 1:

0.47894333805754824, 2:

-0.5194387150567381 ,

3: -0.55573030434749, 4:

1.9657805725027 142, 5:

1,3934058329729904, 6: 0,09290787674371767}

Las primeras dos líneas son declaraciones de código de Python; la segunda declaración crea una variable con nombre `data` que se refiere a un diccionario de Python recién creado. La última línea imprime el valor de los datos en la consola.

Muchos tipos de objetos de Python están formateados para que sean más legibles, o bien impresos, lo cual es distinto de la impresión normal con `print`. Si imprimió la variable de datos anterior en el intérprete estándar de Python, sería mucho menos legible:

```
>>> from numpy.random import randn >>>
```

```
data = {i : randn() for i in range(7)} >>> print(data)
```

```
{0:
```

```
-1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295, 3 :
```

```
0,15455217573074576, 4: -0,24058577449429575, 5: -1,2904897053651216, 6:
```

```
0,3308507317325902}
```

IPython también proporciona funciones para ejecutar bloques arbitrarios de código (a través de un enfoque algo glorificado de copiar y pegar) y secuencias de comandos completas de Python. También puede usar el cuaderno Jupyter Notebook para trabajar con bloques de código más grandes, como veremos pronto.

Ejecución de Jupyter Notebook Uno de los

principales componentes del proyecto Jupyter es el notebook, un tipo de documento interactivo para código, texto (con o sin marcado), visualizaciones de datos y otros resultados. El cuaderno Jupyter interactúa con los núcleos, que son implementaciones de

el protocolo de computación interactiva Jupyter en cualquier número de lenguajes de programación. El kernel Jupyter de Python utiliza el sistema IPython para su comportamiento subyacente.

Para iniciar Jupyter, ejecute el comando `jupyter notebook` en una terminal:

```
$ jupyter notebook [I
15:20:52.739 NotebookApp] Sirviendo portátiles desde el directorio local: /home/wesm/code/
pydata-book [I 15:20:52.739
NotebookApp] 0 núcleos activos [I 15:20:52.739 NotebookApp]
El Jupyter Notebook se ejecuta en: http://localhost:8888/ [I 15:20:52.740 NotebookApp]
Use Control-C para detener
este servidor y cerrar todos los núcleos (dos veces para omitir la confirmación).
```

Se creó una nueva ventana en la sesión del navegador existente.

En muchas plataformas, Jupyter se abrirá automáticamente en su navegador web predeterminado (a menos que lo inicie con `--no-browser`). De lo contrario, puede navegar a la dirección HTTP impresa cuando inició el portátil, aquí <http://localhost:8888/>. Consulte la [Figura 2-1](#) para ver cómo se ve esto en Google Chrome.



Mucha gente usa Jupyter como un entorno informático local, pero también se puede implementar en servidores y acceder de forma remota. No cubriré esos detalles aquí, pero lo animo a explorar este tema en Internet si es relevante para sus necesidades.

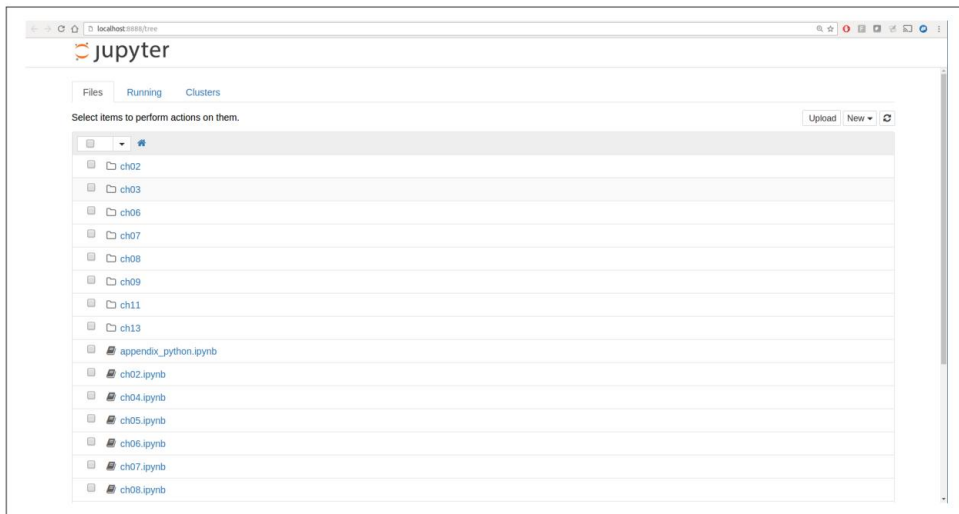


Figura 2-1. Página de destino del cuaderno Jupyter

Para crear un nuevo cuaderno, haga clic en el botón Nuevo y seleccione la opción "Python 3" o "conda [predeterminado]". Debería ver algo como [la Figura 2-2](#). Si es la primera vez, intente hacer clic en la "celda" de código vacía e ingrese una línea de código Python. Luego presione Shift-Enter para ejecutarlo.

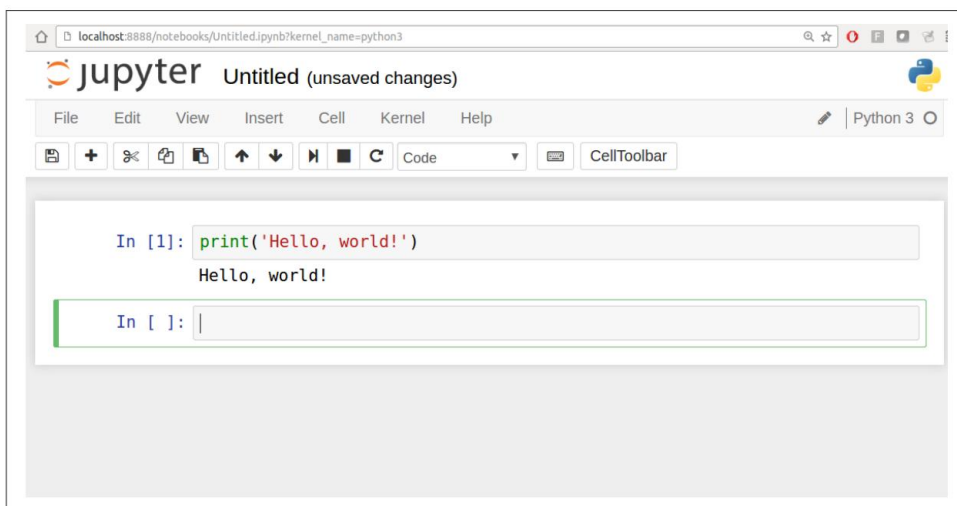


Figura 2-2. Nueva vista de notebook de Jupyter

Cuando guarda el cuaderno (consulte "Guardar y punto de control" en el menú Archivo del cuaderno), se crea un archivo con la extensión .ipynb. Este es un formato de archivo independiente que contiene todo el contenido (incluido cualquier resultado de código evaluado) actualmente en el cuaderno. Estos pueden ser cargados y editados por otros usuarios de Jupyter. Para cargar una libreta existente, coloque el archivo en el mismo directorio donde inició el proceso de la libreta (o en una subcarpeta dentro de este), luego haga doble clic en el nombre de la página de destino. Puede probarlo con los cuadernos de mi repositorio wesm/pydata-book en GitHub.

Consulte [la Figura 2-3](#).

Si bien el cuaderno de Jupyter puede sentirse como una experiencia distinta del shell de IPython, casi todos los comandos y herramientas de este capítulo se pueden usar en cualquier entorno.

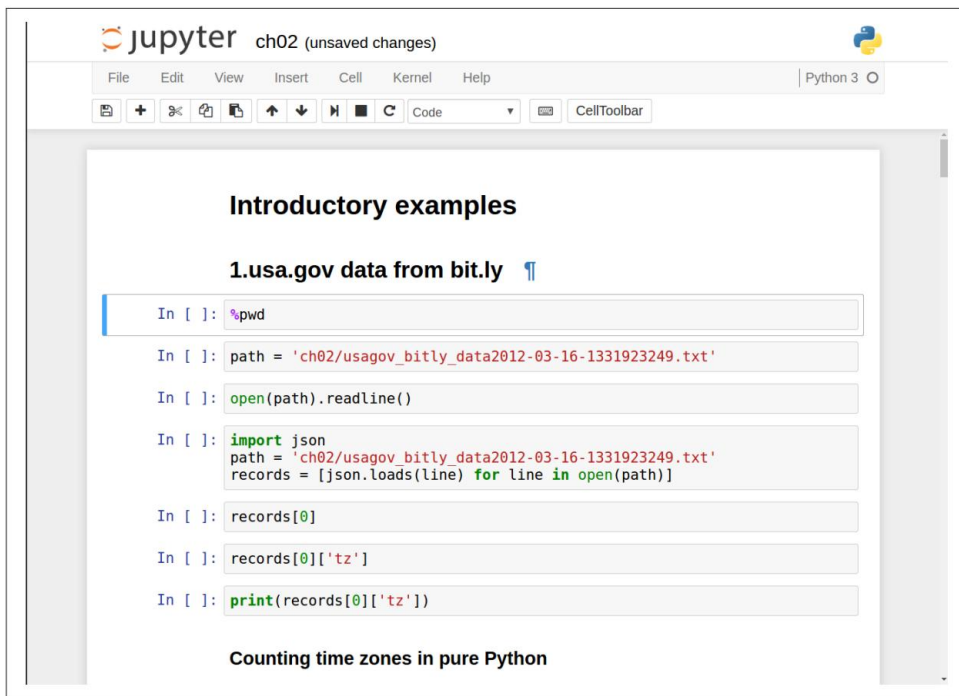


Figura 2-3. Vista de ejemplo de Jupyter para un cuaderno existente

Finalización de

pestañas En la superficie, el shell de IPython parece una versión cosméticamente diferente del intérprete estándar de terminal de Python (invocado con `python`). Una de las principales mejoras con respecto al shell estándar de Python es la finalización con pestañas, que se encuentra en muchos IDE u otros entornos de análisis informático interactivo. Mientras ingresa expresiones en el shell, al presionar la tecla `Tab` buscará en el espacio de nombres cualquier variable (objetos, funciones, etc.) que coincida con los caracteres que ha escrito hasta ahora:

En [1]: `una_manzana = 27`

En [2]: `un_ejemplo = 42`

En [3]: `an<Tab>`
`an_apple` y `un_ejemplo cualquiera`

En este ejemplo, tenga en cuenta que IPython mostró las dos variables que definí, así como la palabra clave de Python y la función incorporada `any`. Naturalmente, también puede completar métodos y atributos en cualquier objeto después de escribir un punto:

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

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

`b.agregar b.contar b.insertar b.revertir b.borrar`
`b.extender b.pop b.copiar` `b.clasificar`
`b.indexar b.eliminar`

Lo mismo ocurre con los módulos:

En [1]: `importar fecha y hora`

En [2]: `fechahora.<Tab>`

`fechahora.fecha` `fechahora.MAXYEAR` `fechahora.timedelta`
`datetime.datetime datetime.MINYEAR` `datetime.timezone`
`datetime.datetime_CAPI datetime.time` `datetime.tzinfo`

En el cuaderno Jupyter y en las versiones más recientes de IPython (5.0 y posteriores), las funciones de finalización automática se muestran en un cuadro desplegable en lugar de como salida de texto.



Tenga en cuenta que IPython oculta de forma predeterminada los métodos y atributos que comienzan con guiones bajos, como los métodos mágicos y los métodos y atributos internos "privados", para evitar saturar la pantalla (¡y confundir a los usuarios novatos!). Estos también se pueden completar con tabuladores, pero primero debe escribir un guión bajo para verlos. Si prefiere ver siempre dichos métodos en la finalización de pestañas, puede cambiar esta configuración en la configuración de IPython. Consulte la documentación de IPython para averiguar cómo hacerlo.

La finalización de pestañas funciona en muchos contextos fuera de la búsqueda en el espacio de nombres interactivo y la finalización de atributos de objetos o módulos. Al escribir cualquier cosa que parezca una ruta de archivo (incluso en una cadena de Python), presionar la tecla Tab completará cualquier cosa en el sistema de archivos de su computadora que coincida con lo que ha escrito:

En [7]: `datasets/movielens.<Tab>` `datasets/`
`movielens/movies.dat datasets/movielens/README datasets/movielens/`
`ratings.dat datasets/movielens/users.dat`

En [7]: `ruta = 'datasets/movielens.<Tab>` `datasets/`
`movielens/movies.dat datasets/movielens/README datasets/movielens/`
`ratings.dat datasets/movielens/users.dat`

En combinación con el comando `%run` (consulte ["El comando %run" en la página 25](#)), esta funcionalidad puede ahorrarle muchas pulsaciones de teclas.

Otra área en la que la finalización con tabulaciones ahorra tiempo es la finalización de los argumentos de palabras clave de función (¡e incluir el signo `=`!). Consulte [la Figura 2-4](#).

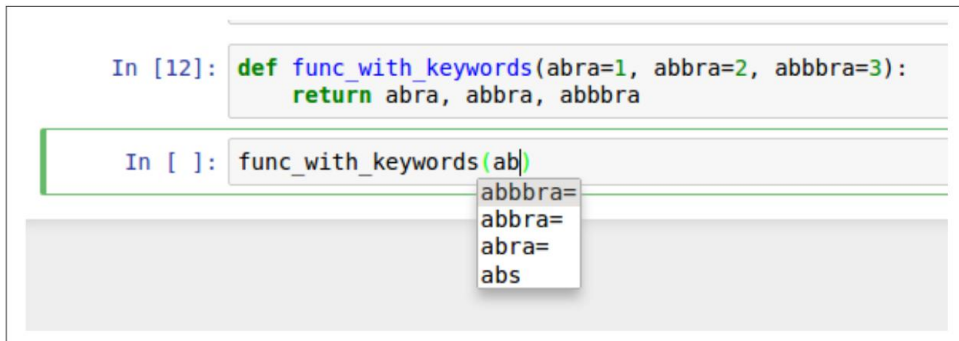


Figura 2-4. Palabras clave de función de autocompletar en el cuaderno Jupyter

Echaremos un vistazo más de cerca a las funciones en un momento.

Introspección El uso

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

```
En [8]: b = [1, 2, 3]
```

```
En [9]: b?
```

Tipo: lista Cadena

Forma:[1, 2, 3]

Longitud: 3

Docstring:

list () -> nueva lista de lista vacía

(iterable) -> nueva lista inicializada a partir de elementos iterables

```
En [10]: imprimir?
```

Cadena de

documentación: imprimir(valor, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Imprime los valores en una secuencia o en sys.stdout de forma predeterminada.

Argumentos de palabras clave

opcionales: archivo: un objeto similar a un archivo (flujo); el valor predeterminado es el sys.stdout

actual . sep: cadena insertada entre valores, por defecto un espacio. final:

cadena añadida después del último valor, por defecto una nueva línea. flush: si se

debe enjuagar a la fuerza la corriente.

builtin_function_or_method Tipo:

Esto se conoce como introspección de objetos. Si el objeto es una función o un método de instancia, también se mostrará la cadena de documentación, si está definida. Supongamos que escribimos la siguiente función (que puedes reproducir en IPython o Jupyter):

```
def sumar_números(a, b):

    Sumar dos números juntos

    Devoluciones
    =====

    the_sum : tipo de argumentos

    devolver a + b
```

Entonces usando ? nos muestra el docstring:

```
En [11]: agregar_números?
Firma: add_numbers(a, b)
Docstring:
suma dos números juntos

Devoluciones
=====

the_sum : tipo de argumentos
Archivo: <ipython-input-9-6a548a216e27>
Tipo: función
```

usando ?? también mostrará el código fuente de la función si es posible:

```
En [12]: agregar_números??
Firma: add_numbers(a, b)
Fuente:
def add_numbers(a, b):

    Sumar dos números juntos

    Devoluciones
    =====

    the_sum : tipo de argumentos

    devolver a + b
Archivo: <ipython-input-9-6a548a216e27> función
Tipo:
```

? tiene un uso final, que es para buscar en el espacio de nombres de IPython de manera similar a la línea de comandos estándar de Unix o Windows. Varios caracteres combinados con el comodín

(*) mostrarán todos los nombres que coincidan con la expresión del comodín. Por ejemplo, podríamos obtener una lista de todas las funciones en el espacio de nombres NumPy de nivel superior que contenga carga:

```
En [13]: np.*cargar*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

El comando %ejecutar

Puede ejecutar cualquier archivo como un programa de Python dentro del entorno de su sesión de IPython usando el comando %run . Suponga que tiene el siguiente script simple almacenado en ipython_script_test.py:

```
def f(x, y, z): return (x
+ y) / z
```

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

```
resultado = f(a, b, c)
```

Puede ejecutar esto pasando el nombre de archivo a %run:

```
En [14]: %ejecutar ipython_script_test.py
```

La secuencia de comandos se ejecuta en un espacio de nombres vacío (sin importaciones u otras variables definidas), por lo que el comportamiento debería ser idéntico a ejecutar el programa en la línea de comandos mediante python script.py. Todas las variables (importaciones, funciones y globales) definidas en el archivo (hasta que se genere una excepción, si la hay) serán accesibles en el shell de IPython:

```
Entrada [15]: c
```

```
Salida [15]: 7,5
```

```
Entrada [16]:
```

```
resultado Salida[16]: 1.4666666666666666
```

Si una secuencia de comandos de Python espera argumentos de la línea de comandos (que se encuentran en sys.argv), estos se pueden pasar después de la ruta del archivo como si se ejecutaran en la línea de comandos.



Si desea dar acceso a un script a las variables ya definidas en el espacio de nombres interactivo de IPython, use %run -i en lugar de %run simple .

En el cuaderno de Jupyter, también puede usar la función mágica %load relacionada, que importa un script en una celda de código:

```
>>> %cargar ipython_script_test.py
```

```
def f(x, y, z): return (x
+ y) / z
```

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

```
resultado = f(a, b, c)
```

Interrumpir el código en

ejecución Al presionar Ctrl-C mientras se ejecuta cualquier código, ya sea un script a través de %run o un comando de ejecución prolongada, se generará una interrupción del teclado. Esto hará que casi todos los programas de Python se detengan de inmediato, excepto en ciertos casos inusuales.



Cuando una parte del código de Python ha llamado a algunos módulos de extensión compilados, presionar Ctrl-C no siempre hará que la ejecución del programa se detenga de inmediato. En tales casos, tendrá que esperar hasta que se devuelva el control al intérprete de Python o, en circunstancias más graves, finalizar por la fuerza el proceso de Python.

Ejecución de código desde el portapapeles Si

está utilizando el cuaderno Jupyter, puede copiar y pegar código en cualquier celda de código y ejecutarlo. También es posible ejecutar código desde el portapapeles en el shell de IPython. Suponga que tiene el siguiente código en alguna otra aplicación:

```
x = 5
y = 7
si x > 5:
    X += 1

y = 8
```

Los métodos más infalibles son las funciones mágicas %paste y %cpaste. %paste toma cualquier texto que esté en el portapapeles y lo ejecuta como un solo bloque en el shell:

```
En [17]: %pegar x
= 5 y
= 7 si
x > 5:
    X += 1

y = 8
## -- Fin del texto pegado --
```

%cpaste es similar, excepto que le brinda un aviso especial para pegar el código en:

```
En [18]: %cpaste
Pegar código; ingrese '--' solo en la línea para detener o use Ctrl-D. :x = 5 :y = 7 :si x >
5:

:    X += 1
:
```

```
: y = 8
:--
```

Con el bloque %cpaste , tienes la libertad de pegar tanto código como quieras antes de ejecutarlo. Puede decidir usar %cpaste para mirar el pegado código antes de ejecutarlo. Si pega accidentalmente el código incorrecto, puede romper del indicador %cpaste presionando Ctrl-C.

Atajos de teclado de terminal

IPython tiene muchos atajos de teclado para navegar por el aviso (que le resultará familiar). iar a los usuarios del editor de texto Emacs o el shell bash de Unix) e interactuar con el historial de comandos de shell. La **Tabla 2-1** resume algunos de los más utilizados atajos Consulte la **Figura 2-5** para ver una ilustración de algunos de estos, como el cursor movimienot.

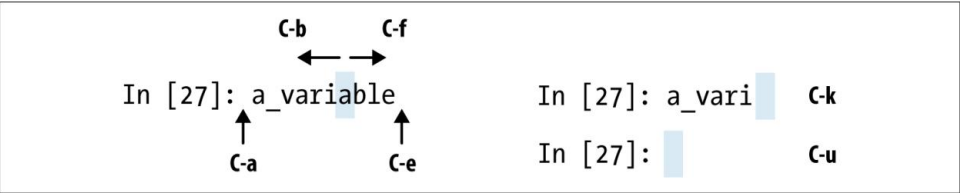


Figura 2-5. Ilustración de algunos métodos abreviados de teclado en el shell de IPython

Tabla 2-1. Métodos abreviados de teclado estándar de IPython

Atajo de teclado	Descripción
Ctrl-P o flecha arriba	Buscar hacia atrás en el historial de comandos los comandos que comiencen con el texto ingresado actualmente
Ctrl-N o flecha hacia abajo	Buscar hacia adelante en el historial de comandos los comandos que comienzan con el texto ingresado actualmente
Ctrl-R	Búsqueda de historial inverso estilo Readline (coincidencia parcial)
Ctrl-Mayús-V	Pegar texto desde el portapapeles
Ctrl-C	Interrumpir el código que se está ejecutando actualmente
Ctrl-A	Mover el cursor al principio de la línea
Ctrl-E	Mover el cursor al final de la línea
Ctrl-K	Eliminar texto del cursor hasta el final de la línea
Ctrl-U	Descartar todo el texto en la línea actual
Ctrl-F	Mover el cursor hacia adelante un carácter
Ctrl-B	Mover el cursor hacia atrás un carácter
Ctrl-L	Pantalla clara

Tenga en cuenta que los portátiles Jupyter tienen un conjunto de métodos abreviados de teclado en gran medida independiente para la navegación. igración y edición. Dado que estos accesos directos han evolucionado más rápidamente que los de IPython, lo animamos a explorar el sistema de ayuda integrado en los menús de Jupyter Notebook.

Acerca de los comandos mágicos

Los comandos especiales de IPython (que no están integrados en Python) se conocen como comandos "mágicos". Estos están diseñados para facilitar tareas comunes y permitirle controlar fácilmente el comportamiento del sistema IPython. Un comando mágico es cualquier comando precedido por el símbolo de porcentaje %. Por ejemplo, puede verificar el tiempo de ejecución de cualquier declaración de Python, como una multiplicación de matrices, utilizando la función mágica `%timeit` (que se analizará con más detalle más adelante):

```
En [20]: a = np.random.randn(100, 100)
```

```
En [20]: %timeit np.dot(a, a) 10000
bucles, lo mejor de 3: 20,9 µs por bucle
```

Los comandos mágicos se pueden ver como programas de línea de comandos que se ejecutan dentro del sistema IPython. Muchos de ellos tienen opciones adicionales de "línea de comandos", que se pueden ver (como era de esperar) usando ?:

```
En [21]: %debug?
```

```
cadena de documentación:
::
```

```
%debug [--breakpoint ARCHIVO:LÍNEA] [declaración [declaración ...]]
```

Active el depurador interactivo.

Este comando mágico admite dos formas de activar el depurador.

Una es activar el depurador antes de ejecutar el código. De esta manera, puede establecer un punto de interrupción para recorrer el código desde el punto.

Puede usar este modo dando declaraciones para ejecutar y, opcionalmente, un punto de interrupción.

El otro es activar el depurador en modo post-mortem. Puede activar este modo simplemente ejecutando `%debug` sin ningún argumento.

Si acaba de ocurrir una excepción, esto le permite inspeccionar sus marcos de pila de forma interactiva. Tenga en cuenta que esto siempre funcionará solo en el último rastreo que ocurrió, por lo que debe llamarlo rápidamente después de que se haya activado una excepción que desea inspeccionar, porque si ocurre otra, golpea a la anterior.

Si desea que IPython haga esto automáticamente en cada excepción, consulte la magia de `%pdb` para obtener más detalles.

argumentos posicionales:

declaración

Código para ejecutar en el depurador. Puede omitir esto en el modo de magia celular.

argumentos opcionales:

--breakpoint <ARCHIVO:LÍNEA>, -b <ARCHIVO:LÍNEA> Establece

el punto de interrupción en la LÍNEA en el ARCHIVO.

Las funciones mágicas se pueden usar de forma predeterminada sin el signo de porcentaje, siempre que no haya variaciones. Se define con el mismo nombre que la función mágica en cuestión. Esta característica es llamada automático y se puede habilitar o deshabilitar con %automagic.

Algunas funciones mágicas se comportan como funciones de Python y su salida se puede asignar a una variable:

```
En [22]: %contraseña
Salida[22]: '/home/wesm/code/pydata-book'
```

```
En [23]: foo = %pwd
```

```
En [24]: Foo
Salida[24]: '/home/wesm/code/pydata-book'
```

Dado que se puede acceder a la documentación de IPython desde el sistema, le animo para explorar todos los comandos especiales disponibles escribiendo %quickref o %magic. La Tabla 2-2 destaca algunos de los más críticos para ser productivo en interactivo informática y desarrollo de Python en IPython.

Tabla 2-2. Algunos comandos mágicos de IPython de uso frecuente

Dominio	Descripción
%refrápida	Mostrar la tarjeta de referencia rápida de IPython
%magia	Mostrar documentación detallada de todos los comandos mágicos disponibles
%depuración	Ingresa el depurador interactivo en la parte inferior del último rastreo de excepción
%hist	Imprima el historial de entrada de comandos (y salida opcional)
%pdb	Ingresa automáticamente al depurador después de cualquier excepción
%pegar	Ejecutar código Python preformateado desde el portapapeles
%cpegar	Abra un aviso especial para pegar manualmente el código de Python para ejecutarlo
%reiniciar	Eliminar todas las variables/nombres definidos en el espacio de nombres interactivo
%pagina OBJETO	Pretty-print el objeto y mostrarlo a través de un buscapersonas
%ejecutar secuencia de comandos.py	Ejecute un script de Python dentro de IPython
declaración %prun	Ejecute declaración con cProfile e informe la salida del generador de perfiles
% declaración de tiempo	Informe el tiempo de ejecución de una sola instrucción
declaración %timeit	Ejecute una declaración varias veces para calcular un tiempo de ejecución promedio conjunto; útil para código de tiempo con tiempo de ejecución muy corto
%who, %who_ls, %whos	Mostrar variables definidas en un espacio de nombres interactivo, con distintos niveles de información/verbosidad
variable %xdel	Elimine una variable e intente borrar cualquier referencia al objeto en las partes internas de IPython

Integración con matplotlib

Una de las razones de la popularidad de IPython en la computación analítica es que se integra bien con visualización de datos y otras bibliotecas de interfaz de usuario como matplotlib. No te preocupes si nunca ha usado matplotlib antes; se discutirá con más detalle más adelante en

este libro. La función mágica `%matplotlib` configura su integración con el shell IPython o el cuaderno Jupyter. Esto es importante, ya que, de lo contrario, los gráficos que cree no aparecerán (notebook) o tomarán el control de la sesión hasta que se cierren (shell).

En el shell de IPython, ejecutar `%matplotlib` configura la integración para que pueda crear varias ventanas de gráficos sin interferir con la sesión de la consola:

En [26]: `%matplotlib`

Usando el backend de matplotlib: `Qt4Agg`

En Jupyter, el comando es un poco diferente (Figura 2-6):

En [26]: `%matplotlib en línea`

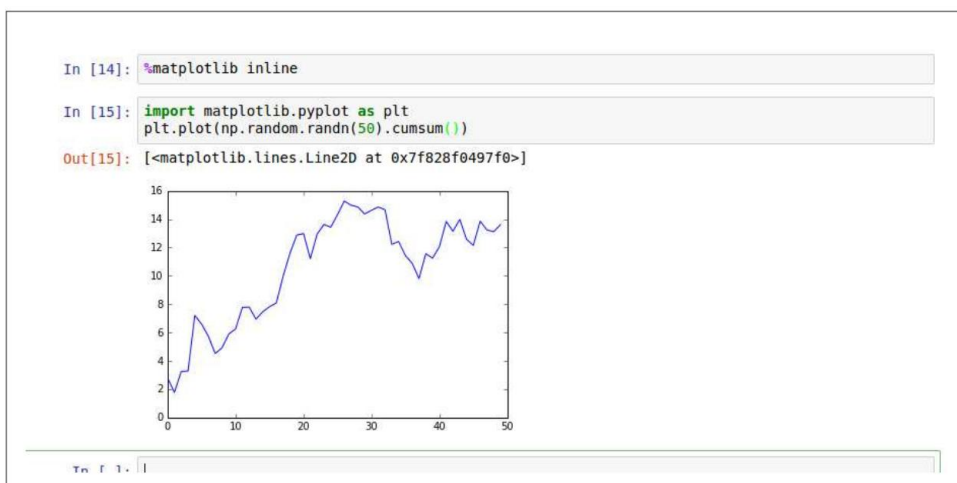


Figura 2-6. Trazado de matplotlib en línea de Jupyter

2.3 Conceptos básicos del lenguaje Python

En esta sección, le daré una descripción general de los conceptos esenciales de programación de Python y la mecánica del lenguaje. En el próximo capítulo, entraré en más detalles sobre las estructuras de datos, funciones y otras herramientas integradas de Python.

Semántica del lenguaje El

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

Sangría, no llaves

Python usa espacios en blanco (tabulaciones o espacios) para estructurar el código en lugar de usar llaves como en muchos otros lenguajes como R, C++, Java y Perl. Considere un bucle for de un algoritmo de clasificación:

```

para x en matriz:
    si x < pivote:
        menos.agregar(x)
    si no:
        mayor.agregar(x)

```

Los dos puntos indican el comienzo de un bloque de código con sangría, después de lo cual todo el código debe tener la misma sangría hasta el final del bloque.

Lo ames o lo odies, los espacios en blanco significativos son una realidad para los programadores de Python y, según mi experiencia, pueden hacer que el código de Python sea más legible que otros lenguajes que he usado. Si bien puede parecer extraño al principio, es de esperar que se acostumbre con el tiempo.



Recomiendo encarecidamente usar cuatro espacios como sangría predeterminada y reemplazar las pestañas con cuatro espacios. Muchos editores de texto tienen una configuración que reemplazará las tabulaciones con espacios automáticamente (¡haz esto!). Algunas personas usan tabulaciones o un número diferente de espacios, y dos espacios no son terriblemente raros. En general, cuatro espacios es el estándar adoptado por la gran mayoría de los programadores de Python, por lo que recomiendo hacerlo en ausencia de una razón convincente de lo contrario.

Como puede ver ahora, las declaraciones de Python tampoco necesitan terminar con punto y coma. Sin embargo, se pueden usar punto y coma para separar varias declaraciones en una sola línea:

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

Por lo general, se desaconseja poner varias declaraciones en una línea en Python, ya que a menudo hace 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, estructura de datos, función, clase, módulo, etc. existe en el intérprete de Python en su propia "caja", que se denomina objeto de Python.

Cada objeto tiene un tipo asociado (por ejemplo, cadena o función) y datos internos. En la práctica, esto hace que el lenguaje sea muy flexible, ya que incluso las funciones pueden tratarse como cualquier otro objeto.

Comentarios

El intérprete de Python ignora cualquier texto precedido por la almohadilla (signo de almohadilla) `#`. Esto se usa a menudo para agregar comentarios al código. A veces también puede querer excluir ciertos bloques de código sin eliminarlos. Una solución fácil es comentar el código:

```

resultados =
[] for línea en file_handle: #
    mantener las líneas vacías por ahora
    # if len(línea) == 0: #
    continuar
    resultados.append(línea.reemplazar('foo', 'bar'))

```

Los comentarios también pueden aparecer después de una línea de código ejecutado. Si bien algunos programadores prefieren que los comentarios se coloquen en la línea que precede a una línea de código en particular, esto puede ser útil a veces:

```
print("Llegué a esta línea") # Informe de estado simple
```

Llamadas a métodos de funciones y objetos

Llamas a funciones usando paréntesis y pasando cero o más argumentos, opcionalmente asignando el valor devuelto a una variable:

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

Casi todos los objetos en Python tienen funciones adjuntas, conocidas como métodos, que tienen acceso al contenido interno del objeto. Puede llamarlos usando la siguiente sintaxis:

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

Las funciones pueden tomar tanto argumentos posicionales como de palabras clave:

```
resultado = f(a, b, c, d=5, e='foo')
```

Más sobre esto más adelante.

Variables y paso de argumentos Al

asignar una variable (o nombre) en Python, está creando una referencia al objeto en el lado derecho del signo igual. En términos prácticos, considere una lista de números enteros:

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

Supongamos que asignamos a a una nueva variable b:

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

En algunos idiomas, esta asignación haría que se copiaran los datos [1, 2, 3]. En Python, a y b en realidad ahora se refieren al mismo objeto, la lista original [1, 2, 3] (consulte [la Figura 2-7](#) para ver una maqueta). Puedes comprobarlo por ti mismo agregando un elemento a a y luego examinando b:

```
En [10]: a.append(4)
```

```
En [11]: b
Salida[11]: [1, 2, 3, 4]
```

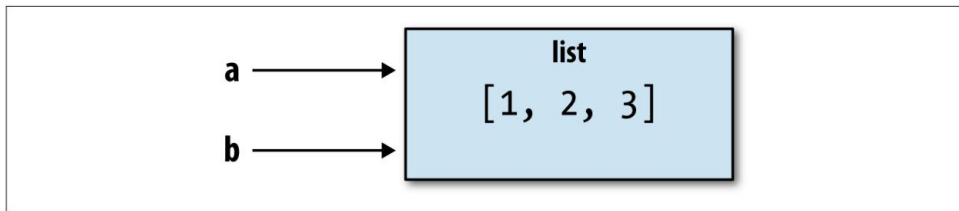


Figura 2-7. Dos referencias para el mismo objeto.

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



La asignación también se conoce como vinculación, ya que vinculamos un nombre a un objeto. Los nombres de variables que se han asignado pueden ocasionalmente denominarse variables vinculadas.

Cuando pasa objetos como argumentos a una función, se crean nuevas variables locales que hacen referencia a los objetos originales sin ninguna copia. Si vincula un nuevo objeto a una variable dentro de una función, ese cambio no se reflejará en el ámbito principal. Por lo tanto, es posible alterar las partes internas de un argumento mutable. Supongamos que tuviéramos la siguiente función:

```
def append_element(alguna_lista, elemento):
    alguna_lista.append(elemento)
```

Entonces nosotros tenemos:

```
En [27]: datos = [1, 2, 3]
```

```
En [28]: append_element(datos, 4)
```

```
En [29]: datos
```

```
Salida[29]: [1, 2, 3, 4]
```

Referencias dinámicas, tipos fuertes

A diferencia de muchos lenguajes compilados, como Java y C++, las referencias a objetos en Python no tienen ningún tipo asociado. No hay problema con lo siguiente:

```
En [12]: a = 5
```

```
En [13]: tipo(a)
```

```
Salida[13]: int
```

```
En [14]: a = 'foo'
```

```
En [15]: tipo(a)
Salida[15]: calle
```

Las variables son nombres de objetos dentro de un espacio de nombres particular; la información de tipo se almacena en el propio objeto. Algunos observadores podrían concluir apresuradamente que Python no es un "lenguaje escrito". Esto no es verdad; considera este ejemplo:

```
En [16]: '5' + 5

TypeError                                Rastreo (última llamada más reciente)
<ipython-input-16-f9dbf5f0b234> en <módulo>() ----> 1 '5'
+ 5
TypeError: debe ser str, no int
```

En algunos lenguajes, como Visual Basic, la cadena '5' puede convertirse (o convertir) implícitamente en un número entero, lo que da como resultado 10. Sin embargo, en otros lenguajes, como JavaScript, el número entero 5 puede convertirse en una cadena, lo que da como resultado la cadena concatenada '55'. En este sentido, Python se considera un lenguaje fuertemente tipado, lo que significa que cada objeto tiene un tipo (o clase) específico, y las conversiones implícitas ocurrirán solo en ciertas circunstancias obvias, como las siguientes:

```
En [17]: a = 4,5

En [18]: b = 2

# Formateo de cadenas, que se visitará más
adelante En [19]: print('a es {0}, b es {1}'.format(type(a), type(b))) a es <class
'float'> , b es <class 'int'>

Entrada [20]: a /
b Salida [20]: 2,25
```

Conocer el tipo de un objeto es importante, y es útil poder escribir funciones que puedan manejar muchos tipos diferentes de entrada. Puede verificar que un objeto es una instancia de un tipo particular usando la función `isinstance`:

```
En [21]: a = 5

En [22]: es instancia (a, int)
Salida[22]: Verdadero
```

`isinstance` puede aceptar una tupla de tipos si desea verificar que el tipo de un objeto se encuentra entre los presentes en la tupla:

```
En [23]: a = 5; b = 4,5

En [24]: isinstance(a, (int, float))
Salida[24]: Verdadero

En [25]: isinstance(b, (int, float))
Salida[25]: Verdadero
```

Atributos y métodos

Los objetos en Python suelen tener atributos (otros objetos de Python almacenados "dentro" del objeto) y métodos (funciones asociadas con un objeto que pueden tener acceso a los datos internos del objeto). Se accede a ambos a través de la sintaxis `obj.attribute_name`:

```
En [1]: a = 'foo'
```

```
En [2]: a.<Presione
```

```
Tabulador> a.en mayúsculas
```

```
a.formato a.centrar a.índice
```

```
a.recuento a.isalnum
```

```
a.decodificar a.isalpha
```

```
a.codificar a.isdigit a.termina
```

```
con a.islower
```

```
a.expandtabs a.isspace a.find
```

```
a.isitle
```

```
a.isupper a.strip a.join a.index a.swapcase a.ljust
```

```
a.rpartition a.title a.lower a.rsplit a.lstrip a.rstrip a.partition
```

```
a.split a.replace a.splitlines a.rfind a.comienza con
```

```
a.traducir
```

```
a.superior
```

```
a.zfill
```

También se puede acceder a los atributos y métodos por nombre a través de la función `getattr` :

```
En [27]: getattr(a, 'dividir')
```

```
Salida[27]: <función str.split>
```

En otros idiomas, el acceso a los objetos por su nombre a menudo se denomina "reflexión".

Si bien no utilizaremos de manera extensa las funciones `getattr` y las funciones relacionadas `hasattr` y `setattr` en este libro, se pueden usar de manera muy efectiva para escribir código genérico y reutilizable.

Tipificación

de pato A menudo, es posible que no le importe el tipo de un objeto, sino solo si tiene ciertos métodos o comportamiento. Esto a veces se denomina "tipado de pato", después del dicho "Si camina como un pato y grazna como un pato, entonces es un pato". Por ejemplo, puede verificar que un objeto es iterable si implementó el protocolo iterador. Para muchos objetos, esto significa que tiene un "método mágico" `__iter__`, aunque una forma alternativa y mejor de verificar es intentar usar la función `iter` :

```
def isiterable(obj): prueba:
```

```
    iter(obj) return True
```

```
excepto TypeError: # no iterable return False
```

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

```
En [29]: isiterable('una cadena')
```

```
Salida[29]: Verdadero
```

```
En [30]: isiterable([1, 2, 3])
```

Salida[30]: Verdadero

En [31]: iterable(5)

Salida[31]: Falso

Un lugar donde uso esta funcionalidad todo el tiempo es para escribir funciones que pueden aceptar múltiples tipos de entrada. Un caso común es escribir una función que pueda aceptar cualquier tipo de secuencia (lista, tupla, ndarray) o incluso un iterador. Primero puede verificar si el objeto es una lista (o una matriz NumPy) y, si no lo es, convertirlo en uno:

si no es instancia (x, lista) e iterable (x): x = lista (x)

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

```
# algún_módulo.py
```

```
PI = 3.14159
```

```
definición f(x):
```

```
    volver x + 2
```

```
definición g(a, b):
```

```
    devolver a + b
```

Si quisiéramos acceder a las variables y funciones definidas en some_module.py, desde otro archivo en el mismo directorio podríamos hacer:

```
importar algún_módulo
resultado = algún_módulo.f(5) pi
= algún_módulo.PI
```

O equivalente:

```
from some_module importar f, g, PI
resultado = g(5, PI)
```

Al usar la palabra clave as , puede dar a las importaciones diferentes nombres de variables:

```
importar algún_módulo como
sm desde algún_módulo importar PI como pi, g como gf
```

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

Operadores binarios y comparaciones

La mayoría de las operaciones y comparaciones matemáticas binarias son como cabría esperar:

En [32]: 5 - 7

Salida[32]: -2

```
Entrada [33]: 12 + 21,5
Salida[33]: 33,5
```

```
En [34]: 5 <= 2
Salida[34]: Falso
```

Consulte la Tabla 2-3 para conocer todos los operadores binarios disponibles.

Para verificar si dos referencias se refieren al mismo objeto, use la palabra clave `is`. `is not` también es perfectamente válido si quieres comprobar que dos objetos no son iguales:

```
En [35]: a = [1, 2, 3]
```

```
En [36]: b = a
```

```
En [37]: c = lista(a)
```

```
En [38]: a es b
Salida[38]: Verdadero
```

```
En [39]: a no es c
Salida[39]: Verdadero
```

Dado que `lista` siempre crea una nueva lista de Python (es decir, una copia), podemos estar seguros de que `c` es distinta de `a`. Comparar con `is not` es lo mismo que con el operador `==`, porque en este caso tenemos:

```
En [40]: a == c
Salida[40]: Verdadero
```

Un uso muy común de `es` y `no es` para verificar si una variable es `Ninguna`, ya que solo hay una instancia de `Ninguna`:

```
En [41]: a = Ninguno
```

```
En [42]: a es Ninguno
Fuera[42]: Verdadero
```

Tabla 2-3. Operadores binarios

Operación	Descripción
<code>a + b</code>	Añadir a y b
<code>un-b</code>	Restar b de a
<code>a * b</code>	Multiplica a por b
<code>un / b</code>	dividir a por b
<code>un // segundo</code>	Floor-divide a por b, descartando cualquier resto fraccionario
<code>a ** b</code>	Elevar a a la potencia b
<code>a & b</code>	True si tanto a como b son True; para números enteros, tome AND bit a bit
<code>un b</code>	Verdadero si a o b es Verdadero; para números enteros, tome el OR bit a bit
<code>a ^ segundo</code>	Para valores booleanos, True si aob es True, pero no ambos; para números enteros, tome el EXCLUSIVO-OR bit a bit

Operación	Descripción
<code>un == segundo</code>	Verdadero si a es igual a b
<code>un != segundo</code>	Verdadero si a no es igual a b
<code>a <= b</code> , <code>a < b</code>	Verdadero si a es menor que (menor o igual) que b
<code>ba > b</code> , <code>a >= b</code>	Verdadero si a es mayor que (mayor que o igual) que b
<code>a es b</code>	Verdadero si a y b hacen referencia al mismo objeto de Python
<code>a no es b</code>	Verdadero si a y b hacen referencia a diferentes objetos de Python

Objetos mutables e inmutables La

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

```
En [43]: una_lista = ['foo', 2, [4, 5]]

En [44]: una_lista[2] = (3, 4)

En [45]: una_lista
Fuera[45]: ['foo', 2, (3, 4)]
```

Otros, como cadenas y tuplas, son inmutables:

```
En [46]: una_tupla = (3, 5, (4, 5))

En [47]: a_tuple[1] = 'cuatro'
```

```
TypeError                                Rastreo (última llamada más reciente)
<ipython-input-47-b7966a9ae0f1> en <módulo>() ----> 1
a_tuple[1] = 'cuatro'
TypeError: el objeto 'tupla' no admite la asignación de elementos
```

Recuerde que el hecho de que pueda mutar un objeto no significa que siempre deba hacerlo. Tales acciones se conocen como efectos secundarios. Por ejemplo, al escribir una función, cualquier efecto secundario debe comunicarse explícitamente al usuario en la documentación o los comentarios de la función. Si es posible, recomiendo tratar de evitar los efectos secundarios y favorecer la inmutabilidad, aunque puede haber objetos mutables involucrados.

Tipos escalares

Python, junto con su biblioteca estándar, tiene un pequeño conjunto de tipos incorporados para manejar datos numéricos, cadenas, valores booleanos (verdadero o falso) y fechas y horas. Estos tipos de "valor único" a veces se denominan tipos escalares y en este libro nos referiremos a ellos como escalares. Consulte la [Tabla 2-4](#) para obtener una lista de los principales tipos de escalares. El manejo de la fecha y la hora se discutirá por separado, ya que estos son proporcionados por el módulo de fecha y hora en la biblioteca estándar.

Tabla 2-4. Tipos escalares estándar de Python

Tipo	Descripción
Ninguno	El valor "nulo" de Python (solo existe una instancia del objeto Ninguno)
calle	tipo de cadena; contiene cadenas Unicode (codificación UTF-8)
bytes	Bytes ASCII sin procesar (o Unicode codificados como bytes)
float	Número de punto flotante de precisión doble (64 bits) (tenga en cuenta que no hay un tipo doble separado)
bool	Un valor verdadero o falso
En t	Entero con signo de precisión arbitraria

Tipos numéricos

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

```
En [48]: ival = 17239871

En [49]: val ** 6
Salida[49]: 26254519291092456596965462913230729701102721
```

Los números de punto flotante se representan con el tipo flotante de Python . Debajo del capó, cada uno es un valor de doble precisión (64 bits). También se pueden expresar con notación científica:

```
En [50]: valorf = 7.243

En [51]: fval2 = 6.78e-5
```

La división de enteros que no da como resultado un número entero siempre producirá un número de coma flotante:

```
Entrada [52]: 3 / 2
Salida [52]: 1,5
```

Para obtener una división de enteros al estilo C (que descarta la parte fraccionaria si el resultado no es un número entero), use el operador de división de piso //:

```
En [53]: 3 // 2
Salida[53]: 1
```

Cadenas Mucha gente usa Python por sus poderosas y flexibles capacidades integradas de procesamiento de cadenas. Puede escribir literales de cadena usando comillas simples o dobles "":

```
a = 'una forma de escribir una cadena' b =
"otra forma"
```

Para cadenas de varias líneas con saltos de línea, puede usar comillas triples, ya sea """ o """:

```
c = """
Esta es una cadena más larga que
abarca varias líneas.
```

Puede que le sorprenda que esta cadena `c` en realidad contiene cuatro líneas de texto; la línea se rompe después de `"""` y las líneas posteriores se incluyen en la cadena. Podemos contar los caracteres de la nueva línea con el método de conteo en `c`:

```
En [55]: c.cuenta("\n")
Salida[55]: 3
```

Las cadenas de Python son inmutables; no se puede modificar una cadena:

```
En [56]: a = 'esto es una cadena'

En [57]: a[10] = 'f'
TypeError                                 Rastreo (última llamada más reciente)
<ipython-input-57-5ca625d1e504> en <módulo>() ----> 1
a[10] = 'f'
TypeError: el objeto 'str' no admite la asignación de elementos

En [58]: b = a.replace('cadena', 'cadena más larga')

En [59]: b
Out[59]: 'esta es una cadena más larga'
```

Después de esta operación, la variable `a` no se modifica:

```
En [60]: un
Out[60]: 'esto es una cadena'
```

Muchos objetos de Python se pueden convertir en una cadena usando la función `str`:

```
En [61]: a = 5.6

En [62]: s = cadena(a)

En [63]: impresión(es)
5.6
```

Las cadenas son una secuencia de caracteres Unicode y, por lo tanto, pueden tratarse como otras secuencias, como listas y tuplas (que exploraremos con más detalle en el próximo capítulo):

```
En [64]: s = 'pitón'

En [65]: lista(s)
Salida[65]: ['p', 'y', 't', 'h', 'o', 'n']

En [66]: s[:3]
Salida[66]: 'pyt'
```

La sintaxis `s[:3]` se llama rebanar y se implementa para muchos tipos de secuencias de Python. Esto se explicará con más detalle más adelante, ya que se usa ampliamente en este libro.

El carácter de barra invertida `\` es un carácter de escape, lo que significa que se utiliza para especificar caracteres especiales como nueva línea `\n` o caracteres Unicode. Para escribir un literal de cadena con barras diagonales inversas, debe escapar de ellas:

```
En [67]: s = '12\\34'
```

```
En [68]: imprimir(s)
12\\34
```

Si tiene una cadena con muchas barras invertidas y sin caracteres especiales, puede que le resulte un poco molesto. Afortunadamente, puede anteponer la comilla inicial de la cadena con `r`, lo que significa que los caracteres deben interpretarse como son:

```
En [69]: s = r'esto\\no\\tiene\\caracteres\\especiales'
```

```
En [70]: s
Out[70]: 'esto\\no\\tiene\\personajes\\especiales'
```

La `r` significa crudo.

Agregar dos cadenas juntas las concatena y produce una nueva cadena:

```
En [71]: a = 'esta es la primera mitad'
```

```
En [72]: b = 'y esta es la segunda mitad'
```

```
En [73]: a + b
Out[73]: 'esta es la primera mitad y esta es la segunda mitad'
```

La creación de plantillas o el formato de cadenas es otro tema importante. La cantidad de formas de hacerlo se ha ampliado con la llegada de Python 3, y aquí describiré brevemente la mecánica de una de las interfaces principales. Los objetos de cadena tienen un método de formato que se puede usar para sustituir argumentos formateados en la cadena, produciendo una nueva cadena:

```
En [74]: template = '{0:.2f} {1:s} valen US${2:d}'
```

En esta cadena,

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

Para sustituir argumentos por estos parámetros de formato, pasamos una secuencia de argumentos al método de formato :

```
En [75]: template.format(4.5560, 'Pesos argentinos', 1)
Fuera[75]: '4,56 pesos argentinos valen US$1'
```

El formato de cadenas es un tema profundo; hay múltiples métodos y numerosas opciones y ajustes disponibles para controlar cómo se formatean los valores en la cadena resultante. Para obtener más información, recomiendo consultar la [documentación oficial de Python](#).

Discuto el procesamiento general de cadenas en relación con el análisis de datos con más detalle en el [Capítulo 8](#).

Bytes y Unicode En

Python moderno (es decir, Python 3.0 y superior), Unicode se ha convertido en el tipo de cadena de primera clase para permitir un manejo más consistente de texto ASCII y no ASCII. En versiones anteriores de Python, las cadenas eran solo bytes sin ninguna codificación Unicode explícita. Puede convertir a Unicode suponiendo que conozca la codificación de caracteres. Veamos un ejemplo:

```
En [76]: val = "español"

En [77]: valor
Fuera[77]: 'español'
```

Podemos convertir esta cadena Unicode a su representación de bytes UTF-8 utilizando el método de codificación :

```
En [78]: val_utf8 = val.encode('utf-8')

En [79]: val_utf8
Fuera[79]: b'espa\xc3\xbfol'

En [80]: tipo (val_utf8)
Salida[80]: bytes
```

Suponiendo que conoce la codificación Unicode de un objeto de bytes , puede volver atrás usando el método de decodificación :

```
En [81]: val_utf8.decode('utf-8')
Fuera[81]: 'español'
```

Si bien se prefiere usar UTF-8 para cualquier codificación, por razones históricas es posible que encuentre datos en varias codificaciones diferentes:

```
En [82]: val.encode('latin1')
Fuera[82]: b'espa\xffol'

En [83]: val.encode('utf-16')
Salida[83]: b'\xff\xfe\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

En [84]: val.encode('utf-16le')
Salida[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

Es más común encontrar objetos de bytes en el contexto de trabajar con archivos, donde es posible que no se desee decodificar implícitamente todos los datos en cadenas Unicode.

Aunque es posible que rara vez necesite hacerlo, puede definir sus propios literales de bytes prefijando una cadena con b:

```
En [85]: bytes_val = b'esto es bytes'
```

```
En [86]: bytes_val
```

```
Salida[86]: b'esto son bytes'
```

```
En [87]: decodificado = bytes_val.decode('utf8')
```

```
En [88]: decodificado # esto es str (Unicode) ahora
```

```
Out[88]: 'esto es bytes'
```

Booleanos

Los dos valores booleanos en Python se escriben como Verdadero y Falso. Las comparaciones y otras expresiones condicionales se evalúan como Verdadero o Falso. Los valores booleanos se combinan con las palabras clave and y or :

```
En [89]: Verdadero y Verdadero
```

```
Salida[89]: Verdadero
```

```
En [90]: Falso o Verdadero
```

```
Salida[90]: Verdadero
```

Tipo de fundición

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

```
En [91]: s = '3.14159'
```

```
En [92]: fval = flotante(s)
```

```
En [93]: tipo (fval)
```

```
Salida[93]: flotante
```

```
En [94]: int(fval)
```

```
Salida[94]: 3
```

```
En [95]: booleano(fval)
```

```
Fuera[95]: Verdadero
```

```
En [96]: bool(0)
```

```
Salida[96]: Falso
```

Ninguno

Ninguno es el tipo de valor nulo de Python. Si una función no devuelve explícitamente un valor, implícitamente devuelve Ninguno:

```
En [97]: a = Ninguno
```

```
En [98]: a es Ninguno
```

```
Salida[98]: Verdadero
```

```
En [99]: b = 5
```

```
En [100]: b no es Ninguno
```

```
Salida[100]: Verdadero
```

Ninguno también es un valor predeterminado común para los argumentos de función:

```
def sumar_y_tal vez_multiplicar(a, b, c=Ninguno): resultado =  
    a + b
```

```
    si c no es Ninguno:
```

```
        resultado = resultado * c
```

```
    resultado devuelto
```

Si bien es un punto técnico, vale la pena tener en cuenta que None no es solo una palabra clave reservada, sino también una instancia única de NoneType:

```
En [101]: tipo (Ninguno)
```

```
Salida[101]: NingunoTipo
```

Fechas y horas

El módulo integrado de fecha y hora de Python proporciona tipos de fecha y hora, fecha y hora. El tipo datetime, como puedes imaginar, combina la información almacenada en date y time y es el más utilizado:

```
In [102]: from datetime import datetime, date, time
```

```
En [103]: dt = fecha y hora (2011, 10, 29, 20, 30, 21)
```

```
Entrada [104]: dt.día
```

```
Salida[104]: 29
```

```
Entrada [105]: dt.minuto
```

```
Salida[105]: 30
```

Dada una instancia de fecha y hora, puede extraer los objetos de fecha y hora equivalentes llamando a métodos en la fecha y hora del mismo nombre:

```
En [106]: dt.fecha()
```

```
Salida[106]: fechahora.fecha(2011, 10, 29)
```

```
En [107]: dt.time()
Salida[107]: fecha hora.hora(20, 30, 21)
```

El método strftime formatea una fecha y hora como una cadena:

```
En [108]: dt.strftime('%m/%d/%Y %H:%M')
Salida[108]: '29/10/2011 20:30'
```

Las cadenas se pueden convertir (analizar) en objetos de fecha y hora con la función.strptime :

```
En [109]: fecha hora.strptime('20091031', '%Y%m%d')
Salida[109]: fecha hora.fecha hora(2009, 10, 31, 0, 0)
```

Consulte la Tabla 2-5 para obtener una lista completa de las especificaciones de formato.

Cuando esté agregando o agrupando datos de series temporales, ocasionalmente será útil reemplazar los campos de tiempo de una serie de fechas y horas, por ejemplo, reemplazar los campos de minutos y segundos con cero:

```
En [110]: dt.replace(minuto=0, segundo=0)
Salida[110]: fecha hora.fecha hora(2011, 10, 29, 20, 0)
```

Dado que datetime.datetime es un tipo inmutable, métodos como estos siempre producen nuevos objetos.

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

```
En [111]: dt2 = fecha y hora (2011, 11, 15, 22, 30)

En [112]: delta = dt2 - dt

Entrada [113]:
delta Salida[113]: fecha hora.timedelta(17, 7179)

En [114]: tipo (delta)
Salida[114]: fecha hora.timedelta
```

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

Agregar un timedelta a una fecha y hora produce una nueva fecha y hora cambiada :

```
Entrada [115]:
dt Salida[115]: fecha hora.fecha hora(2011, 10, 29, 20, 30, 21)

Entrada [116]: dt + delta
Salida [116]: datetime.datetime(2011, 11, 15, 22, 30)
```

Tabla 2-5. Especificación de formato de fecha y hora (compatible con ISO C89)

Tipo	Descripción
%Y	Año de cuatro
%y	dígitos Año de dos dígitos

Tipo	Descripción	%m
	Mes de dos dígitos [01, 12]	%d Día de dos dígitos [01, 31]
	%H Hora (reloj de 24 horas) [00, 23]	%I Hora (reloj de 12 horas) [01, 12]
	%M Minuto de dos dígitos [00, 59]	%S Segundo [00, 61] (los segundos 60, 61 representan los segundos bisiestos) %w
	Día de la semana como número entero [0 (domingo), 6]	%U Número de semana del año [00, 53]; El domingo se considera el primer día de la semana, y los días anteriores al primer domingo del año son "semana 0"
	%W Número de semana del año [00, 53]; Se considera lunes el primer día de la semana, y los días anteriores al primer lunes del año es "semana 0" %z	
	desplazamiento de zona horaria UTC como +HHMM o -HHMM; vacío si la zona horaria es ingenua	
	%F Atajo para %Y-%m-%d (p. ej., 2012-4-18)	
	Acceso directo a %D para %m/%d/%y (p. ej., 18/04/12)	

Flujo de control

Python tiene varias palabras clave integradas para lógica condicional, bucles y otros conceptos de flujo de control estándar que se encuentran en otros lenguajes de programación.

si, elif, y más

La sentencia if es uno de los tipos de sentencias de flujo de control más conocidas. Comprueba una condición que, si es Verdadera, evalúa el código en el bloque que sigue:

```
si x < 0:
    print('Es negativo')
```

Una declaración if puede ser seguida opcionalmente por uno o más bloques elif y un bloque catch all else si todas las condiciones son falsas:

```
si x < 0:
    print('Es negativo')
elif x == 0:
    print('Igual a cero')
elif 0 < x < 5:
    print('Positivo pero menor que 5')
else:
    print('Positivo y mayor o igual a 5')
```

Si alguna de las condiciones es verdadera, no se alcanzarán más bloques elif o else . Con una condición compuesta usando y o o, las condiciones se evalúan de izquierda a derecha y provocarán un cortocircuito:

```
En [117]: a = 5; b = 7
En [118]: c = 8; re = 4
```

```

En [119]: si a < b o c > d: print('Hecho ')
.....:
lo hizo

```

En este ejemplo, la comparación `c > d` nunca se evalúa porque la primera comparación era Verdadero.

También es posible encadenar comparaciones:

```

En [120]: 4 > 3 > 2 > 1
Salida[120]: Verdadero

```

for loops

for loops son para iterar sobre una colección (como una lista o tupla) o un iterador. La sintaxis estándar para un bucle for es:

```

por valor en la colección:
    # hacer algo con valor

```

Puede avanzar un bucle for a la siguiente iteración, omitiendo el resto del bloque, usando la palabra clave `continuar`.

Considere este código, que suma los números enteros en una lista y omite los valores Ninguno:

```

secuencia = [1, 2, Ninguno, 4, Ninguno, 5]
totales = 0
para el valor en secuencia:
    si el valor es Ninguno:
        continuar
    total += valor

```

Se puede salir de un bucle for por completo con la palabra clave `break`. Este código suma los elementos de la lista hasta llegar a un 5:

```

secuencia = [1, 2, 0, 4, 6, 5, 2, 1] total_hasta_5 = 0
para valor en secuencia:
    si valor == 5: romper
    total_hasta_5 +=
        valor

```

La palabra clave `break` solo termina el bucle for más interno; cualquier bucle for externo seguirá corriendo:

```

En [121]: 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 en la colección o el iterador son secuencias (tuplas o listas, por ejemplo), se pueden descomprimir convenientemente en variables en la declaración del bucle `for` :

```
para a, b, c en el iterador: # hacer
    algo
```

mientras que los bucles

Un bucle `while` especifica una condición y un bloque de código que se ejecutará hasta que la condición se evalúe como Falsa o el bucle finalice explícitamente con una interrupción:

```
x = 256
total = 0
mientras x > 0: si
    total > 500: romper

    total += x
    x = x // 2
```

aprobar

`pass` es la declaración "no-op" en Python. Se puede utilizar en bloques en los que no se va a realizar ninguna acción (o como marcador de posición para el código que aún no se ha implementado); solo se requiere porque Python usa espacios en blanco para delimitar bloques:

```
si x < 0:
    print("negativo!") elif x ==
0:
    # TODO: poner algo inteligente aquí
    pase
else:
    print("¡positivo!")
```

rango

La función de `rango` devuelve un iterador que produce una secuencia de enteros espaciados uniformemente:

```
En [122]: rango(10)
Salida[122]: rango(0, 10)
```

```
En [123]: lista(rango(10))
Salida[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

```
En [124]: lista(rango(0, 20, 2))
Salida[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
En [125]: lista(rango(5, 0, -1))
Salida[125]: [5, 4, 3, 2, 1]
```

Como puede ver, el rango produce números enteros hasta el punto final, pero sin incluirlo. Un uso común de rango es para iterar a través de secuencias por índice:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)): val = seq[i]
```

Si bien puede usar funciones como lista para almacenar todos los enteros generados por rango en alguna otra estructura de datos, a menudo la forma de iterador predeterminada será la que desee. Este fragmento suma todos los números del 0 al 99 999 que son múltiplos de 3 o 5:

```
suma = 0
for i in range(100000): # %
    es el operador de módulo si i %
    3 == 0 o i % 5 == 0:
        suma += i
```

Si bien el rango generado puede ser arbitrariamente grande, el uso de la memoria en un momento dado puede ser muy pequeño.

Expresiones ternarias

Una expresión ternaria en Python le permite combinar un bloque if-else que produce un valor en una sola línea o expresión. La sintaxis para esto en Python es:

```
valor = true-expr si la condición es false-expr
```

Aquí, true-expr y false-expr pueden ser cualquier expresión de Python. Tiene el mismo efecto que el más detallado:

```
si condición:
    valor = true-expr
demás:
    valor = false-expr
```

Este es un ejemplo más concreto:

```
En [126]: x = 5

En [127]: 'No negativo' si x >= 0 sino 'Negativo'
Salida[127]: 'No negativo'
```

Al igual que con los bloques if-else , solo se ejecutará una de las expresiones. Por lo tanto, los lados "if" y "else" de la expresión ternaria podrían contener cálculos costosos, pero solo se evalúa la rama verdadera.

Si bien puede ser tentador usar siempre expresiones ternarias para condensar su código, tenga en cuenta que puede sacrificar la legibilidad si la condición, así como las expresiones verdadera y falsa, son muy complejas.

Estructuras de datos, funciones y archivos incorporados

Este capítulo analiza las capacidades integradas en el lenguaje Python que se usarán de manera ubicua a lo largo del libro. Si bien las bibliotecas complementarias como pandas y NumPy agregan funcionalidad computacional avanzada para conjuntos de datos más grandes, están diseñadas para usarse junto con las herramientas de manipulación de datos integradas de Python.

Comenzaremos con las estructuras de datos del caballo de batalla de Python: tuplas, listas, dictados y conjuntos. Luego, discutiremos la creación de sus propias funciones de Python reutilizables. Finalmente, veremos la mecánica de los objetos de archivo de Python y la interacción con su disco duro local.

3.1 Estructuras de datos y secuencias

Las estructuras de datos de Python son simples pero poderosas. Dominar su uso es una parte fundamental para convertirse en un programador de Python competente.

Tupla

Una tupla es una secuencia inmutable y de longitud fija de objetos de Python. La forma más fácil de crear uno es con una secuencia de valores separados por comas:

```
En [1]: tup = 4, 5, 6
```

```
En [2]: sup
```

```
Salida[2]: (4, 5, 6)
```

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

```
En [3]: nested_tup = (4, 5, 6), (7, 8)
```

```
En [4]: nested_tup
```

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

Puede convertir cualquier secuencia o iterador en una tupla invocando la tupla:

```
En [5]: tupla([4, 0, 2])
```

```
Salida[5]: (4, 0, 2)
```

```
En [6]: tup = tupla('cadena')
```

```
En [7]: sup
```

```
Salida[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

Se puede acceder a los elementos con corchetes [] como con la mayoría de los otros tipos de secuencia.

Al igual que en C, C++, Java y muchos otros lenguajes, las secuencias están indexadas en 0 en Python:

```
En [8]: tup[0]
```

```
Salida[8]: 's'
```

Si bien los objetos almacenados en una tupla pueden ser mutables, una vez que se crea la tupla, no es posible modificar qué objeto se almacena en cada ranura:

```
En [9]: tup = tupla(['foo', [1, 2], True])
```

```
En [10]: tup[2] = Falso
```

```
TypeError                                Rastreo (última llamada más reciente)
<ipython-input-10-c7308343b841> en <módulo>() ----> 1
tup[2] = Falso
TypeError: el objeto 'tupla' no admite la asignación de elementos
```

Si un objeto dentro de una tupla es mutable, como una lista, puede modificarlo en el lugar:

```
En [11]: tup[1].append(3)
```

```
En [12]: tup
```

```
Salida[12]: ('foo', [1, 2, 3], Verdadero)
```

Puede concatenar tuplas usando el operador + para producir tuplas más largas:

```
En [13]: (4, Ninguno, 'foo') + (6, 0) + ('bar',)
```

```
Salida[13]: (4, Ninguno, 'foo', 6, 0, 'bar')
```

Multiplicar una tupla por un número entero, como con las listas, tiene el efecto de concatenar juntas tantas copias de la tupla:

```
En [14]: ('foo', 'barra') * 4
```

```
Salida[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Tenga en cuenta que los objetos en sí no se copian, solo las referencias a ellos.

Desempaquetando tuplas

Si intenta asignar una expresión de variables similar a una tupla, Python intentará desempaquetar el valor en el lado derecho del signo igual:

```
En [15]: tup = (4, 5, 6)
```

```
En [16]: a, b, c = tup
```

```
En [17]: b
Salida[17]: 5
```

Incluso las secuencias con tuplas anidadas se pueden desempaquetar:

```
En [18]: tup = 4, 5, (6, 7)
```

```
En [19]: a, b, (c, d) = tup
```

```
En [20]: re
Salida[20]: 7
```

Con esta función, puede cambiar fácilmente los nombres de las variables, una tarea que en muchos idiomas podría verse así:

```
tmp = aa
= b
b = tmp
```

Pero, en Python, el intercambio se puede hacer así:

```
En [21]: a, b = 1, 2
```

```
En [22]: un
Salida[22]: 1
```

```
En [23]: b
Salida[23]: 2
```

```
En [24]: b, a = a, b
```

```
En [25]: un
Salida[25]: 2
```

```
En [26]: b
Salida[26]: 1
```

Un uso común del desempaquetado de variables es iterar sobre secuencias de tuplas o listas:

```
En [27]: secuencia = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
En [28]: para a, b, c in seq: print('a={0},
c=6          b={1}, c={2}'.format(a, b, c)) .....: a=1, b=2, c=3 a=4, b=5,
a=7, b=8, c=9
```

Otro uso común es devolver múltiples valores de una función. Cubriré esto con más detalle más adelante.

El lenguaje Python adquirió recientemente un desempaquetado de tupla más avanzado para ayudar con situaciones en las que es posible que desee "arrancar" algunos elementos desde el principio de una tupla. Esto usa la sintaxis especial `*rest`, que también se usa en firmas de funciones para capturar una lista arbitrariamente larga de argumentos posicionales:

```
En [29]: valores = 1, 2, 3, 4, 5
```

```
En [30]: a, b, *resto = valores
```

```
En [31]: a, b
```

```
Salida[31]: (1, 2)
```

```
En [32]: descanso
```

```
Salida[32]: [3, 4, 5]
```

Este bit de descanso a veces es algo que desea descartar; no hay nada especial en el resto del nombre. Como cuestión de convención, muchos programadores de Python utilizarán el guión bajo (`_`) para las variables no deseadas:

```
En [33]: a, b, *_ = valores
```

Métodos de

tupla Dado que el tamaño y el contenido de una tupla no se pueden modificar, es muy ligero en los métodos de instancia. Uno particularmente útil (también disponible en listas) es `count`, que cuenta el número de ocurrencias de un valor:

```
En [34]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
En [35]: a.cuenta(2)
```

```
Salida[35]: 4
```

Lista

A diferencia de las tuplas, las listas son de longitud variable y su contenido se puede modificar en el lugar. Puede definirlos usando corchetes `[]` o usando la función de tipo de lista :

```
En [36]: a_list = [2, 3, 7, Ninguno]
```

```
En [37]: tup = ('foo', 'bar', 'baz')
```

```
En [38]: b_list = list(tup)
```

```
En [39]: b_lista
```

```
Salida[39]: ['foo', 'bar', 'baz']
```

```
En [40]: b_list[1] = 'cucú'
```

```
En [41]: lista_b
Fuera[41]: ['foo', 'peekaboo', 'baz']
```

Las listas y las tuplas son semánticamente similares (aunque las tuplas no se pueden modificar) y se pueden usar indistintamente en muchas funciones.

La función de lista se usa con frecuencia en el procesamiento de datos como una forma de materializar una expresión iteradora o generadora:

```
En [42]: gen = rango(10)

En [43]: gen
Salida[43]: rango(0, 10)

En [44]: lista(gen)
Salida[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adición y eliminación de elementos.

Los elementos se pueden agregar al final de la lista con el método de agregar :

```
En [45]: b_list.append('enano')

En [46]: b_lista
Fuera[46]: ['foo', 'peekaboo', 'baz', 'enano']
```

Usando insertar puede insertar un elemento en una ubicación específica en la lista:

```
En [47]: b_list.insert(1, 'rojo')

En [48]: b_lista
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

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



insert es computacionalmente costoso en comparación con append, porque las referencias a los elementos subsiguientes deben cambiarse internamente para dejar espacio para el nuevo elemento. Si necesita insertar elementos tanto al principio como al final de una secuencia, es posible que desee explorar `collections.deque`, una cola de dos extremos, para este propósito.

La operación inversa para insertar es pop, que elimina y devuelve un elemento en un índice particular:

```
En [49]: b_list.pop(2)
Fuera[49]: 'peekaboo'

En [50]: b_lista
Fuera[50]: ['foo', 'rojo', 'baz', 'enano']
```

Los elementos se pueden eliminar por valor con `remove`, que localiza el primer valor y lo elimina del último:

```
En [51]: b_list.append('foo')
```

```
En [52]: b_lista
```

```
Fuera[52]: ['foo', 'rojo', 'baz', 'enano', 'foo']
```

```
En [53]: b_list.remove('foo')
```

```
En [54]: b_lista
```

```
Fuera[54]: ['rojo', 'baz', 'enano', 'foo']
```

Si el rendimiento no es una preocupación, al usar agregar y eliminar, puede usar una lista de Python como una estructura de datos de "conjunto múltiple" perfectamente adecuada.

Compruebe si una lista contiene un valor utilizando la palabra clave `in` :

```
En [55]: 'enano' in b_list
```

```
Salida[55]: Verdadero
```

La palabra clave `not` puede usarse para negar en:

```
En [56]: 'enano' not in b_list
```

```
Salida[56]: Falso
```

Verificar si una lista contiene un valor es mucho más lento que hacerlo con dictados y conjuntos (que se presentarán en breve), ya que Python realiza un escaneo lineal a través de los valores de la lista, mientras que puede verificar los demás (basado en tablas hash) en tiempo constante.

Concatenar y combinar listas

Similar a las tuplas, agregar dos listas junto con `+` las concatena:

```
En [57]: [4, Ninguno, 'foo'] + [7, 8, (2, 3)]
```

```
Fuera[57]: [4, Ninguno, 'foo', 7, 8, (2, 3)]
```

Si ya tiene una lista definida, puede agregarle varios elementos utilizando el método de extensión :

```
En [58]: x = [4, Ninguno, 'foo']
```

```
En [59]: x.extend([7, 8, (2, 3)])
```

```
En [60]: x
```

```
Fuera[60]: [4, Ninguno, 'foo', 7, 8, (2, 3)]
```

Tenga en cuenta que la concatenación de listas por adición es una operación comparativamente costosa, ya que se debe crear una nueva lista y copiar los objetos. Usualmente es preferible usar `extend` para agregar elementos a una lista existente, especialmente si está creando una lista grande. De este modo,

```

todo = [] para
fragmento en list_of_lists:
    todo.extend(trozo)

```

es más rápido que la alternativa concatenativa:

```

todo = [] para
fragmento en list_of_lists: todo = todo
    + fragmento

```

Clasificación

Puede ordenar una lista en el lugar (sin crear un nuevo objeto) llamando a su función de ordenación :

```
En [61]: a = [7, 2, 5, 1, 3]
```

```
En [62]: a.sort()
```

```
En [63]: un
```

```
Salida[63]: [1, 2, 3, 5, 7]
```

sort tiene algunas opciones que ocasionalmente serán útiles. Una es la capacidad de pasar una clave de clasificación secundaria, es decir, una función que produce un valor para usar para clasificar los objetos. Por ejemplo, podríamos ordenar una colección de cadenas por sus longitudes:

```
En [64]: b = ['sierra', 'pequeño', 'Él', 'zorros', 'seis']
```

```
En [65]: b.sort(key=len)
```

```
En [66]: b
```

```
Out[66]: ['Él', 'vivo', 'seis', 'pequeño', 'zorros']
```

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

Búsqueda binaria y mantenimiento de una

lista ordenada El módulo bisect incorporado implementa la búsqueda binaria y la inserción en una lista ordenada. bisect.bisect encuentra la ubicación donde se debe insertar un elemento para mantenerlo ordenado, mientras que bisect.insort en realidad inserta el elemento en esa ubicación:

```
En [67]: import bisecar
```

```
En [68]: c = [1, 2, 2, 2, 3, 4, 7]
```

```
En [69]: bisect.bisect(c, 2)
```

```
Salida[69]: 4
```

```
En [70]: bisect.bisect(c, 5)
```

```
Salida[70]: 6
```

En [71]: `bisect.insort(c, 6)`

En [72]: `c`

Salida[72]: `[1, 2, 2, 2, 3, 4, 6, 7]`



Las funciones del módulo Bisect no verifican si la lista está ordenada, ya que hacerlo sería computacionalmente costoso. Por lo tanto, usarlos con una lista desordenada tendrá éxito sin errores, pero puede conducir a resultados incorrectos.

rebanar

Puede seleccionar secciones de la mayoría de los tipos de secuencia utilizando la notación de división, que en su forma básica consiste en iniciar: detener pasado al operador de indexación []:

En [73]: `secuencia = [7, 2, 3, 7, 5, 6, 0, 1]`

En [74]: `sec[1:5]`

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

Los cortes también se pueden asignar con una secuencia:

En [75]: `sec[3:4] = [6, 3]`

En [76]: `siguiente`

Salida[76]: `[7, 2, 3, 6, 3, 5, 6, 0, 1]`

Si bien se incluye el elemento en el índice de inicio, el índice de finalización no se incluye, por lo que el número de elementos en el resultado es detener - iniciar.

Se puede omitir el inicio o la parada, en cuyo caso por defecto son el inicio de la secuencia y el final de la secuencia, respectivamente:

En [77]: `sec[:5]`

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

En [78]: `sec[3:]`

Salida[78]: `[6, 3, 5, 6, 0, 1]`

Los índices negativos dividen la secuencia en relación con el final:

En [79]: `sec[-4:]`

Salida[79]: `[5, 6, 0, 1]`

En [80]: `sec[-6:-2]`

Salida[80]: `[6, 3, 5, 6]`

Se necesita un poco de tiempo para acostumbrarse a la semántica de corte, especialmente si viene de R o MATLAB. Consulte la [Figura 3-1](#) para ver una ilustración útil de dividir con enteros positivos y negativos. En la figura, los índices se muestran en los "bordes de los bins" para ayudar a mostrar dónde comienzan y terminan las selecciones de sectores usando índices positivos o negativos.

También se puede usar un paso después de dos puntos para, por ejemplo, tomar cualquier otro elemento:

```
En [81]: sec[::2]
Salida[81]: [7, 3, 3, 6, 1]
```

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

```
En [82]: sec[::-1]
Salida[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

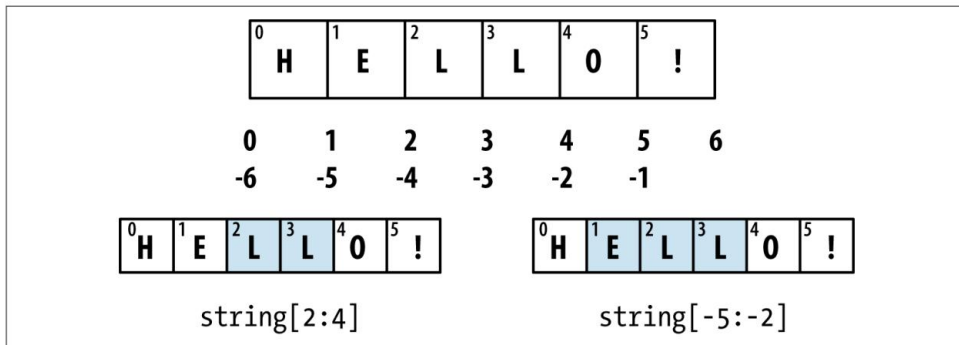


Figura 3-1. Ilustración de las convenciones de corte de Python

Funciones de secuencia incorporadas Python

tiene un puñado de funciones de secuencia útiles con las que debe familiarizarse y usar en cualquier oportunidad.

enumerar

Cuando se itera sobre una secuencia, es común querer realizar un seguimiento del índice del elemento actual. Un enfoque de hágalo usted mismo se vería así:

```
i = 0
para el valor en la colección:
    # hacer algo con valor
    yo += 1
```

Dado que esto es tan común, Python tiene una función integrada, `enumerar`, que devuelve una secuencia de (i, valor) tuplas:

```
para i, valor en enumerar (colección): # hacer algo con
    valor
```

Cuando está indexando datos, un patrón útil que usa `enumerate` es calcular un dictado que asigna los valores de una secuencia (que se supone que son únicos) a sus ubicaciones en la secuencia:

```
En [83]: alguna_lista = ['foo', 'bar', 'baz']
```

```
En [84]: mapeo = {}
```

```
En [85]: for i, v in enumerate(some_list): mapping[v]
        .....:             = i
```

```
En [86]: mapeo
Salida[86]: {'barra': 1, 'baz': 2, 'foo': 0}
```

ordenado

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

```
En [87]: ordenado ([7, 1, 2, 6, 0, 3, 2])
Salida[87]: [0, 1, 2, 2, 3, 6, 7]
```

```
En [88]: sorted(' carrera de caballos')
Fuera[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

La función `ordenada` acepta los mismos argumentos que el método de clasificación en las listas.

zip

zip “empareja” los elementos de una serie de listas, tuplas u otras secuencias para crear una lista de tuplas:

```
En [89]: seq1 = ['foo', 'bar', 'baz']

En [90]: seq2 = ['uno', 'dos', 'tres']

En [91]: comprimido = zip(seq1, seq2)

En [92]: lista (comprimido)
Fuera[92]: [('foo', 'uno'), ('bar', 'dos'), ('baz', 'tres')]
```

zip puede tomar un número arbitrario de secuencias, y el número de elementos que produce está determinado por la secuencia más corta:

```
En [93]: seq3 = [Falso, Verdadero]

En [94]: list(zip(seq1, seq2, seq3))
Fuera[94]: [('foo', 'uno', Falso), ('bar', 'dos', Verdadero)]
```

Un uso muy común de zip es iterar simultáneamente sobre múltiples secuencias, posiblemente también combinado con `enumerar`:

```
En [95]: for i, (a, b) in enumerate(zip(seq1, seq2)): print('{0}: {1},
        .....:                {2}'.format(i, a, b))
        .....:
0: foo, uno 1:
bar, dos 2:
baz, tres
```

Dada una secuencia "comprimida", se puede aplicar zip de una manera inteligente para "descomprimir" la secuencia. Otra forma de pensar en esto es convertir una lista de filas en una lista de columnas. La sintaxis, que parece un poco mágica, es:

```
En [96]: lanzadores = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
.....:                  ('Schilling', 'Curt')]
```

```
En [97]: nombres, apellidos = zip(*lanzadores)
```

```
En [98]: nombres
Fuera[98]: ('Nolan', 'Roger', 'Schilling')
```

```
En [99]: apellidos
Fuera[99]: ('Ryan', 'Clemens', 'Curt')
```

invertido

invertido itera sobre los elementos de una secuencia en orden inverso:

```
En [100]: lista (invertida (rango (10)))
Salida[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Tenga en cuenta que invertido es un generador (que se analizará con más detalle más adelante), por lo que no crea la secuencia invertida hasta que se materializa (por ejemplo, con una lista o un bucle for).

dictar

dict es probablemente la estructura de datos integrada de Python más importante. Un nombre más común para él es mapa hash o matriz asociativa. Es una colección de tamaño flexible de pares clave-valor, donde clave y valor son objetos de Python. Un enfoque para crear uno es usar llaves {} y dos puntos para separar claves y valores:

```
En [101]: vacío_dict = {}
```

```
En [102]: d1 = {'a': 'algún valor', 'b': [1, 2, 3, 4]}
```

```
En [103]: d1
Salida[103]: {'a': 'algún valor', 'b': [1, 2, 3, 4]}
```

Puede acceder, insertar o configurar elementos usando la misma sintaxis que para acceder a elementos de una lista o tupla:

```
En [104]: d1[7] = 'un entero'
```

```
En [105]: d1
Out[105]: {'a': 'algún valor', 'b': [1, 2, 3, 4], 7: 'un número entero'}
```

```
En [106]: d1['b']
Salida[106]: [1, 2, 3, 4]
```

Puede verificar si un dictado contiene una clave utilizando la misma sintaxis utilizada para verificar si una lista o tupla contiene un valor:

```
En [107]: 'b' en d1
```

```
Fuera[107]: Verdadero
```

Puede eliminar valores utilizando la palabra clave del o el método pop (que simultáneamente devuelve el valor y elimina la clave):

```
En [108]: d1[5] = 'algún valor'
```

```
En [109]: d1
```

```
Out[109]:
```

```
{'a': 'algún valor', 'b': [1,
 2, 3, 4], 7: 'un entero',
 5: 'algún valor'}
```

```
En [110]: d1['dummy'] = 'otro valor'
```

```
En [111]: d1
```

```
Out[111]:
```

```
{'a': 'algún valor', 'b': [1,
 2, 3, 4], 7: 'un entero',
 5: 'algún valor',
 'ficticio': 'otro valor'
}
```

```
En [112]: del d1[5]
```

```
En [113]: d1
```

```
Out[113]:
```

```
{'a': 'algún valor', 'b': [1,
 2, 3, 4], 7: 'un entero',
 'ficticio': 'otro valor'}
```

```
En [114]: ret = d1.pop('dummy')
```

```
En [115]: ret
```

```
Out[115]: 'otro valor'
```

```
En [116]: d1
```

```
Out[116]: {'a': 'algún valor', 'b': [1, 2, 3, 4], 7: 'un número entero'}
```

El método de claves y valores le brinda iteradores de las claves y valores del dict, respectivamente. Si bien los pares clave-valor no están en ningún orden en particular, estas funciones generan las claves y los valores en el mismo orden:

```
En [117]: lista (d1.keys())
```

```
Salida[117]: ['a', 'b', 7]
```

```
En [118]: lista(d1.valores())
Out[118]: ['algún valor', [1, 2, 3, 4], 'un número entero']
```

Puede fusionar un dict en otro usando el método de actualización :

```
En [119]: d1.update({'b': 'foo', 'c': 12})

En [120]: d1
Out[120]: {'a': 'algún valor', 'b': 'foo', 7: 'un entero', 'c': 12}
```

El método de actualización cambia los dictados en el lugar, por lo que cualquier clave existente en los datos pasados para actualizar tendrá sus valores antiguos descartados.

Creación de dictados a partir de

secuencias En ocasiones, es común terminar con dos secuencias que desea emparejar por elementos en un dictado. Como primer corte, puede escribir un código como este:

```
mapeo = {}
para clave, valor en zip (key_list, value_list):
    mapeo[clave] = valor
```

Dado que un dict es esencialmente una colección de 2 tuplas, la función dict acepta una lista de 2 tuplas:

```
En [121]: mapeo = dict(zip(rango(5), invertido(rango(5))))

En [122]: mapeo
Salida[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Más adelante hablaremos sobre las comprensiones de dictados, otra forma elegante de construir dictados.

Valores predeterminados

Es muy común tener una lógica como:

```
if clave en some_dict:
    valor = some_dict[key]
else:
    value = default_value
```

Por lo tanto, los métodos dict get y pop pueden tomar un valor predeterminado para devolver, de modo que el bloque if-else anterior se puede escribir simplemente como:

```
valor = some_dict.get(clave, valor_predeterminado)
```

get por defecto devolverá None si la clave no está presente, mientras que pop generará una excepción. Con la configuración de valores, un caso común es que los valores en un dict sean otras colecciones, como listas. Por ejemplo, podría imaginar clasificar una lista de palabras por sus primeras letras como un dictado de listas:

```
En [123]: palabras = ['manzana', 'murciélago', 'barra', 'átomo', 'libro']

En [124]: por_letra = {}
```

```

En [125]: para palabra en palabras:
.....:     letra = palabra[0] si
.....:     letra no en by_letter:
.....:         by_letter[letra] = [palabra] else:
.....:
.....:         by_letter[letra].append(palabra)
.....:

En [126]: por_letra
Salida[126]: {'a': ['manzana', 'átomo'], 'b': ['murciélago', 'barra', 'libro']}

```

El método setdefault dict es precisamente para este propósito. El bucle for anterior se puede reescribir como:

```

para palabra en
    palabras: letra =
        palabra[0] by_letter.setdefault(letra, []).append(palabra)

```

El módulo de colecciones incorporado tiene una clase útil, defaultdict, que lo hace aún más fácil. Para crear uno, pasa un tipo o función para generar el valor predeterminado para cada ranura en el dict:

```

from collections import defaultdict by_letter
= defaultdict(list) for word in words:

    by_letter[palabra[0]].append(palabra)

```

Tipos de claves de

dictado válidos Si bien los valores de un dictado pueden ser cualquier objeto de Python, las claves generalmente deben ser objetos inmutables como tipos escalares (int, float, string) o tuplas (todos los objetos en la tupla también deben ser inmutables). El término técnico aquí es hashability. Puede verificar si un objeto es hashable (se puede usar como clave en un dict) con la función hash :

```

En [127]: hash('cadena')
Salida[127]: 5023931463650008331

```

```

En [128]: hash((1, 2, (2, 3)))
Salida[128]: 1097636502276347782

```

```

En [129]: hash((1, 2, [2, 3])) # falla porque las listas son mutables

```

```

TypeError                                Rastreo (última llamada más reciente)
<ipython-input-129-800cd14ba8be> in <module>() ----> 1
hash((1, 2, [2, 3])) # falla porque las listas son mutables
TypeError: tipo no modificable : 'lista'

```

Para usar una lista como clave, una opción es convertirla en una tupla, que se puede codificar siempre que sus elementos también lo hagan:

En [130]: `d = {}`

En [131]: `d[tupla([1, 2, 3])] = 5`

En [132]: `d`

Salida[132]: `{(1, 2, 3): 5}`

colocar

Un conjunto es una colección desordenada de elementos únicos. Puede pensar en ellos como dictados, pero solo claves, sin valores.

Un conjunto se puede crear de dos maneras: mediante la función de conjunto o mediante un conjunto literal con llaves:

En [133]: `conjunto ([2, 2, 2, 1, 3, 3])`

Salida[133]: `{1, 2, 3}`

En [134]: `{2, 2, 2, 1, 3, 3}`

Salida[134]: `{1, 2, 3}`

Los conjuntos admiten operaciones de conjuntos matemáticos como unión, intersección, diferencia y diferencia simétrica. Considere estos dos conjuntos de ejemplo:

En [135]: `a = {1, 2, 3, 4, 5}`

En [136]: `b = {3, 4, 5, 6, 7, 8}`

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

En [137]: `a.union(b)`

Salida[137]: `{1, 2, 3, 4, 5, 6, 7, 8}`

En [138]: `un | b`

Salida[138]: `{1, 2, 3, 4, 5, 6, 7, 8}`

La intersección contiene los elementos que ocurren en ambos conjuntos. Se puede utilizar el operador `&` o el método de intersección :

En [139]: `a.intersección(b)`

Salida[139]: `{3, 4, 5}`

En [140]: `a & b`

Salida[140]: `{3, 4, 5}`

Consulte la [Tabla 3-1](#) para obtener una lista de los métodos de ajuste más utilizados.

Tabla 3-1. Operaciones de conjuntos de Python

Función	Alternativa sintaxis	Descripción
a.añadir(x)	N / A	Añadir el elemento x al conjunto a
una limpieza()	N / A	Restablezca el conjunto a a un estado vacío, descartando todos sus elementos
a.quitar(x)	N / A	Eliminar el elemento x del conjunto a
Un estallido()	N / A	Eliminar un elemento arbitrario del conjunto a, elevando KeyError si el conjunto está vacío
a.unión(b)	un b	Todos los elementos únicos en a y b
a.actualizar(b)	un = segundo	Establecer el contenido de a para que sea la unión de los elementos en a y b
a.intersección(b)	a & b	Todos los elementos tanto en a como en b
a.intersection_update(b)	un &= segundo	Establezca el contenido de a para que sea la intersección de la elementos en a y b
a.diferencia(b)	un-b	Los elementos de a que no están en b
a.difference_update(b)	un -= segundo	Establecer a a los elementos en a que no están en b
a.diferencia_simétrica(b)	a ^ segundo	Todos los elementos en a o b pero no en ambos
a.diferencia_simétrica_actualización(b) a ^= b		Establezca a para que contenga los elementos en a o b pero no ambos
a.es un subconjunto(b)	N / A	Verdadero si los elementos de a están todos contenidos en b
a.essuperconjunto(b)	N / A	Verdadero si los elementos de b están todos contenidos en a
a.es disjunto(b)	N / A	Verdadero si a y b no tienen elementos en común

Todas las operaciones de conjuntos lógicos tienen contrapartes en el lugar, que le permiten reemplace el contenido del conjunto en el lado izquierdo de la operación con el resultado. Para conjuntos muy grandes, esto puede ser más eficiente:

```
En [141]: c = a.copia()

En [142]: c |= b

En [143]: c
Salida[143]: {1, 2, 3, 4, 5, 6, 7, 8}

En [144]: d = a.copia()

En [145]: d &= b

En [146]: d
Salida[146]: {3, 4, 5}
```

Al igual que los dictados, los elementos establecidos generalmente deben ser inmutables. Para tener elementos tipo lista, usted debe convertirlo en una tupla:

```
En [147]: mis_datos = [1, 2, 3, 4]
```

```
En [148]: mi_conjunto = {tupla(mis_datos)}
```

```
En [149]: mi_conjunto
```

```
Salida[149]: {(1, 2, 3, 4)}
```

También puede verificar si un conjunto es un subconjunto de (está contenido en) o un superconjunto de (contiene todos los elementos de) otro conjunto:

```
En [150]: un_conjunto = {1, 2, 3, 4, 5}
```

```
En [151]: {1, 2, 3}.issubset(a_set)
```

```
Fuera[151]: Verdadero
```

```
En [152]: a_set.issuperset({1, 2, 3})
```

```
Salida[152]: Verdadero
```

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

```
En [153]: {1, 2, 3} == {3, 2, 1}
```

```
Salida[153]: Verdadero
```

Comprensiones de listas, conjuntos y dictados Las

comprensiones de listas son una de las características más queridas del lenguaje Python. Le permiten formar de manera concisa una nueva lista filtrando los elementos de una colección, transformando los elementos que pasan el filtro en una expresión concisa. Toman la forma básica:

```
[expr para val en colección si condición]
```

Esto es equivalente al siguiente bucle for :

```
resultado = []
para val en colección: si
    condición:
        resultado.append(expr)
```

La condición de filtro se puede omitir, dejando solo la expresión. Por ejemplo, dada una lista de cadenas, podríamos filtrar las cadenas con una longitud de 2 o menos y también convertirlas a mayúsculas de esta manera:

```
En [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

```
En [155]: [x.upper() para x en cadenas si len(x) > 2]
```

```
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Las comprensiones de conjuntos y dictados son una extensión natural, produciendo conjuntos y dictados de una manera idiomáticamente similar en lugar de listas. Una comprensión dictada se ve así:

```
dict_comp = {key-expr : value-expr para el valor en la colección
              si condición}
```

Una comprensión de conjunto se parece a la comprensión de lista equivalente, excepto que lleva llaves en lugar de corchetes:

```
set_comp = {expr para valor en colección si condición}
```

Al igual que las comprensiones de listas, las comprensiones de conjuntos y dictados son en su mayoría conveniencias, pero de manera similar pueden hacer que el código sea más fácil de escribir y leer. Considere la lista de cadenas de antes. Supongamos que queremos un conjunto que contenga solo las longitudes de las cadenas contenidas en la colección; Podríamos calcular esto fácilmente usando una comprensión establecida:

```
En [156]: longitudes_únicas = {len(x) para x en cadenas}
```

```
En [157]: longitudes_únicas
```

```
Salida[157]: {1, 2, 3, 4, 6}
```

También podríamos expresar esto de manera más funcional utilizando la función de mapa, que se presenta en breve:

```
En [158]: set(mapa(longitud, cadenas))
```

```
Salida[158]: {1, 2, 3, 4, 6}
```

Como ejemplo simple de comprensión de dictados, podríamos crear un mapa de búsqueda de estas cadenas en sus ubicaciones en la lista:

```
En [159]: loc_mapping = {val : índice por índice, val en enumerar (cadenas)}
```

```
En [160]: loc_mapping
```

```
Salida[160]: {'a': 0, 'como': 1, 'murciélago': 2, 'coche': 3, 'paloma': 4, 'pitón': 5}
```

Comprensiones de listas anidadas

Supongamos que tenemos una lista de listas que contienen algunos nombres en inglés y español:

```
En [161]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
.....:                ['María', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

Es posible que haya obtenido estos nombres de un par de archivos y haya decidido organizarlos por idioma.

Ahora, supongamos que queremos obtener una sola lista que contenga todos los nombres con dos o más e en ellos. Ciertamente podríamos hacer esto con un bucle for simple :

```
names_of_interest = [] para
nombres en all_data:
    suficientes_es = [nombre por nombre en nombres si nombre.cuenta('e') >=
2] nombres_de_interes.extend(suficientes)
```

En realidad, puede envolver toda esta operación en una sola comprensión de lista anidada, que se verá así:

```
En [162]: resultado = [nombre para nombres en todos los datos para nombre en
.....:                 nombres si nombre.cuenta('e') >= 2]
```

```
En [163]: resultado
```

```
Fuera[163]: ['Steven']
```

Al principio, las comprensiones de listas anidadas son un poco difíciles de entender. Las partes for de la comprensión de la lista se organizan de acuerdo con el orden de anidamiento, y cualquier condición de filtro se coloca al final como antes. Aquí hay otro ejemplo en el que "aplanamos" una lista de tuplas de enteros en una lista simple de enteros:

```
En [164]: algunas_tuplas = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
En [165]: flattened = [x para tup en some_tuplas para x en tup]
```

```
En [166]: aplanado
```

```
Salida[166]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Tenga en cuenta que el orden de las expresiones for sería el mismo si escribiera un bucle for anidado en lugar de una lista de comprensión:

```
aplanado = []
```

```
for tup en some_tuplas:
    for x in tup:
        flattened.append(x)
```

Puede tener arbitrariamente muchos niveles de anidamiento, aunque si tiene más de dos o tres niveles de anidamiento, probablemente debería comenzar a preguntarse si esto tiene sentido desde el punto de vista de la legibilidad del código. Es importante distinguir la sintaxis que se acaba de mostrar de una comprensión de lista dentro de una comprensión de lista, que también es perfectamente válida:

```
En [167]: [[x for x in tup] for tup in some_tuplas]
```

```
Fuera[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Esto produce una lista de listas, en lugar de una lista plana de todos los elementos internos.

3.2 Funciones

Las funciones son el método principal y más importante de organización y reutilización de código en Python. Como regla general, si prevé que necesitará repetir el mismo código o uno muy similar más de una vez, puede valer la pena escribir una función reutilizable. Las funciones también pueden ayudar a que su código sea más legible dando un nombre a un grupo de sentencias de Python.

Las funciones se declaran con la palabra clave def y se devuelven con la palabra clave return :

```
def mi_funcion(x, y, z=1.5):
    si z > 1:
        devuelve * (x + y)
    z
    sino: devuelve z / (x + y)
```

No hay problema con tener varias declaraciones de devolución. Si Python llega al final de una función sin encontrar una declaración de retorno, se devuelve Ninguno automáticamente.

Cada función puede tener argumentos posicionales y argumentos de palabras clave. Los argumentos de palabras clave se usan más comúnmente para especificar valores predeterminados o argumentos opcionales. En la función anterior, x e y son argumentos posicionales mientras que z es un argumento de palabra clave. Esto significa que la función se puede llamar de cualquiera de estas formas:

```
mi_funcion(5, 6, z=0.7)
mi_funcion(3.14, 7, 3.5)
mi_funcion(10, 20)
```

La principal restricción en los argumentos de función es que los argumentos de palabras clave deben seguir a los argumentos posicionales (si los hay). Puede especificar argumentos de palabras clave en cualquier orden; esto lo libera de tener que recordar en qué orden se especificaron los argumentos de la función y solo cuáles son sus nombres.



También es posible usar palabras clave para pasar argumentos posicionales. En el ejemplo anterior, también podríamos haber escrito:

```
mi_funcion(x=5, y=6, z=7)
mi_funcion(y=6, x=5, z=7)
```

En algunos casos, esto puede ayudar con la legibilidad.

Espacios de nombres, ámbito y funciones locales Las

funciones pueden acceder a variables en dos ámbitos diferentes: global y local. Un nombre alternativo y más descriptivo que describe un ámbito variable en Python es un espacio de nombres. Todas las variables que se asignan dentro de una función de forma predeterminada se asignan al espacio de nombres local. El espacio de nombres local se crea cuando se llama a la función y se llena inmediatamente con los argumentos de la función. Una vez finalizada la función, el espacio de nombres local se destruye (con algunas excepciones que están fuera del alcance de este capítulo). Considere la siguiente función:

```
def func(): a
    = [] for
    i in range(5):
        a.append(i)
```

Cuando se llama a func(), se crea la lista vacía a, se agregan cinco elementos y luego se destruye a cuando la función sale. Supongamos que, en cambio, hubiéramos declarado a de la siguiente manera:

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Es posible asignar variables fuera del alcance de la función, pero esas variables deben declararse como globales a través de la palabra clave global :

```
En [168]: a = Ninguno

En [169]: def bind_a_variable(): global
.....:     aa =
.....:     [] .....:
.....:     bind_a_variable()
.....:

En [170]: imprimir(a) []
```



Generalmente desaconsejo el uso de la palabra clave global . Normalmente, las variables globales se utilizan para almacenar algún tipo de estado en un sistema. Si te encuentras usando muchos de ellos, puede indicar una necesidad de programación orientada a objetos (usando clases).

Devolver valores múltiples Cuando

programé por primera vez en Python después de haber programado en Java y C++, una de mis funciones favoritas fue la capacidad de devolver valores múltiples desde una función con una sintaxis simple. Aquí hay un ejemplo:

```
definición
f(): a =
5 b = 6
c = 7
devuelve a, b, c
```

```
a, b, c = f()
```

En el análisis de datos y otras aplicaciones científicas, es posible que se encuentre haciendo esto con frecuencia. Lo que sucede aquí es que la función en realidad solo devuelve un objeto, es decir, una tupla, que luego se desempaqueta en las variables de resultado. En el ejemplo anterior, podríamos haber hecho esto en su lugar:

```
valor_devuelto = f()
```

En este caso, `return_value` sería una tupla de 3 con las tres variables devueltas. Una alternativa potencialmente atractiva para devolver múltiples valores como antes podría ser devolver un dict en su lugar:

```
definición
f(): a =
5 b = 6
c = 7
devuelve {'a': a, 'b': b, 'c': c}
```

Esta técnica alternativa puede ser útil según lo que intente hacer.

Las funciones son objetos

Dado que las funciones de Python son objetos, se pueden expresar fácilmente muchas construcciones que son difíciles de hacer en otros lenguajes. Supongamos que estábamos haciendo una limpieza de datos y necesitábamos aplicar un montón de transformaciones a la siguiente lista de cadenas:

```
En [171]: estados = ['Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FLOrIda', 'south carolina##', 'West
.....:              virginia?']
```

Cualquiera que haya trabajado alguna vez con datos de encuestas enviados por usuarios ha visto resultados confusos como estos. Deben suceder muchas cosas para que esta lista de cadenas sea uniforme y esté lista para el análisis: eliminar los espacios en blanco, eliminar los símbolos de puntuación y estandarizar con mayúsculas adecuadas. Una forma de hacer esto es usar métodos de cadena incorporados junto con el módulo de biblioteca estándar para expresiones regulares:

```
importar re

def clean_strings(cadenas):
    resultado = []
    for valor en cadenas:
        valor = valor.strip()
        valor = re.sub('[#?]', '', valor)
        valor = valor.titulo()
        resultado.agregar(valor)
    resultado.devuelto
```

El resultado se ve así:

```
En [173]: clean_strings(estados)
Fuera[173]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'Carolina del Sur',
 'Virginia del Oeste']
```

Un enfoque alternativo que puede resultarle útil es hacer una lista de las operaciones que desea aplicar a un conjunto particular de cadenas:

```
def remove_puntuación(valor):
    volver re.sub('[#?]', '', valor)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(cadenas, operaciones):
    resultado = []
    para valor en cadenas:
        para función en ops:
```

```

        valor = función(valor)
    resultado.agregar(valor)
devolver resultado

```

Luego tenemos lo siguiente:

```

En [175]: clean_strings(estados, clean_ops)
Fuera[175]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'Carolina del Sur',
 'Virginia del Oeste']

```

Un patrón más funcional como este le permite modificar fácilmente cómo se transforman las cadenas a un nivel muy alto. La función `clean_strings` ahora también es más reutilizable y genérica.

Puede usar funciones como argumentos para otras funciones, como la función de mapa integrada, que aplica una función a una secuencia de algún tipo:

```

En [176]: para x en el mapa (remove_punctuation, estados):
.....:     imprimir (x)
Alabama
Georgia
georgia
georgia
Florida
Carolina del Sur
Virginia del Oeste

```

Funciones anónimas (Lambda) Python tiene

soporte para las llamadas funciones anónimas o lambda, que son una forma de escribir funciones que consisten en una sola declaración, cuyo resultado es el valor de retorno. Se definen con la palabra clave `lambda`, que no tiene otro significado que "estamos declarando una función anónima":

```

def función_corto(x):
    volver x * 2

equiv_anon = lambda x: x * 2

```

Normalmente me refiero a estas como funciones lambda en el resto del libro. Son especialmente convenientes en el análisis de datos porque, como verá, hay muchos casos en los que las funciones de transformación de datos tomarán funciones como argumentos. A menudo es menos tipeo (y más claro) pasar una función lambda en lugar de escribir una declaración de función completa o incluso asignar la función lambda a una variable local. Por ejemplo, considere este ejemplo tonto:

```
def aplicar_a_la_lista(alguna_lista, f):
    devuelve [f(x) por x en alguna_lista]
```

```
enteros = [4, 0, 1, 5, 6]
apply_to_list(enteros, lambda x: x * 2)
```

También podría haber escrito `[x * 2 para x en enteros]`, pero aquí pudimos pasar sucintamente un operador personalizado a la función `apply_to_list`.

Como otro ejemplo, suponga que desea ordenar una colección de cadenas por el número de letras distintas en cada cadena:

```
En [177]: cadenas = ['foo', 'tarjeta', 'bar', 'aaaa', 'abab']
```

Aquí podríamos pasar una función lambda al método de clasificación de la lista :

```
En [178]: strings.sort(key=lambda x: len(set(list(x))))
```

```
En [179]: cadenas
```

```
Salida[179]: ['aaaa', 'foo', 'abab', 'bar', 'tarjeta']
```



Una de las razones por las que las funciones lambda se denominan funciones anónimas es que, a diferencia de las funciones declaradas con la palabra clave `def`, la función objeto en sí nunca recibe un atributo `__name__` explícito.

Currying: aplicación de argumentos parciales Currying es

una jerga informática (llamada así por el matemático Haskell Curry) que significa derivar nuevas funciones de las existentes mediante la aplicación de argumentos parciales. Por ejemplo, supongamos que tenemos una función trivial que suma dos números:

```
def suma_números(x, y):
    devuelve x + y
```

Usando esta función, podríamos derivar una nueva función de una variable, `suma_cinco`, que suma 5 a su argumento:

```
sumar_cinco = lambda y: suma_números(5, y)
```

Se dice que el segundo argumento de `add_numbers` está currado. No hay nada muy elegante aquí, ya que todo lo que hemos hecho realmente es definir una nueva función que llama a una función existente.

El módulo integrado de funciones puede simplificar este proceso utilizando la función parcial :

```
de functools importar parcial
add_five = parcial (add_numbers, 5)
```

Generadores

Tener una forma consistente de iterar sobre secuencias, como objetos en una lista o líneas en un archivo, es una característica importante de Python. Esto se logra mediante el protocolo iterador, una forma genérica de hacer que los objetos sean iterables. Por ejemplo, iterar sobre un dict produce las claves de dict:

```
En [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}
```

```
En [181]: for key in some_dict:
.....:     print(key)
a
b
c
```

Cuando escribe para clave en some_dict, el intérprete de Python primero intenta crear un iterador a partir de some_dict:

```
En [182]: dict_iterator = iter(some_dict)
```

```
En [183]: dict_iterator
```

```
Salida[183]: <dict_keyiterator en 0x7fbbd5a9f908>
```

Un iterador es cualquier objeto que le dará objetos al intérprete de Python cuando se usa en un contexto como un bucle for . La mayoría de los métodos que esperan una lista o un objeto similar a una lista también aceptarán cualquier objeto iterable. Esto incluye métodos integrados como min, max y sum, y constructores de tipos como list y tuple:

```
En [184]: lista(dict_iterator)
```

```
Salida[184]: ['a', 'b', 'c']
```

Un generador es una forma concisa de construir un nuevo objeto iterable. Mientras que las funciones normales se ejecutan y devuelven un solo resultado a la vez, los generadores devuelven una secuencia de múltiples resultados de forma perezosa, deteniéndose después de cada uno hasta que se solicita el siguiente. Para crear un generador, use la palabra clave yield en lugar de return en una función:

```
def squares(n=10):
    print('Generando cuadrados de 1 a {}'.format(n ** 2))
    for i in range(1, n + 1): yield i ** 2
```

Cuando realmente llama al generador, no se ejecuta ningún código inmediatamente:

```
En [186]: gen = cuadrados()
```

```
En [187]: gen
```

```
Out[187]: <cuadrados del objeto generador en 0x7fbbd5ab4570>
```

No es hasta que solicita elementos del generador que comienza a ejecutar su código:

```

En [188]: para x en gen:
.....:         print(x, end=' ')
Generando cuadrados del 1 al 100 1 4
9 16 25 36 49 64 81 100

```

Expresiones

generadoras Otra forma aún más concisa de hacer un generador es usando una expresión generadora. Este es un generador análogo a listas, dictados y conjuntos de comprensiones; para crear uno, encierre lo que de otro modo sería una lista de comprensión entre paréntesis en lugar de corchetes:

```

En [189]: gen = (x ** 2 para x en el rango (100))

En [190]: gen
Salida[190]: <objeto generador <genexpr> en 0x7fbbd5ab29e8>

```

Esto es completamente equivalente al siguiente generador más detallado:

```

def _make_gen():
    for x in range(100):
        rendimiento x
** 2 gen = _make_gen()

```

Las expresiones generadoras se pueden usar en lugar de listas por comprensión como argumentos de funciones en muchos casos:

```

En [191]: suma (x ** 2 para x en el rango (100))
Salida[191]: 328350

En [192]: dict((i, i **2) for i in range(5))
Salida[192]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

```

módulo itertools

El módulo itertools de la biblioteca estándar tiene una colección de generadores para muchos algoritmos de datos comunes. Por ejemplo, groupby toma cualquier secuencia y una función, agrupando elementos consecutivos en la secuencia por el valor de retorno de la función. Aquí hay un ejemplo:

```

En [193]: importar itertools

En [194]: primera_letra = lambda x: x[0]

En [195]: nombres = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

En [196]: para letra, nombres en itertools.groupby(nombres, primera_letra):
.....:         print(letra, lista(nombres)) # nombres es un generador
A ['Alan', 'Adán']
W ['Wes', 'Voluntad']
A ['Alberto']
S ['Steven']

```

Consulte [la Tabla 3-2](#) para obtener una lista de algunas otras funciones de `itertools` que con frecuencia he encontrado útiles. Puede consultar [la documentación oficial de Python](#) para obtener más información sobre este útil módulo de utilidad incorporado.

Tabla 3-2. Algunas funciones útiles de `itertools`

Función	Descripción
<code>combinaciones(iterable, k)</code>	Genera una secuencia de todas las k-tuplas posibles de elementos en el orden iterable, ignorando y sin reemplazo (consulte también la función complementaria <code>combinaciones_con_reemplazo</code>)
<code>permutaciones(iterable, k)</code>	Genera una secuencia de todas las k-tuplas posibles de elementos en el orden iterable respetando
<code>groupby(iterable[, keyfunc])</code>	Genera (clave, sub-iterador) para cada producto clave único (*iterables, repeat=1) Genera el producto cartesiano de los iterables de entrada como tuplas, similar a un bucle for anidado

Manejo de errores y excepciones El manejo

correcto de errores o excepciones de Python es una parte importante de la construcción de programas robustos. En las aplicaciones de análisis de datos, muchas funciones solo funcionan en ciertos tipos de entrada. Como ejemplo, la función `float` de Python es capaz de convertir una cadena en un número de coma flotante, pero falla con `ValueError` en entradas incorrectas:

```
En [197]: float('1.2345')
Salida[197]: 1.2345
```

```
En [198]: float('algo')
```

```
ValueError                                Rastreo (última llamada más reciente)
<ipython-input-198-439904410854> en <módulo>() ----> 1
float('algo')
ValueError: no se pudo convertir la cadena en flotante: 'algo'
```

Supongamos que queremos una versión de `float` que falle correctamente y devuelva el argumento de entrada. Podemos hacer esto escribiendo una función que incluya la llamada a `float` en un bloque `try/except`:

```
def intent_float(x): try: return
float(x) except:
    volver x
```

El código en la parte `except` del bloque solo se ejecutará si `float(x)` genera una excepción:

```
En [200]: intent_float('1.2345')
Salida[200]: 1.2345
```

```
En [201]: intent_float('algo')
Fuera[201]: 'algo'
```

Puede notar que float puede generar excepciones distintas de ValueError:

```
En [202]: flotante ((1, 2))
```

```
TypeError                                     Rastreo (última llamada más reciente)
<ipython-input-202-842079ebb635> en <módulo>() ----> 1
float((1, 2))
TypeError: el argumento float() debe ser una cadena o un número, no una 'tupla'
```

Es posible que desee suprimir solo ValueError, ya que un TypeError (la entrada no era una cadena o un valor numérico) podría indicar un error legítimo en su programa. Para hacer eso, escriba el tipo de excepción después de excepto:

```
def intent_float(x): try: return

    float(x) excepto
    ValueError: return x
```

Tenemos entonces:

```
En [204]: intent_float((1, 2))
```

```
TypeError                                     Rastreo (última llamada más reciente)
<ipython-input-204-9bdfd730cead> en <módulo>() ----> 1
intent_float((1, 2)) <ipython-
input-203-3e06b8379b6b> en intent_float(x)
    1 def intent_float(x): 2 try:
    return float(x)
----> 3                     excepto ValueError:
    4
    5 volver x
TypeError: el argumento float() debe ser una cadena o un número, no una 'tupla'
```

Puede capturar varios tipos de excepción escribiendo una tupla de tipos de excepción en su lugar (los paréntesis son obligatorios):

```
def intent_float(x): try: return

    float(x) excepto
    (TypeError, ValueError):
        volver x
```

En algunos casos, es posible que no desee suprimir una excepción, pero desea que se ejecute algún código independientemente de si el código en el bloque de prueba tiene éxito o no. Para hacer esto, use finalmente:

```
f = abierto(ruta, 'w')
```

```
prueba: write_to_file(f)
```

```

finalmente:
    f.cerrar()

```

Aquí, el identificador de archivo `f` siempre se cerrará. De manera similar, puede tener un código que se ejecuta solo si el bloque `try`: tiene éxito usando `else`:

```

f = abierto(ruta, 'w')

intente: write_to_file(f)
excepto:
    print('Failed') else:

print('Succeeded') finalmente:
f.close()

```

Excepciones en IPython

Si se genera una excepción mientras está %ejecutando un script o ejecutando cualquier declaración, IPython imprimirá de forma predeterminada un seguimiento completo de la pila de llamadas (traceback) con unas pocas líneas de contexto alrededor de la posición en cada punto de la pila:

```

En [10]: %ejecutar ejemplos/ipython_bug.py
AssertionError / Rastreo (última llamada más reciente)
/home/wesm/code/pydata-book/examples/ipython_bug.py en <módulo>()
    13     throws_an_exception()
    14
----> 15 llamando_cosas()

/home/wesm/code/pydata-book/examples/ipython_bug.py in llamar_cosas() 11 def
    llamar_cosas(): 12
        funciona_bien()
----> 13     lanza_una_excepción()
    14
    15 llamando_cosas()

/home/wesm/code/pydata-book/examples/ipython_bug.py en throws_an_exception()
    7     a = 5
    8     b = 6
----> 9     afirmar (a + b == 10)
    10
    11 def llamando_cosas():

```

Error de aserción:

Tener contexto adicional por sí mismo es una gran ventaja sobre el intérprete estándar de Python (que no proporciona ningún contexto adicional). Puede controlar la cantidad de contexto que se muestra usando el comando mágico `%xmode`, desde `Simple` (igual que el intérprete estándar de Python) hasta `Verbose` (que incluye valores de argumento de función y

más). Como verá más adelante en el capítulo, puede pasar a la pila (usando las magias `%debug` o `%pdb`) después de que haya ocurrido un error para la depuración post-mortem interactiva.

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 en estructuras de datos de Python. Sin embargo, es importante comprender los conceptos básicos de cómo trabajar con archivos en Python. Afortunadamente, es muy simple, lo cual es una de las razones por las que Python es tan popular para la manipulación de archivos y texto.

Para abrir un archivo para lectura o escritura, utilice la función de apertura integrada con una ruta de archivo relativa o absoluta:

```
En [207]: ruta = 'ejemplos/segismundo.txt'
```

```
En [208]: f = abrir(ruta)
```

De forma predeterminada, el archivo se abre en modo de solo lectura 'r'. Luego podemos tratar el identificador de archivo `f` como una lista e iterar sobre las líneas de la siguiente manera:

```
para línea en f:
```

```
    aprobar
```

Las líneas salen del archivo con los marcadores de fin de línea (EOL) intactos, por lo que a menudo verá código para obtener una lista de líneas sin EOL en un archivo como:

```
En [209]: líneas = [x.rstrip() for x in open(ruta)]
```

```
In [210]: líneas
```

```
Out[210]:
```

```
['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 usa `abrir` para crear objetos de archivo, es importante cerrar explícitamente el archivo cuando haya terminado con él. Al cerrar el archivo, se liberan sus recursos al sistema operativo:

```
En [211]: f.cerrar()
```

Una de las formas de facilitar la limpieza de archivos abiertos es usar la instrucción with :

```
En [212]: con abierto (ruta) como f:
.....:         líneas = [x.rstrip() para x en f]
```

Esto cerrará automáticamente el archivo f al salir del bloque with .

Si hubiésemos tecleado `f = open(ruta, 'w')`, se habría creado un nuevo archivo en ejemplos/segismundo.txt (¡ojo!), sobrescribiendo cualquiera en su lugar. También existe el modo de archivo 'x' , que crea un archivo grabable pero falla si la ruta del archivo ya existe. Consulte la [Tabla 3-3](#) para obtener una lista de todos los modos válidos de lectura/escritura de archivos.

Para archivos legibles, algunos de los métodos más utilizados son leer, buscar y decir. `read` devuelve un cierto número de caracteres del archivo. Lo que constituye un "carácter" está determinado por la codificación del archivo (p. ej., UTF-8) o simplemente bytes sin formato si el archivo se abre en modo binario:

```
En [213]: f = abrir (ruta)

En [214]: f.read(10)
Fuera[214]: 'Sueña el r'

En [215]: f2 = open(ruta, 'rb') # Modo binario

En [216]: f2.read(10)
Fuera[216]: b'Sue\xc3\xb1a el '
```

El método de lectura avanza la posición del identificador de archivo por el número de bytes leídos. `tell` da la posición actual:

```
En [217]: f.decir()
Salida[217]: 11

En [218]: f2.decir()
Salida[218]: 10
```

Aunque leemos 10 caracteres del archivo, la posición es 11 porque se necesitaron tantos bytes para decodificar 10 caracteres usando la codificación predeterminada. Puede verificar la codificación predeterminada en el módulo `sys` :

```
En [219]: importar sistema

En [220]: sys.getdefaultencoding()
Salida[220]: 'utf-8'
```

`seek` cambia la posición del archivo al byte indicado en el archivo:

```
En [221]: f.buscar(3)
Salida[221]: 3

En [222]: f.read(1)
Fuera[222]: 'ñ'
```

Por último, recordamos cerrar los archivos:

```
En [223]: f.cerrar()
```

```
En [224]: f2.cerrar()
```

Tabla 3-3. Modos de archivo 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 falla si la ruta del archivo ya existe
a	Agregar al archivo existente (crear el archivo si aún no existe)
r+	Lee y escribe
b	Agregar al modo para archivos binarios (es decir, 'rb' o 'wb')
t	Modo de texto para archivos (decodificación automática de bytes a Unicode). Este es el valor predeterminado si no se especifica. Agregue t a otros modos para usar esto (es decir, 'rt' o 'xt')

Para escribir texto en un archivo, puede usar los métodos write o writelines del archivo . Por ejemplo, podríamos crear una versión de prof_mod.py sin líneas en blanco como esta:

```
En [225]: con open('tmp.txt', 'w') como identificador:
.....:         handle.writelines(x for x in open(path) if len(x) > 1)
```

```
En [226]: con open('tmp.txt') como f: líneas =
.....:         f.readlines()
```

```
In [227]: líneas
Out[227]:
['Sueña el rico en su riqueza,\n', 'que más
cuidados le ofrece,\n', 'sueña el pobre que
padecen\n', 'su miseria y su pobreza,\n',
'sueña el que a medrar empieza,\n',
'sueña el que afana y finge,\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 la Tabla 3-4 para conocer muchos de los métodos de archivo más utilizados.

Tabla 3-4. Métodos o atributos importantes del archivo de Python

Método	Descripción
leer ([tamaño])	Devuelve datos del archivo como una cadena, con un argumento de tamaño opcional que indica la cantidad de bytes para leer
líneas de lectura ([tamaño])	Devuelve la lista de líneas en el archivo, con argumento de tamaño opcional
escribir (cadena)	Escribir la cadena pasada en el archivo

Método	Descripción
writelines(strings)	Escribe la secuencia pasada de cadenas en el archivo
cerca()	Cierra el mango
enjuagar()	Vacía el búfer de E/S interno en el disco
buscar (pos)	Mover a la posición de archivo indicada (entero)
decir()	Devuelve la posición actual del archivo como un número entero
cerrado	Verdadero si el archivo está cerrado

Bytes y Unicode con archivos EI

comportamiento predeterminado de los archivos de Python (ya sean de lectura o escritura) es el modo de texto, lo que significa que pretende trabajar con cadenas de Python (es decir, Unicode). Esto contrasta con el modo binario, que puede obtener agregando `b` al modo de archivo.

Veamos el archivo (que contiene caracteres no ASCII con codificación UTF-8) de la sección anterior:

```
En [230]: con open(ruta) como f: chars
.....:         = f.read(10)
```

```
En [231]: caracteres
Fuera[231]: 'Sueña el r'
```

UTF-8 es una codificación Unicode de longitud variable, por lo que cuando solicité cierta cantidad de caracteres del archivo, Python lee suficientes bytes (que podrían ser tan solo 10 o tantos como 40 bytes) del archivo para decodificar esa cantidad de caracteres. . Si abro el archivo en modo 'rb' , lea las solicitudes con el número exacto de bytes:

```
En [232]: con open(ruta, 'rb') como f: data =
.....:         f.read(10)
```

```
En [233]: datos
Fuera[233]: b'Sue\x03\xb1a el '
```

Según la codificación del texto, es posible que pueda decodificar los bytes en un objeto `str` usted mismo, pero solo si cada uno de los caracteres Unicode codificados está completamente formado:

```
En [234]: data.decode('utf8')
Fuera[234]: 'Sueña el'
```

```
En [235]: datos[:4].decode('utf8')
```

```
UnicodeDecodeError                                Rastreo (última llamada más reciente)
<ipython-input-235-300e0af10bb7> en <módulo>() ----> 1
datos[:4].decode('utf8')
UnicodeDecodeError: el códec 'utf-8' no puede decodificar el byte 0xc3 en la posición 3: final inesperado
de los datos
```

El modo de texto, combinado con la opción de codificación de abierto, proporciona una manera conveniente de convertir de una codificación Unicode a otra:

```
En [236]: fregadero_ruta = 'sink.txt'
```

```
En [237]: con open(ruta) como fuente:
```

```
..... con open(sink_path, 'xt', encoding='iso-8859-1') como sumidero:
..... sumidero.escribir(fuente.leer())
```

```
En [238]: con open(sink_path, encoding='iso-8859-1') como f: print(f.read(10))
```

```
.....
```

```
Sueña el r
```

Tenga cuidado al usar la búsqueda al abrir archivos en cualquier modo que no sea binario. Si la posición del archivo se encuentra en medio de los bytes que definen un carácter Unicode, las lecturas posteriores generarán un error:

```
En [240]: f = abrir (camino)
```

```
En [241]: f.read(5)
```

```
Fuera[241]: 'Sueña'
```

```
En [242]: f.buscar(4)
```

```
Salida[242]: 4
```

```
En [243]: f.read(1)
```

```
UnicodeDecodeError
```

Rastreo (última llamada más reciente)

```
<ipython-input-243-7841103e33f5> en <módulo>() ----> 1
```

```
f.read(1) /miniconda/
```

```
envs/book-env/lib/python3.6/codecs.py en decode(self, entrada, final) # decodificar entrada (teniendo
```

```
319 en cuenta el búfer) data = self.buffer + input (resultado,
```

```
320 consumido) =
```

```
--> 321 self._buffer_decode(data, self.errors, final
```

```
)
```

```
322 # mantener la entrada sin decodificar hasta la próxima
```

```
323 llamada self.buffer = data[consumed:]
```

```
UnicodeDecodeError: el códec 'utf-8' no puede decodificar el byte 0xb1 en la posición 0: byte de inicio no válido
```

```
En [244]: f.cerrar()
```

Si se encuentra realizando regularmente análisis de datos en datos de texto que no son ASCII, será valioso dominar la funcionalidad Unicode de Python. Consulte [la documentación en línea de Python](#) para obtener más información.

3.4 Conclusión

Con algunos de los conceptos básicos y el entorno y el lenguaje de Python ahora en nuestro haber, es hora de seguir adelante y aprender sobre NumPy y la computación orientada a matrices en Python.

Conceptos básicos de NumPy: matrices y Computación Vectorizada

NumPy, abreviatura de Numerical Python, es uno de los paquetes fundamentales más importantes para la computación numérica en Python. La mayoría de los paquetes computacionales que brindan funcionalidad científica utilizan los objetos de matriz de NumPy como lengua franca para el intercambio de datos.

Estas son algunas de las cosas que encontrarás en NumPy:

- `ndarray`, un arreglo multidimensional eficiente que proporciona operaciones aritméticas orientadas a arreglos rápidos y capacidades de transmisión flexibles.
- Funciones matemáticas para operaciones rápidas en arreglos completos de datos sin tener que escribir bucles.

Herramientas para leer/escribir datos de matriz en el disco y trabajar con mapas de memoria

`archivos`

- Capacidades de álgebra lineal, generación de números aleatorios y transformada de Fourier.

AC API para conectar NumPy con bibliotecas escritas en C, C++ o FORTRAN.

Debido a que NumPy proporciona una API C fácil de usar, es sencillo pasar datos a bibliotecas externas escritas en un lenguaje de bajo nivel y también para que las bibliotecas externas devuelvan datos a Python como matrices NumPy. Esta característica ha convertido a Python en el lenguaje de elección para envolver bases de código C/C++/Fortran heredadas y brindarles una interfaz dinámica y fácil de usar.

Si bien NumPy por sí mismo no proporciona funciones científicas o de modelado, comprender los arreglos de NumPy y la computación orientada a arreglos lo ayudará a usar herramientas con semántica orientada a arreglos, como `pandas`, de manera mucho más efectiva. Desde

NumPy es un tema amplio, cubriré muchas funciones avanzadas de NumPy como la transmisión con más profundidad más adelante (consulte el [Apéndice A](#)).

Para la mayoría de las aplicaciones de análisis de datos, las principales áreas de funcionalidad en las que me centraré son:

- Operaciones de matrices vectorizadas rápidas para la recolección y limpieza de datos, creación de subconjuntos y filtrado, transformación y cualquier otro tipo de cómputo
- Algoritmos de matrices comunes como operaciones de clasificación, únicas y de conjuntos
- Estadísticas descriptivas eficientes y agregación/resumen de datos
- Alineación de datos y manipulaciones de datos relacionales para fusionar y unir juntos conjuntos de datos heterogéneos
- Expresar lógica condicional como expresiones de matriz en lugar de bucles con if-elif otras ramas
- Manipulaciones de datos grupales (agregación, transformación, aplicación de funciones).
ción)

Si bien NumPy proporciona una base computacional para el procesamiento general de datos numéricos, muchos lectores querrán usar pandas como base para la mayoría de los tipos de estadísticas o análisis, especialmente en datos tabulares. pandas también proporciona algunas funciones más específicas de dominio, como la manipulación de series temporales, que no está presente en NumPy.



La computación orientada a matrices en Python se remonta a 1995, cuando Jim Hugunin creó la biblioteca Numeric. Durante los siguientes 10 años, muchas comunidades científicas de programación comenzaron a programar matrices en Python, pero el ecosistema de la biblioteca se fragmentó a principios de la década de 2000. En 2005, Travis Oliphant pudo forjar el proyecto NumPy a partir de los entonces proyectos Numeric y Numarray para unir a la comunidad en torno a un marco informático de matriz única.

Una de las razones por las que NumPy es tan importante para los cálculos numéricos en Python es porque está diseñado para ser eficiente en grandes conjuntos de datos. Hay un número de razones para esto:

- NumPy almacena datos internamente en un bloque de memoria contiguo, independiente de otros objetos integrados de Python. La biblioteca de algoritmos de NumPy escrita en lenguaje C puede operar en esta memoria sin ningún tipo de verificación u otra sobrecarga.
Las matrices NumPy también usan mucha menos memoria que las secuencias integradas de Python.
- Las operaciones NumPy realizan cálculos complejos en arreglos completos sin necesidad de bucles for de Python.

Para darle una idea de la diferencia de rendimiento, considere una matriz NumPy de un millón de enteros y la lista Python equivalente:

```
En [7]: importar numpy como np
```

```
En [8]: my_arr = np.arange(1000000)
```

```
En [9]: mi_lista = lista(rango(1000000))
```

Ahora multipliquemos cada secuencia por 2:

```
In [10]: %time for in range(10): my_arr2 = my_arr Tiempos de CPU: usuario      * 2
20 ms, sys: 50 ms, total: 70 ms Tiempo de pared: 72,4 ms
```

```
En [11]: % de tiempo para __ en el rango (10): mi_lista2 = [x * 2 para x en mi_lista]
tiempos de CPU: usuario 760 ms, sys: 290 ms, total: 1,05 s Tiempo de
pared: 1,05 s
```

Los algoritmos basados en NumPy son generalmente de 10 a 100 veces más rápidos (o más) que sus contrapartes de Python puro y usan mucha menos memoria.

4.1 NumPy ndarray: un objeto de matriz multidimensional

Una de las características clave de NumPy es su objeto de matriz N-dimensional, o ndarray, que es un contenedor rápido y flexible para grandes conjuntos de datos en Python. Las matrices le permiten realizar operaciones matemáticas en bloques completos de datos utilizando una sintaxis similar a las operaciones equivalentes entre elementos escalares.

Para darle una idea de cómo NumPy permite los cálculos por lotes con una sintaxis similar a los valores escalares en los objetos integrados de Python, primero importo NumPy y genero una pequeña matriz de datos aleatorios:

```
En [12]: importar numpy como np
```

```
# Generar algunos datos aleatorios
```

```
En [13]: datos = np.random.randn(2, 3)
```

```
Entrada [14]: salida
de datos
```

```
[14]: matriz ([[ -0.2047,  0.4789, -0.5194],
               [-0.5557,  1.9658,  1.3934]])
```

Luego escribo operaciones matemáticas con datos:

```
Entrada [15]: datos * 10
```

```
Salida [15]:
```

```
matriz ([[ -2.0471,  4.7894, -5.1944],
          [-5.5573, 19.6578,
           13.9341]])
```

```
En [16]: datos + datos
```

```
Fuera[16]:
```

```
matriz ([[ -0.4094, 0.9579, -1.0389],
         [-1.1115, 3.9316, 2.7868]])
```

En el primer ejemplo, todos los elementos se han multiplicado por 10. En el segundo, los valores correspondientes en cada "celda" de la matriz se han sumado entre sí.



En este capítulo y en todo el libro, uso la convención NumPy estándar de usar siempre `import numpy` como `np`. Por supuesto, puedes poner `from numpy import *` en tu código para evitar tener que escribir `np.`, pero te aconsejo que no lo conviertas en un hábito. El espacio de nombres `numpy` es grande y contiene una serie de funciones cuyos nombres entran en conflicto con las funciones integradas de Python (como `min` y `max`).

Un `ndarray` es un contenedor multidimensional genérico para datos homogéneos; es decir, todos los elementos deben ser del mismo tipo. Cada matriz tiene una forma, una tupla que indica el tamaño de cada dimensión y un `dtype`, un objeto que describe el tipo de datos de la matriz:

```
Entrada [17]: data.shape
```

```
Salida[17]: (2, 3)
```

```
Entrada [18]:
```

```
datos.dtype Salida[18]: dtype('float64')
```

Este capítulo le presentará los conceptos básicos del uso de matrices NumPy y debería ser suficiente para continuar con el resto del libro. Si bien no es necesario tener una comprensión profunda de NumPy para muchas aplicaciones de análisis de datos, volverse competente en la programación y el pensamiento orientados a arreglos es un paso clave en el camino para convertirse en un gurú científico de Python.



Cada vez que vea "matriz", "matriz NumPy" o "ndarray" en el texto, con pocas excepciones, todos se refieren a lo mismo: el objeto `ndarray`.

Creación de ndarrays

La forma más sencilla de crear una matriz es utilizar la función `matriz`. Esto acepta cualquier objeto similar a una secuencia (incluidas otras matrices) y produce una nueva matriz NumPy que contiene los datos pasados. Por ejemplo, una lista es un buen candidato para la conversión:

```
En [19]: dato1 = [6, 7.5, 8, 0, 1]
```

```
En [20]: arr1 = np.array(dato1)
```

```
Entrada [21]:
```

```
arr1 Salida[21]: matriz([ 6, 7.5, 8. , 0. , 1. ])
```

Las secuencias anidadas, como una lista de listas de igual longitud, se convertirán en una matriz multidimensional:

```
En [22]: dato2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
En [23]: matriz2 = np.matriz(dato2)
```

```
En [24]: arr2
```

```
Salida[24]:
```

```
matriz([[1, 2, 3, 4], [5, 6, 7, 8]])
```

Dado que data2 era una lista de listas, la matriz NumPy arr2 tiene dos dimensiones con forma inferida de los datos. Podemos confirmar esto inspeccionando los atributos ndim y shape :

```
Entrada [25]: arr2.ndim
```

```
Salida [25]: 2
```

```
Entrada [26]: arr2.forma
```

```
Salida[26]: (2, 4)
```

A menos que se especifique explícitamente (más sobre esto más adelante), np.array intenta inferir un buen tipo de datos para la matriz que crea. El tipo de datos se almacena en un objeto de metadatos dtype especial; por ejemplo, en los dos ejemplos anteriores tenemos:

```
Entrada [27]: arr1.dtype
```

```
Salida[27]: dtype('float64')
```

```
Entrada [28]: arr2.dtype
```

```
Salida[28]: dtype('int64')
```

Además de np.array, hay otras funciones para crear nuevas matrices. Como ejemplos, los ceros y los unos crean matrices de 0 o 1, respectivamente, con una longitud o forma determinada. vacío crea una matriz sin inicializar sus valores a ningún valor en particular. Para crear una matriz de mayor dimensión con estos métodos, pase una tupla para la forma:

```
En [29]: np.ceros(10)
```

```
Salida[29]: matriz([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
En [30]: np.ceros((3, 6))
```

```
Salida[30]:
```

```
matriz([[ 0., 0., 0., 0., 0., 0.], [ 0., 0., 0., 0., 0., 0.],
        [ 0., 0., 0., 0., 0., 0.]])
```

```
En [31]: np.empty((2, 3, 2))
```

```
Salida[31]:
```

```
matriz([[[ 0., 0.], [ 0., 0.], [ 0., 0.]],
```

```
[[ 0., 0.], [ 0.,  
0.], [ 0., 0.]])
```



No es seguro asumir que np.empty devolverá una matriz de ceros.
En algunos casos, puede devolver valores "basura" no inicializados.

arange es una versión con valores de matriz de la función de rango integrada de Python :

```
En [32]: np.arange(15)  
Salida[32]: matriz([ 0, 1, 2, 3, 4, 5 , 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Consulte la [Tabla 4-1](#) para obtener una breve lista de funciones estándar de creación de arreglos. Dado que NumPy se centra en la computación numérica, el tipo de datos, si no se especifica, en muchos casos será float64 (coma flotante).

Tabla 4-1. Funciones de creación de matrices

Función	Descripción
formación	Convierta los datos de entrada (lista, tupla, matriz u otro tipo de secuencia) en un ndarray ya sea infiriendo un dtype o especificando explícitamente un dtype; copia los datos de entrada por defecto
una matriz	Convierta la entrada a ndarray, pero no copie si la entrada ya es un ndarray
aranja	Como el rango incorporado pero devuelve un ndarray en lugar de una lista
unos,	Produce una matriz de todos los 1 con la forma y el tipo dados; ones_like toma otra matriz y produce una matriz de
unos_como	unos de la misma forma y tipo
ceros,	Como ones y ones_like pero produciendo matrices de 0s en su lugar
ceros_como	
vacío,	Cree nuevas matrices asignando nueva memoria, pero no las complete con valores como unos y
vacío como	ceros
lleno,	Producir una matriz de la forma y tipo dados con todos los valores establecidos en el "valor de relleno" indicado
full_like	full_like toma otra matriz y produce una matriz completa de la misma forma y tipo de d ojo, identidad Cree una matriz de
identidad	cuadrada N × N (1s en la diagonal y ceros en otros lugares)

Tipos de datos para ndarrays

El tipo de datos o dtype es un objeto especial que contiene la información (o metadatos, datos sobre datos) que ndarray necesita para interpretar una porción de memoria como un tipo particular de datos:

```
En [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)  
  
En [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)  
  
Entrada [35]: arr1.dtype  
Salida[35]: dtype('float64')
```

```
En [36]: arr2.dtype
Salida[36]: dtype('int32')
```

dtypes son una fuente de flexibilidad de NumPy para interactuar con datos provenientes de otros sistemas En la mayoría de los casos, proporcionan una asignación directamente a un disco subyacente o representación de memoria, lo que facilita la lectura y escritura de flujos binarios de datos a disco y también para conectarse a código escrito en un lenguaje de bajo nivel como C o Fortran. Los dtypes numéricos se nombran de la misma manera: un nombre de tipo, como float o int, sigue seguido por un número que indica el número de bits por elemento. Un valor estándar de punto flotante de precisión doble (lo que se usa debajo del capó en el objeto flotante de Python) ocupa 8 bytes o 64 bits. Por lo tanto, este tipo se conoce en NumPy como float64. Ver

Tabla 4-2 para obtener una lista completa de los tipos de datos compatibles con NumPy.



No se preocupe por memorizar los tipos de dNumPy, especialmente si eres un nuevo usuario. A menudo solo es necesario preocuparse por el general tipo de datos con los que está tratando, ya sea punto flotante, complejo, entero, booleano, cadena u objeto general de Python. Cuando lo necesites más control sobre cómo se almacenan los datos en la memoria y en el disco, conjuntos de datos especialmente grandes, es bueno saber que tiene el control sobre el tipo de almacenamiento.

Tabla 4-2. Tipos de datos NumPy

Escriba el		Descripción
int8, uint8	i1, u1	Tipos enteros de 8 bits (1 byte) con y sin signo
int16, uint16	i2, u2	Tipos enteros de 16 bits con y sin signo
int32, uint32	i4, u4	Tipos enteros de 32 bits con y sin signo
int64, uint64	i8, u8	Tipos enteros de 64 bits con y sin signo
flotar16	f2	Punto flotante de media precisión
flotar32	f4 o f	Coma flotante estándar de precisión simple; compatible con flotador C
flotar64	f8 o d	Punto flotante estándar de doble precisión; compatible con C doble y Objeto flotante de Python
flotar128	f16 o g	Punto flotante de precisión extendida
complejo64, complejo128, complejo256	c8, c16, c32	Números complejos representados por dos flotantes de 32, 64 o 128, respectivamente
bool	?	Tipo booleano que almacena valores verdaderos y falsos
objeto	O	tipo de objeto Python; un valor puede ser cualquier objeto Python
cadena_	S	Tipo de cadena ASCII de longitud fija (1 byte por carácter); por ejemplo, para crear un cadena dtype con longitud 10, use 'S10'
unicode_	tu	Tipo Unicode de longitud fija (número de bytes específico de la plataforma); mismo especificación semántica como string_ (p. ej., 'U10')

Puede convertir o convertir explícitamente una matriz de un dtype a otro utilizando el método `astype` de `ndarray` :

```
En [37]: matriz = np.matriz([1, 2, 3, 4, 5])
```

```
Entrada [38]: arr.dtype
Salida[38]: dtype('int64')
```

```
En [39]: float_arr = arr.astype(np.float64)
```

```
Entrada [40]: float_arr.dtype
Salida[40]: dtype('float64')
```

En este ejemplo, los enteros se convirtieron en punto flotante. Si hago que algunos números de coma flotante sean de tipo entero, la parte decimal se truncará:

```
En [41]: matriz = np.matriz([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
Entrada [42]:
matriz Salida[42]: matriz([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
En [43]: arr.astype(np.int32)
Salida[43]: matriz([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

Si tiene una matriz de cadenas que representan números, puede usar un tipo para convertirlos a forma numérica:

```
En [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
```

```
En [45]: numeric_strings.astype(float)
Salida[45]: matriz([ 1.25, -9.6 42. ]) ,
```



Es importante tener cuidado al usar el tipo `numpy.string_` , ya que los datos de cadena en NumPy tienen un tamaño fijo y pueden truncar la entrada sin previo aviso. pandas tiene un comportamiento listo para usar más intuitivo en datos no numéricos.

Si la conversión fallara por alguna razón (como una cadena que no se puede convertir a `float64`), se generará un `ValueError` . Aquí fui un poco perezoso y escribí `float` en lugar de `np.float64`; NumPy asigna alias a los tipos de Python con sus propios tipos de datos equivalentes.

También puede usar el atributo `dtype` de otra matriz:

```
En [46]: int_array = np.arange(10)
```

```
En [47]: calibres = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
```

```
En [48]: int_array.astype(calibres.dtype)
Salida[48]: matriz([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]
```

Hay cadenas de código de tipo abreviado que también puede usar para referirse a un dtype:

```
En [49]: empty_uint32 = np.empty(8, dtype='u4')
```

```
En [50]: vacío_uint32
```

```
Fuera[50]:
```

```
0, 1075314688, 0, 1075707904, 0,
matriz([[ 1075838976, 0, 1072693248], dtype=uint32)
```



Llamar a `astype` siempre crea una nueva matriz (una copia de los datos), incluso si el nuevo dtype es el mismo que el antiguo dtype.

Aritmética con matrices NumPy

Las matrices son importantes porque le permiten expresar operaciones por lotes en datos sin escribir ningún bucle `for`. Los usuarios de NumPy llaman a esto vectorización. Cualquier aritmética Las operaciones entre matrices de igual tamaño aplican la operación por elementos:

```
En [51]: matriz = np.matriz([[1., 2., 3.], [4., 5., 6.]])
```

```
En [52]: arr
```

```
Fuera[52]:
```

```
matriz([[ 1., 2., 3.],
        [ 4., 5., 6.]])
```

```
Entrada [53]: Arr * Arr
```

```
Salida[53]:
```

```
matriz ([[ 1., 4., 9.],
        [ 16., 25., 36.]])
```

```
En [54]: arr - arr
```

```
Fuera[54]:
```

```
matriz([[ 0., 0., 0.],
        [ 0., 0., 0.]])
```

Las operaciones aritméticas con escalares propagan el argumento escalar a cada elemento en la matriz:

```
En [55]: 1 / arr
```

```
Fuera[55]:
```

```
matriz ([[ 1.      ,  0.5      ,  0.3333],
        [ 0.25     ,  0.2      ,  0.1667]])
```

```
Entrada [56]: Arr ** 0.5
```

```
Salida[56]:
```

```
matriz ([[ 1.      ,  1.4142,  1.7321],
        [ 2.      ,  2.2361,  2.4495]])
```

Las comparaciones entre matrices del mismo tamaño producen matrices booleanas:

```
En [57]: matriz2 = np.matriz([[0., 4., 1.], [7., 2., 12.]])
```

```
Entrada [58]:
```

```
arr2
```

```
Salida[58]:      4., 1., 2.,  
             matriz([[ 0., 12.]])
```

```
En [59]: arr2 > arr
```

```
Salida[59]:
```

```
array([[Falso, Verdadero, Falso],  
       [Verdadero, Falso, Verdadero]], dtype=bool)
```

Las operaciones entre arreglos de diferentes tamaños se llaman transmisión y se analizarán con más detalle en el [Apéndice A](#). No es necesario tener un conocimiento profundo de la transmisión para la mayor parte de este libro.

Indexación básica y corte La indexación

de matriz NumPy es un tema rico, ya que hay muchas maneras en las que puede querer seleccionar un subconjunto de sus datos o elementos individuales. Los arreglos unidimensionales son simples; en la superficie actúan de manera similar a las listas de Python:

```
En [60]: arr = np.arange(10)
```

```
En [61]: arr
```

```
Salida[61]: matriz([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
En [62]: arr[5]
```

```
Salida[62]: 5
```

```
En [63]: arr[5:8]
```

```
Salida[63]: matriz([5, 6, 7])
```

```
En [64]: arr[5:8] = 12
```

```
En [65]: arr
```

```
Salida[65]: matriz([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

Como puede ver, si asigna un valor escalar a un segmento, como en `arr[5:8] = 12`, el valor se propaga (o se transmite de ahora en adelante) a toda la selección. Una primera distinción importante de las listas integradas de Python es que los segmentos de matriz son vistas en la matriz original.

Esto significa que los datos no se copian y cualquier modificación a la vista se reflejará en la matriz de origen.

Para dar un ejemplo de esto, primero creo una porción de `arr`:

```
En [66]: arr_segmento = arr[5:8]
```

```
En [67]: arr_slice
```

```
Salida[67]: matriz([12, 12, 12])
```

Ahora, cuando cambio valores en `arr_slice`, las mutaciones se reflejan en el original

matriz matriz :

```
En [68]: arr_slice[1] = 12345
```

```
En [69]: arr
```

```
Salida[69]: matriz([ 9])    0,      1,      2,      3,      4,    12, 12345,    12,      8,
```

El segmento "desnudo" `[:]` se asignará a todos los valores en una matriz:

```
En [70]: arr_slice[:] = 64
```

```
En [71]: arr
```

```
Salida[71]: matriz([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

Si es nuevo en NumPy, es posible que esto le sorprenda, especialmente si ha utilizado otros lenguajes de programación de matrices que copian datos con más entusiasmo. Como NumPy ha sido diseñado para poder trabajar con matrices muy grandes, podría imaginar problemas de rendimiento y memoria si NumPy insistiera en copiar siempre los datos.



Si desea una copia de una porción de un `ndarray` en lugar de una vista, deberá copiar explícitamente la matriz, por ejemplo, `arr[5:8].copy()`.

Con matrices de mayor dimensión, tiene muchas más opciones. En una matriz bidimensional, los elementos en cada índice ya no son escalares sino matrices unidimensionales:

```
En [72]: matriz2d = np.matriz([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
En [73]: arr2d[2]
```

```
Salida[73]: matriz([7, 8, 9])
```

Por lo tanto, se puede acceder a los elementos individuales de forma recursiva. Pero eso es demasiado trabajo, por lo que puede pasar una lista de índices separados por comas para seleccionar elementos individuales.

Entonces estos son equivalentes:

```
En [74]: arr2d[0][2]
```

```
Salida[74]: 3
```

```
En [75]: arr2d[0, 2]
```

```
Salida[75]: 3
```

Consulte la [Figura 4-1](#) para ver una ilustración de la indexación en una matriz bidimensional. Encuentro útil pensar en el eje 0 como las "filas" de la matriz y el eje 1 como las "columnas".

		axis 1		
		0	1	2
axis 0	0	0, 0	0, 1	0, 2
	1	1, 0	1, 1	1, 2
	2	2, 0	2, 1	2, 2

Figura 4-1. Elementos de indexación en una matriz NumPy

En las matrices multidimensionales, si omite los índices posteriores, el objeto devuelto será un ndarray de dimensión inferior que constará de todos los datos de las dimensiones superiores. Entonces, en la matriz $2 \times 2 \times 3$ arr3d:

```
En [76]: matriz3d = np.matriz([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
En [77]: arr3d
```

```
Salida[77]:
```

```
matriz([[[ 1, 2, 3], [ 4, 5, 6]], [[ 7, 8, 9],
[10, 11, 12]])
```

arr3d[0] es una matriz de 2×3 :

```
En [78]: arr3d[0]
```

```
Salida[78]:
```

```
matriz([[1, 2, 3], [4, 5, 6]])
```

Tanto los valores escalares como las matrices se pueden asignar a arr3d[0]:

```
En [79]: valores_antiguos = arr3d[0].copy()
```

```
En [80]: arr3d[0] = 42
```

```
En [81]: arr3d
```

```
Salida[81]:
```

```
matriz([[[42, 42, 42], [42, 42, 42]],
[[ 7, 8, 9],
[10, 11, 12]])
```

```
En [82]: arr3d[0] = valores_antiguos
```

```

En [83]: arr3d
Salida[83]:
matriz([[[[ 1, 2, 3], [ 4, 5,
           6]],
          [[ 7, 8, 9],
           [10, 11, 12]]]])

```

De manera similar, `arr3d[1, 0]` le brinda todos los valores cuyos índices comienzan con (1, 0), formando una matriz unidimensional:

```

En [84]: arr3d[1, 0]
Salida[84]: matriz([7, 8, 9])

```

Esta expresión es la misma que si hubiéramos indexado en dos pasos:

```

En [85]: x = arr3d[1]

En [86]: x
Salida[86]:
matriz([[ 7, 8, 9], [10, 11,
                  12]])

En [87]: x[0]
Salida[87]: matriz([7, 8, 9])

```

Tenga en cuenta que en todos estos casos en los que se han seleccionado subsecciones de la matriz, las matrices devueltas son vistas.

Indexación con cortes

Al igual que los objetos unidimensionales, como las listas de Python, los `ndarrays` se pueden dividir con la sintaxis familiar:

```

En [88]: arr
Salida[88]: matriz([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])

En [89]: arr[1:6]
Salida[89]: matriz([ 1, 2, 3, 4, 64])

```

Considere la matriz bidimensional de antes, `arr2d`. Cortar esta matriz es un poco diferente:

```

En [90]: arr2d
Salida[90]:
matriz([[[1, 2, 3], [4,
                  5, 6], [7, 8,
                  9]])

En [91]: arr2d[:2]
Salida[91]:
matriz([[[1, 2, 3], [4,
                  5, 6]])

```

Como 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 ser útil leer la expresión `arr2d[:2]` como "seleccione las dos primeras filas de `arr2d`".

Puede pasar varias porciones al igual que puede pasar múltiples índices:

```
En [92]: arr2d[:2, 1:]
Salida[92]:
matriz([[2, 3], [5,
6]])
```

Al cortar de esta manera, siempre obtiene vistas de matriz del mismo número de dimensiones. Al mezclar índices enteros y sectores, se obtienen sectores de menor dimensión.

Por ejemplo, puedo seleccionar la segunda fila pero solo las dos primeras columnas así:

```
En [93]: arr2d[1, :2]
Salida[93]: matriz([4, 5])
```

De manera similar, puedo seleccionar la tercera columna pero solo las dos primeras filas así:

```
En [94]: arr2d[:2, 2]
Salida[94]: matriz([3, 6])
```

Consulte la [Figura 4-2](#) para ver una ilustración. Tenga en cuenta que los dos puntos en sí mismos significan tomar todo el eje, por lo que puede dividir solo los ejes de dimensiones superiores haciendo:

```
En [95]: arr2d[:, :1]
Salida[95]:
matriz([[1],
[4],
[7]])
```

Por supuesto, la asignación a una expresión de sector asigna a toda la selección:

```
En [96]: arr2d[:2, 1:] = 0
```

```
En [97]: arr2d
Salida[97]:
matriz([[1, 0, 0], [4,
0, 0], [7, 8,
9]])
```

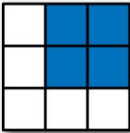
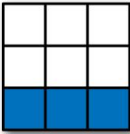
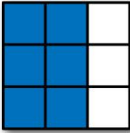
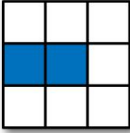
	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code>	<code>(3,)</code>
	<code>arr[2, :]</code>	<code>(3,)</code>
	<code>arr[2:, :]</code>	<code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code>	<code>(2,)</code>
	<code>arr[1:2, :2]</code>	<code>(1, 2)</code>

Figura 4-2. Corte de matriz bidimensional

Indexación booleana

Consideremos un ejemplo en el que tenemos algunos datos en una matriz y una matriz de nombres con duplicados. Voy a usar aquí la función `randn` en `numpy.random` para generar algunos datos aleatorios normalmente distribuidos:

```
En [98]: nombres = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
En [99]: datos = np.random.randn(7, 4)
```

```
En [100]: nombres
```

```
Salida[100]:
```

```
matriz(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
Entrada [101]:
```

```
salida de
```

```
datos [101]: matriz ([[ 0.0929, 0.2817, 0.769 , 1.2464],
 [ 1.0072, -1.2962, 0.275 , 0.2289], [ 1.3529,
 0.8864, -2.0016, -0.3718], [ 1.669 , -0.4386,
 -0.5397, 0.477 ], [ 3.2489, -1.0212, -0.5771,
 0.1241],
```

```
[ 0,3026, 0,5238, 0,0009, 1,3438], [-0,7135,
-0,8312, -2,3702, -1,8608]])
```

Supongamos que cada nombre corresponde a una fila en la matriz de datos y queremos seleccionar todas las filas con el nombre correspondiente 'Bob'. Al igual que las operaciones aritméticas, las comparaciones (como ==) con matrices también se vectorizan. Por lo tanto, comparar nombres con la cadena 'Bob' produce una matriz booleana:

```
En [102]: nombres == 'Bob'
Salida[102]: matriz([ Verdadero, Falso, Falso, Verdadero, Falso, Falso, Falso], dtype=bool)
```

Esta matriz booleana se puede pasar al indexar la matriz:

```
En [103]: datos[nombres == 'Bob']
Salida[103]:
matriz([[ 0.0929, 0.2817, 0.769 [ 1.669 , , 1.2464],
-0.4386, -0.5397, 0.477 ]])
```

La matriz booleana debe tener la misma longitud que el eje de la matriz que está indexando. Incluso puede mezclar y combinar matrices booleanas con segmentos o enteros (o secuencias de enteros; más sobre esto más adelante).



La selección booleana no fallará si la matriz booleana no tiene la longitud correcta, por lo que recomiendo tener cuidado al usar esta función.

En estos ejemplos, selecciono de las filas donde nombres == 'Bob' e indexo las columnas también:

```
En [104]: datos[nombres == 'Bob', 2:]
Salida[104]:
matriz([[ 0.769 , 1.2464],
[-0.5397, 0.477 ]])
```

```
En [105]: datos[nombres == 'Bob', 3]
Salida[105]: matriz([ 1.2464, 0.477 ])
```

Para seleccionar todo menos 'Bob', puede usar != o negar la condición usando ~:

```
En [106]: nombres != 'Bob'
Salida[106]: matriz([Falso, Verdadero, Verdadero, Falso, Verdadero, Verdadero, Verdadero], dtype=bool)
```

```
En [107]: datos[~(nombres == 'Bob')]
Salida[107]:
matriz([[ 1.0072, -1.2962, 0.275 , 0.2289], [ 1.3529,
0.8864, -2.0016, -0.3718], [ 3.2489 , -1.0212 ,
-0.5771, 0.1241], [ 0.3026, 0.5238 , 0.0009,
1.3438], [-0.7135, -0.8312, -2.3702, -1.8608]])
```

El operador ~ puede ser útil cuando desea invertir una condición general:

```
En [108]: cond = nombres == 'Bob'

En [109]: datos[~cond]
Salida[109]:
matriz([[ 1.0072, -1.2962, 0.275 , 0.2289], [ 1.3529,
0.8864, -2.0016, -0.3718], [ 3.2489 , -1.0212 ,
-0.5771, 0.1241], [ 0.3026, 0.5238 , 0.0009,
1.3438], [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Seleccionando dos de los tres nombres para combinar múltiples condiciones booleanas, use operadores aritméticos booleanos como & (y) y | (o):

```
En [110]: máscara = (nombres == 'Bob') | (nombres == 'Voluntad')

En [111]: máscara
Salida[111]: matriz([ Verdadero, Falso, Verdadero, Verdadero, Verdadero, Falso, Falso], dtype=bool)

En [112]: datos[máscara]
Salida[112]:
matriz([[ 0.0929, 0.2817, 0.769 , 1.2464], [ 1.3529,
0.8864, -2.0016, -0.3718], [ 1.669 , -0.4386,
-0.5397, 0.477 ], [ 3.2489, - 1.0212 , -0.5771 ,
0.1241]])
```

La selección de datos de una matriz mediante la indexación booleana siempre crea una copia de los datos, incluso si la matriz devuelta no ha cambiado.



Las palabras clave de Python y y o no funcionan con matrices booleanas. Utilice & (y) y | (o) en su lugar.

Establecer valores con matrices booleanas funciona con sentido común. Para establecer todos los valores negativos en los datos en 0, solo necesitamos hacer:

```
En [113]: datos[datos < 0] = 0

Entrada [114]:
salida de
datos [114]: matriz ([[ 0.0929, 0.2817, 0.769 , 1.2464],
[ 1.0072, 0. , 0.2289], [ 1.3529, 0.8864, 0.0. ],
[ 1.669 0.0. 0.477 ], [ 3.2489, 0. , 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],[ 0. ]])
, 0.
, 0. , 0. , 0.
```

Establecer filas o columnas completas usando una matriz booleana unidimensional también es fácil:

```
En [115]: datos[nombres != 'Joe'] = 7
```

Entrada [116]:

salida de

```
datos [116]: matriz([[ 7., [ 1.0072, 0., 0.2289], [ 7.
7., 7., ], [ 7., [ 7., [ 0.3026, 0.5238, 0.0009,
1.3438], [ 0. ]]) , 7., ,
, 7., , 7., , 7.
, 7., , 7., , 7.
, 0., , 0., , 0.
```

Como veremos más adelante, este tipo de operaciones sobre datos bidimensionales son convenientes para hacer con pandas.

Fancy Indexing Fancy

indexing es un término adoptado por NumPy para describir la indexación utilizando matrices de enteros.

Supongamos que tuviéramos una matriz de 8×4 :

```
En [117]: arr = np.vacío((8, 4))
```

```
En [118]: for i in range(8): arr[i] = i
.....:
```

Entrada [119]:

matriz

```
Salida[119]: matriz([[ 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 orden particular, simplemente puede pasar una lista o ndarray de enteros especificando el orden deseado:

```
En [120]: arr[[4, 3, 0, 6]]
```

Salida[120]:

```
matriz([[ 4., 4., 4., 4.], [ 3., 3., 3.,
3.], [ 0., 0., 0., 0.], [ 6., 6.,
6., 6.]])
```

¡Ojalá este código haya hecho lo que esperabas! El uso de índices negativos selecciona filas desde el final:

```
En [121]: arr[-3, -5, -7]]
```

Fuera[121]:

```
matriz ([[ 5., 5., 5., 5.], [ 3., 3., 3.,
3.], [ 1., 1., 1., 1.]])
```

Pasar varias matrices de índice hace algo ligeramente diferente; selecciona una matriz unidimensional de elementos correspondientes a cada tupla de índices:

```
En [122]: arr = np.arange(32).reshape((8, 4))
```

```
En [123]: arr
```

```
Salida[123]:
```

```
matriz([[ 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]])
```

```
En [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
Salida[124]: matriz([ 4, 23, 29, 10])
```

Veremos el método de remodelación con más detalle en el [Apéndice A](#).

Aquí se seleccionaron los elementos (1, 0), (5, 3), (7, 1) y (2, 2). Independientemente de cuántas dimensiones tenga la matriz (aquí, solo 2), el resultado de la indexación elegante siempre es unidimensional.

El comportamiento de la indexación elegante en este caso es un poco diferente de lo que algunos usuarios podrían haber esperado (incluido yo mismo), que es la región rectangular formada al seleccionar un subconjunto de filas y columnas de la matriz. Aquí hay una forma de conseguirlo:

```
En [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

```
Salida[125]:
```

```
matriz([[ 4, 7, 5, 6], [20, 23,
21, 22], [28, 31, 29,
30], [ 8, 11, 9, 10]])
```

Tenga en cuenta que la indexación sofisticada, a diferencia del corte, siempre copia los datos en una nueva matriz.

Transposición de matrices y ejes de intercambio La

transposición es una forma especial de remodelación que, de manera similar, devuelve una vista de los datos subyacentes sin copiar nada. Las matrices tienen el método de transposición y también el atributo `T` especial :

```
En [126]: arr = np.arange(15).reshape((3, 5))
```

```
En [127]: arr
```

```
Salida[127]:
matriz([[ 0, 1, 2, 3, 4], [ 5, 6, 7, 8,
9], [10, 11, 12, 13, 14]])
```

```
Entrada [128]:
arreglo.T
Salida[128]: arreglo ([[ 0,
5, 10], [ 1, 6,
11], [ 2, 7, 12],
[ 3, 8, 13], [ 4 ,
9, 14]])
```

Al realizar cálculos matriciales, puede hacerlo muy a menudo, por ejemplo, al calcular el producto de la matriz interna mediante `np.dot`:

```
En [129]: arr = np.random.randn(6, 3)
```

```
Entrada [130]:
matriz
Salida[130]: matriz ([[ -0.8608, 0.5601,
-1.2659], [ 0.1198, -1.0635,
0.3329], [-2.3594, -0.1995,
-1.542 ], [-0.9707, -1.307 , 0.2863],
[ 0.378 , -0.7539, 0.3313], [ 1.3497,
0.0699, 0.2467]])
```

```
En [131]: np.dot(arr.T, arr)
Salida[131]:
matriz([[ 9.2291, 0.9394, 4.948 ], [ 0.9394,
3.7662, -1.3622], [ 4.948 , -1.3622,
4.3437]])
```

Para matrices de mayor dimensión, la transposición aceptará una tupla de números de eje para permutar los ejes (para una mayor flexión de la mente):

```
En [132]: arr = np.arange(16).reshape((2, 2, 4))
```

```
En [133]: arr
Salida[133]:
matriz([[[[ 0, 1, 2, 3], [ 4, 5, 6,
7]],
[[ 8, 9, 10, 11],
[12, 13, 14, 15]]]])
```

```
En [134]: arr.transpose((1, 0, 2))
Salida[134]:
matriz([[[[ 0, 1, 2, 3], [ 8, 9, 10,
11]],
[[ 4, 5, 6, 7],
[12, 13, 14, 15]]]])
```

Aquí, los ejes se han reordenado con el segundo eje primero, el primer eje segundo, y el último eje sin cambios.

La transposición simple con `.T` es un caso especial de intercambio de ejes. `ndarray` tiene el método `swapaxes`, que toma un par de números de eje y cambia los ejes indicados a la parte trasera. rango de los datos:

```
En [135]: arr
Fuera[135]:
matriz ( [[[ 0, 1, 2, 3],
            [ 4, 5, 6, 7]],
          [[ 8, 9, 10, 11],
            [12, 13, 14, 15]])
```

```
En [136]: arr.swapaxes(1, 2)
Fuera[136]:
matriz ( [[[ 0, 4],
            [ 1, 5],
            [ 2, 6],
            [ 3, 7]],
          [[ 8, 12],
            [ 9, 13],
            [10, 14],
            [11, 15]])
```

`swapaxes` devuelve de manera similar una vista de los datos sin hacer una copia.

4.2 Funciones universales: matriz rápida de elementos Funciones

Una función universal, o `ufunc`, es una función que realiza operaciones con elementos sobre datos en `ndarrays`. Puede pensar en ellos como envoltorios vectorizados rápidos para simples funciones que toman uno o más valores escalares y producen uno o más resultados escalares.

Muchos `ufuncs` son simples transformaciones de elementos, como `sqrt` o `exp`:

```
En [137]: arr = np.arange(10)
```

```
En [138]: arr
Salida[138]: matriz([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
En [139]: np.sqrt(arr)
Fuera[139]:
matriz ([ 0. 1. , 1.4142, 1.7321, 2. 2.6458, 2.8284, 3. ]) , 2.2361, 2.4495,
```

```
En [140]: np.exp(arr)
Fuera[140]:
formación([ 1. , 2.7183, 7.3891, 20.0855, 54.5982,
            148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Estos se denominan ufuncs unarios. Otros, como agregar o máximo, toman dos matrices (por lo tanto, ufuncs binarios) y devuelve una sola matriz como resultado:

```
En [141]: x = np.random.randn(8)

En [142]: y = np.random.randn(8)

En [143]: x
Fuera[143]:
matriz ([-0.0119, 1.0048, 1.3272, -0.9193, -1.5491, 0.0222, 0.7584,
        -0.6605])

En [144]: y
Fuera[144]:
matriz ([ 0.8626, -0.01      ,  0.05      , 0.6702, 0.853      , -0.9559, -0.0235,
        -2.3042])

En [145]: np.maximum(x, y)
Fuera[145]:
matriz ([ 0.8626, 1.0048, 1.3272, 0.6702, 0.853 -0.6605])      , 0.0222, 0.7584,
```

Aquí, `numpy.maximum` calculó el máximo por elemento de los elementos en `x` y `y`.

Si bien no es común, un ufunc puede devolver varias matrices. `modf` es un ejemplo, un vec versión torizada del `divmod` integrado de Python ; devuelve el fraccionario y el integral partes de una matriz de punto flotante:

```
En [146]: arr = np.random.randn(7) * 5

En [147]: arr
Salida[147]: matriz([-3.2623, -6.0915, -6.663      , 5.3731, 3.6182, 3.45      , 5.0077])

En [148]: resto, parte_completa = np.modf(arr)

En [149]: resto
Salida[149]: matriz([-0.2623, -0.0915, -0.663      , 0.3731, 0.6182, 0.45      , 0.0077])

En [150]: parte_completa
Salida[150]: matriz([-3., -6., -6., 5., 3., 3., 5.])
```

Ufuncs acepta un argumento de salida opcional que les permite operar en el lugar en arreglos:

```
En [151]: arr
Salida[151]: matriz([-3.2623, -6.0915, -6.663      , 5.3731, 3.6182, 3.45      , 5.0077])

En [152]: np.sqrt(arr)
Salida[152]: matriz([      yaya,      yaya,      nan, 2.318      , 1.9022, 1.8574, 2.2378])

En [153]: np.sqrt(arr, arr)
```

```
Salida[153]: matriz([      yaya,      yaya,      nan, 2.318      , 1.9022, 1.8574, 2.2378])

En [154]: arr
Salida[154]: matriz([      yaya,      yaya,      nan, 2.318      , 1.9022, 1.8574, 2.2378])
```

Consulte las Tablas 4-3 y 4-4 para obtener una lista de ufuncs disponibles.

Tabla 4-3. ufunciones unarias

Función	Descripción
absdominales, fabulosos	Calcule el valor absoluto por elementos para valores enteros, de punto flotante o complejos
sqrt	Calcule la raíz cuadrada de cada elemento (equivalente a arr ** 0.5)
cuadrado	Calcule el cuadrado de cada elemento (equivalente a arr ** 2)
Exp	Calcule el exponente ex de cada elemento
registro, registro10, registro2, registro1p	Logaritmo natural (base e), logaritmo en base 10, logaritmo en base 2 y log(1 + x), respectivamente
fimar	Calcule el signo de cada elemento: 1 (positivo), 0 (cero) o -1 (negativo)
fortificar techo	Calcule el techo de cada elemento (es decir, el entero más pequeño mayor o igual que ese número)
piso	Calcule el piso de cada elemento (es decir, el entero más grande menor o igual a cada elemento)
imprimir	Redondear elementos al entero más cercano, conservando el dtype
modelo	Devolver partes fraccionarias e integrales de una matriz como una matriz separada
isnan	Devuelve una matriz booleana que indica si cada valor es NaN (no es un número)
isfinito, isinf	Devuelve una matriz booleana que indica si cada elemento es finito (no inf, no NaN) o infinito, respectivamente
cos, cosh, pecado, sinh, bronceado, tanh	Funciones trigonométricas regulares e hiperbólicas
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Funciones trigonométricas inversas
lógico_no	Calcule el valor de verdad de not x elemento-sabio (equivalente a ~arr).

Tabla 4-4. Funciones universales binarias

Función	Descripción
agregar	Agregar elementos correspondientes en matrices
sustraer	Restar elementos en la segunda matriz de la primera matriz
multiplicar	Multiplicar elementos de matriz
dividir, piso_dividir	Dividir o dividir piso (truncando el resto)
fuerza	Eleve los elementos de la primera matriz a las potencias indicadas en la segunda matriz
máximo, fmax	Máximo por elementos; fmax ignora NaN
mínimo, fmin	Mínimo elemental; fmin ignora NaN
modificación	Módulo elemental (resto de la división)
copia firmante	Copie el signo de los valores en el segundo argumento a los valores en el primer argumento

Función	Descripción
mayor, mayor_igual, menor, menor_igual, igual, no_igual	Realice una comparación por elementos, lo que genera una matriz booleana (equivalente a los operadores infijos >, >=, <, <=, ==, !=)
lógico_y, lógico_o, lógico_xor	Calcule el valor de verdad de los elementos de la operación lógica (equivalente a los operadores infijos &, , ^)

4.3 Programación orientada a matrices con matrices

El uso de matrices NumPy le permite expresar muchos tipos de tareas de procesamiento de datos como expresiones de matriz concisas que de otro modo podrían requerir bucles de escritura. Esta práctica de reemplazar bucles explícitos con expresiones de matriz se conoce comúnmente como vectorización. En general, las operaciones de matrices vectorizadas suelen ser de uno o dos (o más) órdenes de magnitud más rápido que sus equivalentes puros de Python, con el mayor impacto en cualquier tipo de cálculo numérico. Más adelante, en el Apéndice A, explico la radiodifusión, un poderoso método para vectorizar cálculos.

Como ejemplo simple, supongamos que deseamos evaluar la función $\sqrt{x^2 + y^2}$ a través de una cuadrícula regular de valores. La función `np.meshgrid` toma dos matrices 1D y produce dos matrices 2D correspondientes a todos los pares de (x, y) en las dos matrices:

```
En [155]: puntos = np.arange(-5, 5, 0.01) # 1000 puntos igualmente espaciados

En [156]: xs, ys = np.meshgrid(puntos, puntos)

En [157]: sí
Fuera[157]:
matriz([[ -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 usarías escribir con dos puntos:

```
En [158]: z = np.sqrt(xs ** 2 + ys ** 2)

En [159]: z
Fuera[159]:
matriz([[ 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]])
```

Como vista previa del **Capítulo 9**, uso matplotlib para crear visualizaciones de esta matriz bidimensional:

En [160]: `importar matplotlib.pyplot como plt`

En [161]: `plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()`

Salida[161]: `<matplotlib.colorbar.Colorbar en 0x7f715e3fa630>`

En [162]: `plt.title(" Gráfico de imagen de  $\sqrt{x^2 + y^2}$  para una cuadrícula de valores")`

Salida[162]: `<matplotlib.text.Text en 0x7f715d2de748>`

Consulte la **Figura 4-3**. Aquí utilicé la función matplotlib imshow para crear un gráfico de imagen a partir de una matriz bidimensional de valores de función.

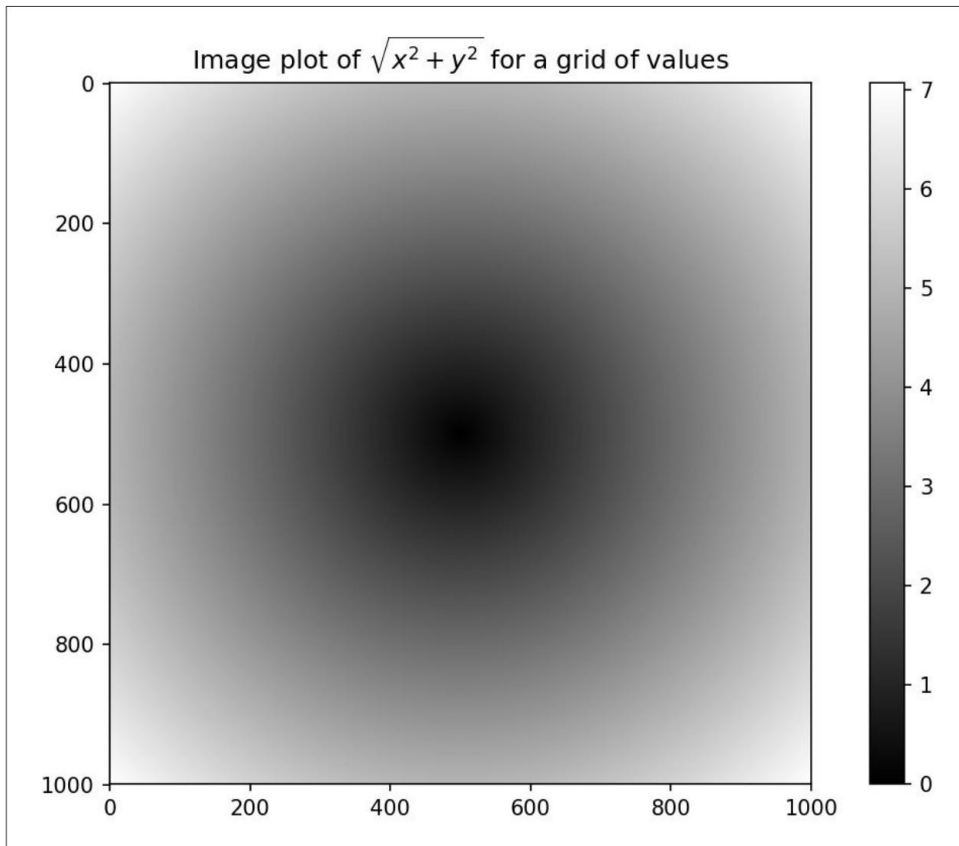


Figura 4-3. Parcela de función evaluada en cuadrícula

Expresión de la lógica condicional como operaciones de matriz La función

`numpy.where` es una versión vectorizada de la expresión ternaria `x if condition else y`. Supongamos que tuviéramos una matriz booleana y dos matrices de valores:

```
En [165]: xarr = np.matriz([1.1, 1.2, 1.3, 1.4, 1.5])
```

```
En [166]: yarr = np.matriz([2.1, 2.2, 2.3, 2.4, 2.5])
```

```
En [167]: cond = np.array([Verdadero, Falso, Verdadero, Verdadero, Falso])
```

Supongamos que quisiéramos tomar un valor de `xarr` siempre que el valor correspondiente en `cond` sea `True` y, de lo contrario, tomar el valor de `yarr`. Una lista de comprensión haciendo esto podría verse así:

```
En [168]: resultado = [(x si c sino y)
.....:                  para x, y, c en zip (xarr, yarr, cond)]
```

```
Entrada [169]:
```

```
resultado Salida[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

Esto tiene múltiples problemas. Primero, no será muy rápido para matrices grandes (porque todo el trabajo se realiza en código Python interpretado). En segundo lugar, no funcionará con arreglos multidimensionales. Con `np.where` puedes escribir esto de manera muy concisa:

```
En [170]: resultado = np.where(cond, xarr, yarr)
```

```
Entrada [171]:
```

```
resultado Salida[171]: array([ 1.1, 2.2, 1.3, 1.4, 2.5])
```

El segundo y tercer argumento de `np.where` no necesitan ser matrices; uno o ambos pueden ser escalares. Un uso típico de `where` en el análisis de datos es producir una nueva matriz de valores basada en otra matriz. Suponga que tiene una matriz de datos generados aleatoriamente y desea reemplazar todos los valores positivos con 2 y todos los valores negativos con -2. Esto es muy fácil de hacer con `np.where`:

```
En [172]: arr = np.random.randn(4, 4)
```

```
En [173]: Arr
```

```
Out [173]:
```

```
Array ([[ -0.5031, -0.6223, -0.9212, -0.7262], [ 0.2229,
         0.0513, -1.1577, 0.8167], [ 0.4336, 1.0107,
         1.8249, -0.9975], [ 0.8506, -0.1316, 0.9124,
         0.1882]])
```

```
En [174]: arr > 0
```

```
Salida[174]:
```

```
array([[Falso, Falso, Falso, Falso],
       [Verdadero, Verdadero, Falso, Verdadero],
       [Verdadero, Verdadero, Verdadero, Falso],
       [Verdadero, Falso, Verdadero, Verdadero]], dtype=bool)
```

```
En [175]: np.where(arr > 0, 2, -2)
```

```
Salida[175]:
```

```
matriz([[ -2, -2, -2, -2], [ 2, 2,
        -2, 2],
```

```
[ 2, 2, 2, -2],
[ 2, -2, 2, 2]])
```

Puede combinar escalares y matrices al usar `np.where`. Por ejemplo, puedo reemplazar todos los valores positivos en `arr` con la constante 2 así:

```
En [176]: np.where(arr > 0, 2, arr) # establecer solo valores positivos en 2
Fuera[176]:
matriz ([[ -0.5031, -0.6223, -0.9212, -0.7262],
[ 2. ],      , 2. ,      , -1.1577, 2. [ 2.
2. 2. , -0.9975],      ,
[ 2. , -0.1316, 2. 2. ]])
```

Las matrices pasadas a `np.where` pueden ser más que simples matrices o escalares del mismo tamaño.

Métodos Matemáticos y Estadísticos

Un conjunto de funciones matemáticas que calculan estadísticas sobre un arreglo completo o sobre los datos a lo largo de un eje son accesibles como métodos de la clase de matriz. Puede usar agregados (a menudo llamadas reducciones) como `suma`, `media` y `std` (desviación estándar) ya sea por llamando al método de instancia de matriz o usando la función NumPy de nivel superior.

Aquí genero algunos datos aleatorios distribuidos normalmente y calculo algunos agregados Estadísticas:

```
En [177]: arr = np.random.randn(5, 4)
```

```
En [178]: arr
```

```
Fuera[178]:
```

```
matriz ([[ 2.1695, -0.1149, 2.0037, 0.0296],
[ 0.7953, 0.1181, -0.7485, 0.585 ],
[ 0.1527, -1.5657, -0.5625, -0.0327],
[-0.929 , -0.4826, -0.0363, 1.0954],
[ 0.9809, -0.5895, 1.5817, -0.5287]])
```

```
En [179]: arr.media()
```

```
Salida[179]: 0.19607051119998253
```

```
En [180]: np.mean(arr)
```

```
Salida[180]: 0.19607051119998253
```

```
En [181]: arr.sum()
```

```
Salida[181]: 3.9214102239996507
```

Funciones como `mean` y `sum` toman un argumento de eje opcional que calcula la estadística sobre el eje dado, lo que da como resultado una matriz con una dimensión menos:

```
En [182]: arr.media(eje=1)
```

```
Salida[182]: matriz([ 1.022 , 0.1875, -0.502 , -0.0881, 0.3611])
```

```
En [183]: arr.sum(eje=0)
```

```
Salida[183]: matriz([ 3.1693, -2.6345, 2.2381, 1.1486])
```

Aquí, `arr.mean(1)` significa "calcular la media de las columnas", donde `arr.sum(0)` significa "calcular la suma de las filas".

Otros métodos como `cumsum` y `cumprod` no se agregan, sino que producen una matriz de resultados intermedios:

```
En [184]: matriz = np.matriz ([0, 1, 2, 3, 4, 5, 6, 7])
```

```
En [185]: arr.cumsum()  
Salida[185]: matriz([ 0, 1, 3, 6, 10, 15, 21, 28])
```

En matrices multidimensionales, las funciones de acumulación como `cumsum` devuelven una matriz del mismo tamaño, pero con los agregados parciales calculados a lo largo del eje indicado de acuerdo con cada segmento dimensional inferior:

```
En [186]: matriz = np.matriz ([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

```
En [187]: arr  
Salida[187]:  
matriz([[0, 1, 2], [3, 4,  
                    5], [6, 7, 8]])
```

```
En [188]: arr.cumsum(eje=0)  
Salida[188]:  
matriz([[ 0, 1, 2], [ 3, 5, 7],  
        [ 9, 12, 15]])
```

```
En [189]: arr.cumprod(eje=1)  
Salida[189]:  
matriz([[ 0, 0], 0, [ 3, 12, 60], [ 6,  
                    42, 336]])
```

Consulte la **Tabla 4-5** para obtener una lista completa. Veremos muchos ejemplos de estos métodos en acción en capítulos posteriores.

Tabla 4-5. Métodos estadísticos básicos de matriz

Método	Descripción
suma	Suma de todos los elementos de la matriz oa lo largo de un eje; las matrices de longitud cero tienen suma 0
significar	Significado aritmético; las matrices de longitud cero tienen una media de NaN
estándar, variable	Desviación estándar y varianza, respectivamente, con ajuste opcional de grados de libertad (denominador predeterminado n)
min, max	mínimo y máximo
argmin, argmax	Índices de elementos mínimo y máximo, respectivamente
cumsum	Suma acumulada de elementos a partir de 0
Cumprod	Producto acumulativo de elementos a partir de 1

Métodos para matrices booleanas Los valores

booleanos se convierten en 1 (Verdadero) y 0 (Falso) en los métodos anteriores. Por lo tanto, la suma se usa a menudo como un medio para contar valores verdaderos en una matriz booleana:

```
En [190]: arr = np.random.randn(100)
```

```
In [191]: (arr > 0).sum() # Número de valores positivos Out[191]: 42
```

Hay dos métodos adicionales, cualquiera y todos, útiles especialmente para arreglos booleanos. any comprueba si uno o más valores de una matriz son verdaderos, mientras que all comprueba si todos los valores son verdaderos:

```
En [192]: bools = np.array([Falso, Falso, Verdadero, Falso])
```

```
En [193]: bools.any()
```

```
Salida[193]: Verdadero
```

```
En [194]: bools.todo()
```

```
Salida[194]: Falso
```

Estos métodos también funcionan con matrices no booleanas, donde los elementos distintos de cero evalúan como Verdadero.

Clasificación

Al igual que el tipo de lista integrado de Python, las matrices NumPy se pueden ordenar en el lugar con el método de clasificación :

```
En [195]: arr = np.random.randn(6)
```

```
Entrada [196]:
```

```
matriz Salida[196]: matriz([ 0.6095, -0.4938, 1.24      , -0.1357, 1.43      , -0.8469])
```

```
En [197]: arr.sort()
```

```
Entrada [198]:
```

```
matriz Salida[198]: matriz([-0.8469, -0.4938, -0.1357, 0.6095, 1.24      , 1.43      ])
```

Puede ordenar cada sección unidimensional de valores en una matriz multidimensional en su lugar a lo largo de un eje pasando el número de eje para ordenar:

```
En [199]: arr = np.random.randn(5, 3)
```

```
Entrada [200]:
```

```
matriz
```

```
Salida[200]: matriz([[ 0.6033, 1.2636,
                    -0.2555], [-0.4457, 0.4684,
                    -0.9616], [-1.8245, 0.6254, 1.0229],
                    [ 1.1074, 0.0909, -0.3501 ] ,
                    [ 0.218 , -0.8948, -1.7415]])
```

En [201]: `arr.sort(1)`

Entrada [202]:
matriz

Salida[202]: matriz ([[-0.2555, 0.6033,
1.2636], [-0.9616, -0.4457,
0.4684], [-1.8245, 0.6254, 1.0229],
[-0.3501, 0.0909, 1.1074] ,
[-1.7415, -0.8948, 0.218]])

El método de nivel superior `np.sort` devuelve una copia ordenada de una matriz en lugar de modificar la matriz en el lugar. Una forma rápida y sucia de calcular los cuantiles de una matriz es ordenarla y seleccionar el valor en un rango particular:

En [203]: `large_arr = np.random.randn(1000)`

En [204]: `large_arr.sort()`

In [205]: `large_arr[int(0.05 * len(large_arr))]` # 5% cuantil Out[205]:
-1.5311513550102103

Para obtener más detalles sobre el uso de los métodos de clasificación de NumPy y técnicas más avanzadas como las clasificaciones indirectas, consulte el [Apéndice A](#). En pandas también se pueden encontrar otros tipos de manipulación de datos relacionados con la clasificación (p. ej., clasificar una tabla de datos por una o más columnas).

Unique y Other Set Logic NumPy tiene

algunas operaciones básicas de conjuntos para ndarrays unidimensionales. Uno de uso común es `np.unique`, que devuelve los valores únicos ordenados en una matriz:

En [206]: `nombres = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])`

En [207]: `np.unique(nombres)`

Salida[207]:
matriz(['Bob', 'Joe', 'Will'],
dtype='<U4')

En [208]: `enteros = np.matriz([3, 3, 3, 2, 2, 1, 1, 4, 4])`

En [209]: `np.unique(enteros)`

Salida[209]: matriz([1, 2, 3, 4])

Contraste `np.unique` con la alternativa pura de Python:

En [210]: `ordenados = conjunto(nombres)`
Fuera[210]: ['Bob', 'Joe', 'Will']

Otra función, `np.in1d`, prueba la pertenencia de los valores de una matriz a otra y devuelve una matriz booleana:

```
En [211]: valores = np.array([6, 0, 0, 3, 2, 5, 6])

En [212]: np.in1d(valores, [2, 3, 6])
Salida[212]: matriz([ Verdadero, Falso, Falso, Verdadero, Verdadero, Falso, Verdadero], dtype=bool)
```

Consulte la [Tabla 4-6](#) para obtener una lista de las funciones establecidas en NumPy.

Tabla 4-6. Operaciones de conjuntos de matrices

Método	Descripción
en x	Calcule los elementos únicos ordenados
unique(x) intersect1d(x, y)	Calcule los elementos comunes ordenados en x e y
union1d(x, y)	Calcular la unión ordenada de elementos
en1d(x, y)	Calcule una matriz booleana que indique si cada elemento de x está contenido en y
setdiff1d(x, y)	Diferencia de conjuntos, elementos en x que no están en y
conjuntoxor1d(x, y)	Establecer diferencias simétricas; elementos que están en cualquiera de las matrices, pero no en ambas

4.4 Entrada y salida de archivos con matrices

NumPy puede guardar y cargar datos desde y hacia el disco, ya sea en formato de texto o binario. En esta sección solo analizo el formato binario incorporado de NumPy, ya que la mayoría de los usuarios preferirán pandas y otras herramientas para cargar texto o datos tabulares (consulte el [Capítulo 6](#) para obtener más información).

np.save y np.load son las dos funciones de caballo de batalla para guardar y cargar datos de matriz en el disco de manera eficiente. Las matrices se guardan de forma predeterminada en un formato binario sin comprimir con la extensión de archivo .npy:

```
En [213]: arr = np.arange(10)

En [214]: np.save('some_array', arr)
```

Si la ruta del archivo aún no termina en .npy, se agregará la extensión. La matriz en el disco se puede cargar con np.load:

```
En [215]: np.load('some_array.npy')
Salida[215]: matriz([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Guarde varias matrices en un archivo sin comprimir usando np.savez y pasando las matrices como argumentos de palabras clave:

```
En [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

Al cargar un archivo .npz, obtiene un objeto similar a un dictado que carga las matrices individuales de forma perezosa:

```
En [217]: arch = np.load('array_archive.npz')

En [218]: arco['b']
Salida[218]: matriz([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Si sus datos se comprimen bien, es posible que desee utilizar `numpy.savez_compressed` en su lugar:

```
En [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

4.5 Álgebra lineal

El álgebra lineal, como la multiplicación de matrices, las descomposiciones, los determinantes y otras matemáticas de matrices cuadradas, es una parte importante de cualquier biblioteca de matrices. A diferencia de algunos lenguajes como MATLAB, multiplicar dos matrices bidimensionales con `*` es un producto de elementos en lugar de un producto de punto de matriz. Por lo tanto, hay un punto de función, tanto un método de matriz como una función en el espacio de nombres `numpy`, para la multiplicación de matrices:

```
En [223]: x = np.matriz([[1., 2., 3.], [4., 5., 6.]])
```

```
En [224]: y = np.matriz([[6., 23.], [-1, 7], [8, 9]])
```

Entrada

```
[225]: x
```

```
Salida[225]: array([[ 1., 2., 3.],
                  [ 4., 5., 6.]])
```

Entrada

```
[226]: y
```

```
Salida[226]: array([[ 6., 23.],
                  [-1., 7.],
                  [ 8., 9.]])
```

```
En [227]: x.punto(y)
```

```
Salida[227]: matriz([[ 28., 64.],
                    [ 67., 181.]])
```

`x.dot(y)` es equivalente a `np.dot(x, y)`:

```
En [228]: np.dot(x, y)
```

```
Salida[228]: matriz([[ 28., 64.],
                    [ 67., 181.]])
```

Un producto matricial entre una matriz bidimensional y una matriz unidimensional de tamaño adecuado da como resultado una matriz unidimensional:

```
En [229]: np.dot(x, np.ones(3))
```

```
Salida[229]: matriz([ 6., 15.] )
```

El símbolo `@` (a partir de Python 3.5) también funciona como un operador infijo que realiza la multiplicación de matrices:

```
En [230]: x @ np.ones(3)
```

```
Salida[230]: matriz([ 6., 15.] )
```

numpy.linalg tiene un conjunto estándar de descomposición de matrices y cosas como inversa y determinante. Estos se implementan bajo el capó a través de las mismas bibliotecas de álgebra lineal estándar de la industria utilizadas en otros lenguajes como MATLAB y R, como BLAS, LAPACK o posiblemente (dependiendo de su compilación NumPy) Intel MKL (Math Kernel Library) propietario:

```
En [231]: from numpy.linalg import inv, qr

En [232]: X = np.random.randn(5, 5)

En [233]: mat = XTdot(X)

En [234]: inv(tapete)
Salida[234]:
matriz([[ 933.1189, 871.8258, -1417.6902, -1460.4005, 1782.1391], [ 871.8258,
815.3929, -1325.9965, -1365.9242 , 1666.9347], [- 1417.6902, -1325.9965,
2158.4424, 2222.0191, -2711.6822] , [-1460.4005, -1365.9242, 2222.0191,
2289.0575, -2793.422 ], [ 1782.1391, 1666.9347, -2711.6822, -2793.422
, 3409.5128]])

En [235]: mat.dot(inv(mat))
Salida[235]:
matriz([[ 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.] ] )

En [236]: q, r = qr(mat)

Entrada [237]:
r
Salida[237]: array([[ -1.6914, 0.1757, 0.4075, -0.7838], ,
4.38 [ 0. , -2.6436, 0.1939, -3.072 [ 0. [ 0. -1.0702], ,
[ 0. , 0. -0.8138, 1.5414, 0.6155], , -2.6445,
, 0. , 0. -2.1669], 0. , 0.0002]])
, 0. , 0. ,
```

La expresión XTdot(X) calcula el producto escalar de X con su transpuesta XT

Consulte [la tabla 4-7](#) para obtener una lista de algunas de las funciones de álgebra lineal más utilizadas.

Tabla 4-7. Funciones numpy.linalg comúnmente utilizadas

Función descriptiva	
diagnostico	Devuelva los elementos diagonales (o fuera de la diagonal) de una matriz cuadrada como una matriz 1D, o convierta una matriz 1D en una matriz cuadrada con ceros fuera de la diagonal
punto	Traza de la multiplicación
de matrices	Calcular la suma de los elementos de la diagonal
det	Calcule el determinante de la matriz

Función descriptiva	
<code>eig</code>	Calcule los valores propios y los vectores propios de una matriz cuadrada
<code>inv</code>	Calcular la inversa de una matriz cuadrada
<code>Pinv</code>	Calcule la pseudo-inversa de Moore-Penrose de una matriz
<code>qr</code>	Calcular la descomposición QR
<code>svd</code>	Calcule la descomposición en valores singulares (SVD)
solve Resolver el sistema lineal $Ax = b$ para x , donde A es una matriz cuadrada	
lstsq Calcular la solución de mínimos cuadrados para $Ax = b$	

4.6 Generación de números pseudoaleatorios

El módulo `numpy.random` complementa el Python aleatorio incorporado con funciones para generar de manera eficiente matrices completas de valores de muestra de muchos tipos de distribuciones de probabilidad. Por ejemplo, puede obtener una matriz de muestras de 4×4 de la distribución normal estándar usando `normal`:

```
En [238]: muestras = np.random.normal(size=(4, 4))

In [239]: muestras
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796], [ 0.575 ,  0.4339,
        -0.7658, -1.237 ], [-0.5367,  1.8545, -0.92 ,
        -0.1082], [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

El módulo aleatorio incorporado de Python, por el contrario, solo muestra un valor a la vez. Como puede ver en este punto de referencia, `numpy.random` es mucho más rápido en un orden de magnitud para generar muestras muy grandes:

```
En [240]: from random import normalvariate

En [241]: N = 1000000

En [242]: %timeit muestras = [normalvariate(0, 1) for _ in range(N)]
ms per loop (media +- desv. estándar de 7 ejecuciones, 1 loop cada una)

En [243]: %timeit np.random.normal(size=N)
61.7 ms
+- 1.32 ms por bucle (media +- desviación estándar de 7 ejecuciones, 10 bucles cada una)
```

Decimos que son números pseudoaleatorios porque son generados por un algoritmo con comportamiento determinista basado en la semilla del generador de números aleatorios. Puede cambiar la semilla de generación de números aleatorios de NumPy usando `np.random.seed`:

```
En [244]: np.random.seed(1234)
```

Las funciones de generación de datos en `numpy.random` usan una semilla aleatoria global. Para evitar el estado global, puede usar `numpy.random.RandomState` para crear un generador de números aleatorios aislado de los demás:

```
En [245]: rng = np.random.RandomState(1234)

En [246]: rng.randn(10)
Salida[246]:
matriz([ 0.4714, -1.191      , 1.4327, -0.3127, -0.7206 , 0.8872 , 0.8596, -0.6365 , 0.0157,
        -2.2427])
```

Consulte la [Tabla 4-8](#) para obtener una lista parcial de las funciones disponibles en `numpy.random`. Daré algunos ejemplos de cómo aprovechar la capacidad de estas funciones para generar grandes matrices de muestras de una vez en la siguiente sección.

Tabla 4-8. Lista parcial de funciones `numpy.random`

Función	Descripción
semilla	Sembrar la permutación del generador de
números aleatorios	Devolver una permutación aleatoria de una secuencia, o devolver un rango permutado
barajar	Permutar aleatoriamente una secuencia en el lugar
rand	Extraer muestras de una distribución uniforme
al azar	Dibuja números enteros aleatorios de un rango bajo a alto dado
rancio	Extraiga muestras de una distribución normal con media 0 y desviación estándar 1 (interfaz similar a MATLAB)
binomio	Extraer muestras de una distribución binomial
normal	Extraiga muestras de una distribución normal (gaussiana)
beta	Extraer muestras de una distribución beta
chi cuadrado	Extraer muestras de una distribución de chi-cuadrado
gama	Extraer muestras de una distribución gamma
uniforme	Extraiga muestras de una distribución uniforme [0, 1)

4.7 Ejemplo: paseos aleatorios

La simulación de **paseos aleatorios** proporciona una aplicación ilustrativa de la utilización de operaciones de matriz. Consideremos primero un paseo aleatorio simple que comienza en 0 con pasos de 1 y -1 que ocurren con la misma probabilidad.

Aquí hay una forma pura de Python para implementar una sola caminata aleatoria con 1,000 pasos usando el módulo aleatorio incorporado :

```
En [247]: import aleatorio .....:
         posición = 0 .....: caminar =
         [posición] .....: pasos = 1000 .....:
         para i en rango (pasos):
         paso = 1 si random.randint(0, 1) else -1
         .....:         posición += paso
         .....:
```

```
..... caminar.añadir(posición)
.....
```

Consulte la [Figura 4-4](#) para ver un gráfico de ejemplo de los primeros 100 valores en uno de estos recorridos aleatorios:

```
En [249]: plt.plot(caminar[:100])
```

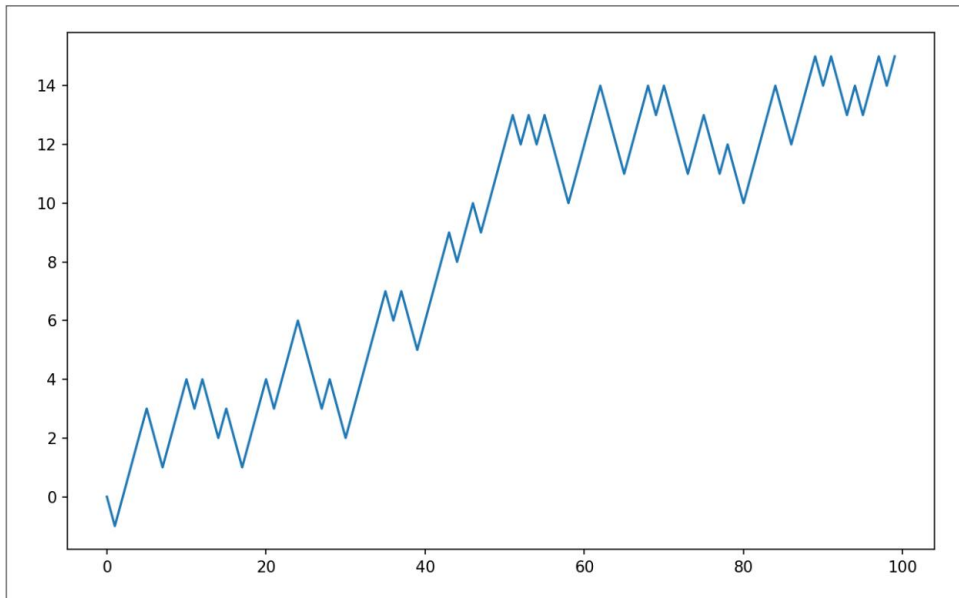


Figura 4-4. Un simple paseo aleatorio

Puede hacer la observación de que la caminata es simplemente la suma acumulativa de los pasos aleatorios y podría evaluarse como una expresión de matriz. Por lo tanto, uso el módulo `np.random` para dibujar 1,000 lanzamientos de monedas a la vez, los establezco en 1 y -1, y calculo la suma acumulada:

```
En [251]: npasos = 1000
```

```
En [252]: sorteos = np.random.randint(0, 2, tamaño=npasos)
```

```
En [253]: pasos = np.where(dibuja > 0, 1, -1)
```

```
En [254]: caminar = pasos.cumsum()
```

A partir de esto podemos comenzar a extraer estadísticas como el valor mínimo y máximo a lo largo de la trayectoria de la caminata:

```
En [255]: caminar.min()
```

```
Salida[255]: -3
```

En [256]: `caminar.max()`

Salida[256]: 31

Una estadística más complicada es el primer tiempo de cruce, el paso en el que la caminata aleatoria alcanza un valor particular. Aquí podríamos querer saber cuánto tiempo le tomó a la caminata aleatoria alejarse al menos 10 pasos del origen 0 en cualquier dirección. `np.abs(caminar) >= 10` nos da una matriz booleana que indica dónde la caminata ha alcanzado o excedido 10, pero queremos el índice de los primeros 10 o -10. Resulta que podemos calcular esto usando `argmax`, que devuelve el primer índice del valor máximo en la matriz booleana (Verdadero es el valor máximo):

En [257]: `(np.abs(caminar) >= 10).argmax()`

Salida[257]: 37

Tenga en cuenta que usar `argmax` aquí no siempre es eficiente porque siempre realiza un escaneo completo de la matriz. En este caso especial, una vez que se observa un Verdadero sabemos que es el valor máximo.

Simulación de muchos paseos aleatorios a la vez Si su

objetivo era simular muchos paseos aleatorios, digamos 5000 de ellos, puede generar todos los paseos aleatorios con modificaciones menores al código anterior. Si se pasa una tupla de 2, las funciones `numpy.random` generarán una matriz bidimensional de sorteos, y podemos calcular la suma acumulada en las filas para calcular las 5000 caminatas aleatorias de una sola vez:

En [258]: `ncaminatas = 5000`

En [259]: `npasos = 1000`

En [260]: `sorteos = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 o 1`

En [261]: `pasos = np.where(dibuja > 0, 1, -1)`

En [262]: `caminatas = pasos.cumsum(1)`

En [263]: `paseos`

Salida[263]:

```
matriz([[ 1, [ 1, 0, 1, ..., 8, 7, 8], 0, -1, ..., 34, 33,
           [ 1, 32], 0, -1, ..., 4, 4], 5,
...,
         [ 1, 1, ..., 24, 25, 26], 2, [ 1, 2, 3, ..., 14, 13,
         14], [-1, -2, -3, ..., -24, -23, -22]])
```

Ahora, podemos calcular los valores máximos y mínimos obtenidos en todas las caminatas:

```
En [264]: paseos.max()
Salida[264]: 138
```

```
En [265]: paseos.min()
Salida[265]: -133
```

De estas caminatas, calculemos el tiempo mínimo de cruce en 30 o -30. Esto es un poco complicado porque no todos los 5000 llegan a 30. Podemos verificar esto usando cualquier método:

```
En [266]: hits30 = (np.abs(walks) >= 30).any(1)

In [267]: hits30
Out[267]: array([False,  True,  False, ...,  False,  True,  False], dtype=bool)

In [268]: hits30.sum() # Número que alcanzó 30 o -30
Out[268]: 3410
```

Podemos usar esta matriz booleana para seleccionar las filas de caminatas que realmente cruzan el nivel 30 absoluto y llamar a `argmax` a través del eje 1 para obtener los tiempos de cruce:

```
En [269]: cross_times = (np.abs(walks[hits30]) >= 30).argmax(1)

En [270]: cross_times.mean()
Salida[270]: 498.88973607038122
```

Siéntase libre de experimentar con otras distribuciones para los pasos que no sean lanzamientos de monedas del mismo tamaño. Solo necesita usar una función de generación de números aleatorios diferente, como normal para generar pasos distribuidos normalmente con alguna media y desviación estándar:

```
En [271]: pasos = np.random.normal(loc=0, scale=0.25,
.....:                             tamaño=(ncaminatas, npasos))
```

4.8 Conclusión

Si bien gran parte del resto del libro se centrará en desarrollar habilidades de manejo de datos con pandas, continuaremos trabajando en un estilo similar basado en matrices. En el [Apéndice A](#), profundizaremos en las características de NumPy para ayudarlo a desarrollar aún más sus habilidades de computación de matrices.

Primeros pasos con los pandas

los pandas serán una importante herramienta de interés a lo largo de gran parte del resto del libro. Contiene estructuras de datos y herramientas de manipulación de datos diseñadas para hacer que la limpieza y el análisis de datos sean rápidos y fáciles en Python. pandas a menudo se usa junto con herramientas de computación numérica como NumPy y SciPy, bibliotecas analíticas como statsmodels y scikit-learn, y bibliotecas de visualización de datos como matplotlib. pandas adopta partes significativas del estilo idiomático de computación basada en arreglos de NumPy, especialmente funciones basadas en arreglos y una preferencia por el procesamiento de datos sin bucles for .

Si bien pandas adopta muchos lenguajes de codificación de NumPy, la mayor diferencia es que pandas está diseñado para trabajar con datos tabulares o heterogéneos. NumPy, por el contrario, es más adecuado para trabajar con datos de matrices numéricas homogéneas.

Desde que se convirtió en un proyecto de código abierto en 2010, pandas ha madurado hasta convertirse en una biblioteca bastante grande que se puede aplicar en un amplio conjunto de casos de uso del mundo real. La comunidad de desarrolladores ha crecido a más de 800 colaboradores distintos, que han estado ayudando a construir el proyecto a medida que lo utilizan para resolver sus problemas de datos cotidianos.

A lo largo del resto del libro, utilizo la siguiente convención de importación para pandas:

```
En [1]: importar pandas como pd
```

Por lo tanto, cada vez que vea `pd.` en código, se refiere a pandas. También puede resultarle más fácil importar `Series` y `DataFrame` al espacio de nombres local, ya que se utilizan con mucha frecuencia:

```
En [2]: from pandas import Series, DataFrame
```

5.1 Introducción a las estructuras de datos pandas

Para comenzar con pandas, deberá familiarizarse con sus dos estructuras de datos de caballo de batalla: Series y DataFrame. Si bien no son una solución universal para todos los problemas, brindan una base sólida y fácil de usar para la mayoría de las aplicaciones.

Serie

Una serie es un objeto similar a una matriz unidimensional que contiene una secuencia de valores (de tipos similares a los tipos NumPy) y una matriz asociada de etiquetas de datos, denominada índice. La serie más simple se forma solo a partir de una matriz de datos:

```
En [11]: obj = pd.Series([4, 7, -5, 3])
```

```
En [12]: objeto
```

```
Salida[12]:
```

```
0 4
```

```
1 7
```

```
2 -5
```

```
3 3
```

```
dtipo: int64
```

La representación de cadena de una serie que se muestra de forma interactiva muestra el índice a la izquierda y los valores a la derecha. Dado que no especificamos un índice para los datos, se crea uno predeterminado que consta de los números enteros de 0 a N - 1 (donde N es la longitud de los datos). Puede obtener la representación de matriz y el objeto de índice de la Serie a través de sus valores y atributos de índice, respectivamente:

```
In [13]: obj.values Out[13]:
```

```
array([ 4, 7, -5, 3])
```

```
En [14]: obj.index # como rango (4)
```

```
Salida[14]: RangeIndex(inicio=0, parada=4, paso=1)
```

A menudo, será deseable crear una serie con un índice que identifique cada punto de datos con una etiqueta:

```
En [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
En [16]: obj2
```

```
Salida[16]: base de
```

```
4
```

```
datos 7
```

```
a -5
```

```
c 3
```

```
tipo: int64
```

```
In [17]: obj2.index Out[17]:
```

```
Index(['d', 'b', 'a', 'c'], dtype='objeto')
```

En comparación con las matrices NumPy, puede usar etiquetas en el índice al seleccionar solo valores o un conjunto de valores:

```
En [18]: obj2['a']
Salida[18]: -5

En [19]: obj2['d'] = 6

En [20]: obj2[['c', 'a', 'd']]
Fuera[20]:
c      3
a     -5
      6
d tipo: int64
```

Aquí ['c', 'a', 'd'] se interpreta como una lista de índices, aunque contiene cadenas en lugar de enteros.

Usar funciones NumPy u operaciones similares a NumPy, como filtrar con un valor booleano matriz, la multiplicación escalar o la aplicación de funciones matemáticas, conservará el valor del índice enlace:

```
En [21]: obj2[obj2 > 0]
Fuera[21]:
6 _
b      7
c      3
tipo: int64

En [22]: obj2 * 2
Fuera[22]:
d 12
b     14
a    -10
c      6
tipo: int64

En [23]: np.exp(obj2)
Fuera[23]:
d      403.428793
b    1096.633158
a       0.006738
c     20.085537
tipo: float64
```

Otra forma de pensar en una serie es como un dictado ordenado de longitud fija, ya que es un mapa. ping de valores de índice a valores de datos. Se puede utilizar en muchos contextos en los que podría usa un dictado:

```
En [24]: 'b' en obj2
Salida[24]: Verdadero
```

```
En [25]: 'e' en obj2
Salida[25]: Falso
```

Si tiene datos contenidos en un dictado de Python, puede crear una Serie a partir de él pasando el dictado:

```
En [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregón': 16000, 'Utah': 5000}

En [27]: obj3 = pd.Series(sdata)

En [28]: obj3
Fuera[28]:
Ohio      35000
Oregón    16000
Texas     71000
Utah      5000
tipo: int64
```

Cuando solo está pasando un dict, el índice en la serie resultante tendrá el dict llaves en orden ordenado. Puede anular esto pasando las teclas de dictado en el orden que desea que aparezcan en la Serie resultante:

```
En [29]: estados = ['California', 'Ohio', 'Oregón', 'Texas']

En [30]: obj4 = pd.Series(sdata, index=estados)

En [31]: obj4
Fuera[31]:
California      Yaya
Ohio            35000.0
Oregón          16000.0
Texas           71000.0
tipo: float64
```

Aquí, tres valores encontrados en sdata se colocaron en las ubicaciones apropiadas, pero dado que no se encontró ningún valor para 'California', aparece como NaN (no es un número), lo cual es sidered en pandas para marcar los valores que faltan o NA. Dado que 'Utah' no se incluyó en estados, se excluye del objeto resultante.

Usaré los términos "faltante" o "NA" indistintamente para referirme a los datos faltantes. El Las funciones isnull y notnull en pandas deben usarse para detectar datos faltantes:

```
En [32]: pd.isnull(obj4)
Fuera[32]:
california ohio      Verdadero
Oregón          FALSO
Texas           FALSO
dtipo: booleano

En [33]: pd.notnull(obj4)
Fuera[33]:
```

```

California    FALSO
Ohio          Verdadero
Oregón        Verdadero
Texas         Verdadero
dtype: bool

```

Series también tiene estos como métodos de instancia:

```

En [34]: obj4.isnull()
Fuera[34]:
california ohio    Verdadero
Oregón          FALSO
Texas           FALSO
dtype: bool

```

Discuto el trabajo con datos faltantes con más detalle en el [Capítulo 7](#).

Una función de Series útil para muchas aplicaciones es que se alinea automáticamente por índice etiqueta en operaciones aritméticas:

```

En [35]: obj3
Fuera[35]:
Ohio      35000
Oregón    16000
Tejas     71000
Utah      5000
tipo: int64

En [36]: obj4
Fuera[36]:
California      Yaya
Ohio            35000.0
Oregón          16000.0
Texas           71000.0
tipo: float64

En [37]: obj3 + obj4
Fuera[37]:
California      Yaya
Ohio            70000.0
Oregón          32000.0
Texas           142000.0
Utah            Yaya
dtype: float64

```

Las funciones de alineación de datos se abordarán con más detalle más adelante. si tienes experiencia con las bases de datos, puede pensar en esto como algo similar a una operación de combinación.

Tanto el objeto Serie en sí como su índice tienen un atributo de nombre , que se integra con otras áreas clave de la funcionalidad de pandas:

```
En [38]: obj4.name = 'población'
```

```
En [39]: obj4.index.name = 'estado'
```

```
En [40]: obj4
```

```
Fuera[40]:
```

```
estado
```

```
California      Yaya
```

```
Ohio            35000.0
```

```
Oregón          16000.0
```

```
Texas           71000.0
```

```
Nombre: población, dtype: float64
```

El índice de una serie se puede modificar en el lugar mediante asignación:

```
En [41]: objeto
```

```
Fuera[41]:
```

```
0 4
```

```
1    7
```

```
2   -5
```

```
3    3
```

```
tipo: int64
```

```
En [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

```
En [43]: objeto
```

```
Fuera[43]:
```

```
Beto      4
```

```
Steve     7
```

```
jeff     -5
```

```
Tipo      3
```

```
de Ryan: int64
```

Marco de datos

Un DataFrame representa una tabla rectangular de datos y contiene una colección ordenada. ción de columnas, cada una de las cuales puede ser un tipo de valor diferente (numérico, cadena, booleano, etc). El DataFrame tiene un índice de fila y columna; se puede pensar en como un dictado de Serie, todos comparten el mismo índice. Debajo del capó, los datos se almacenan como uno o más bloques bidimensionales en lugar de una lista, dictado o alguna otra colección de matrices unidimensionales. Los detalles exactos de las partes internas de DataFrame están fuera del alcance de este libro.



Si bien un DataFrame es físicamente bidimensional, puede usarlo para representar datos dimensionales superiores en un formato tabular usando indexación archivística, un tema que discutiremos en [el Capítulo 8](#) y un ingrediente en algunas de las características más avanzadas de manejo de datos en pandas

Hay muchas formas de construir un DataFrame, aunque una de las más comunes es a partir de listas de igual longitud o matrices NumPy:

```
datos = {'estado': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'año': [2000, 2001, 2002, 2001, 2002, 2003], 'pop': [1.5, 1.7,
        3.6, 2.4, 2.9, 3.2]}
marco = pd.DataFrame(datos)
```

El DataFrame resultante tendrá su índice asignado automáticamente como con Series, y las columnas se colocarán en orden:

```
Entrada [45]: marco
Salida[45]:
      estado pop año 0 1,5
Ohio 2000 1 1,7 Ohio 2001
2 3,6 Ohio 2002 3 2,4
Nevada 2001

4 2,9 Nevada 2002 5 3,2
Nevada 2003
```

Si está utilizando el cuaderno Jupyter, los objetos Pandas DataFrame se mostrarán como una tabla HTML más amigable para el navegador.

Para DataFrames grandes, el método head selecciona solo las primeras cinco filas:

```
En [46]: cuadro.cabeza()
Salida[46]:
      estado pop año 0 1,5
Ohio 2000 1 1,7 Ohio 2001
2 3,6 Ohio 2002

3 2,4 Nevada 2001 4 2,9
Nevada 2002
```

Si especifica una secuencia de columnas, las columnas de DataFrame se organizarán en ese orden:

```
En [47]: pd.DataFrame(datos, columnas=['año', 'estado', 'pop'])
Out[47]:
      año población estatal 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 pasa una columna que no está contenida en el dict, aparecerá con valores faltantes en el resultado:

```
En [48]: cuadro2 = pd.DataFrame(datos, columnas=['año', 'estado', 'población', 'deuda'], índice=['uno', 'dos',
.....:                                     'tres', 'cuatro', 'cinco', 'seis'])
.....:
```

En [49]: fotograma2

Fuera[49]:

	año	deuda	pop	estatal
uno	2000	Ohio	1,5	NaN
dos	2001	Ohio	1,7	NaN
tres	2002	Ohio	3,6	NaN
cuatro	2001	Nevada	2.4	NaN
cinco	2002	Nevada	2.9	NaN
seis	2003	Nevada	3.2	NaN

En [50]: frame2.columns

Salida[50]: Índice(['año', 'estado', 'pop', 'deuda'], dtype='objeto')

Una columna en un DataFrame se puede recuperar como una Serie ya sea por notación similar a dict o por atributo:

En [51]: frame2['estado']

Fuera[51]:

uno	Ohio
dos	Ohio
tres	Ohio
cuatro	Nevada
cinco	Nevada
seis	Nevada

Nombre: estado, dtype: objeto

En [52]: cuadro2.año

Fuera[52]:

uno	2000
dos	2001
tres	2002
cuatro	2001
cinco	2002
seis	2003

Nombre: año, dtype: int64



Acceso similar a un atributo (p. ej., marco2.año) y finalización con pestañas de la columna. umn nombres en IPython se proporciona para su comodidad.

frame2[column] funciona para cualquier nombre de columna, pero frame2.column solo funciona cuando el nombre de la columna es una variable de Python válida nombre.

Tenga en cuenta que la Serie devuelta tiene el mismo índice que el DataFrame y su nombre El atributo se ha establecido correctamente.

Las filas también se pueden recuperar por posición o nombre con el atributo especial loc (muy más sobre esto más adelante):

```

En [53]: frame2.loc['tres']
Fuera[53]:
estado      2002
del año     Ohio
deuda       3.6
popular     Yaya
Nombre: tres, dtype: objeto

```

Las columnas se pueden modificar por asignación. Por ejemplo, la columna vacía 'deuda' se le podría asignar un valor escalar o una matriz de valores:

```
En [54]: frame2['deuda'] = 16.5
```

```

En [55]: fotograma2
Fuera[55]:
          año deuda pop estatal
uno      2000 Ohio 1,5 16,5
dos      2001 Ohio 1,7 16,5
tres     2002 Ohio 3,6 16,5
cuatro   2001 Nevada 2.4 16.5
cinco    2002 Nevada 2.9 16.5
seis     2003 Nevada 3.2 16.5

```

```
En [56]: frame2['deuda'] = np.arange(6.)
```

```

En [57]: fotograma2
Fuera[57]:
          año deuda pop estatal
uno      2000 Ohio 1,5 0,0
dos      2001 Ohio 1,7 1,0
tres     2002 Ohio 3,6 2,0
cuatro   2001 Nevada 2.4 3.0
cinco    2002 Nevada 2.9 4.0
seis     2003 Nevada 3.2 5,0

```

Cuando está asignando listas o matrices a una columna, la longitud del valor debe coincidir con la longitud de la trama de datos. Si asigna una serie, sus etiquetas se realinearán exactamente para el índice de DataFrame, insertando valores faltantes en cualquier agujero:

```
En [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['dos', 'cuatro', 'cinco'])
```

```
En [59]: frame2['deuda'] = val
```

```

En [60]: fotograma2
Fuera[60]:
          año deuda pop estatal
uno      2000 Ohio 1,5 NaN
dos      2001 Ohio 1,7 -1,2
tres     2002 Ohio 3,6 NaN
cuatro   2001 Nevada 2.4 -1.5
cinco    2002 Nevada 2.9 -1.7
seis     2003 Nevada 3.2 NaN

```

Asignar una columna que no existe creará una nueva columna. La palabra clave del eliminar columnas como con un dict.

Como ejemplo de del, primero agrego una nueva columna de valores booleanos donde el estado columna es igual a 'Ohio':

```
En [61]: cuadro2['oriental'] = cuadro2.estado == 'Ohio'

En [62]: fotograma2
Fuera[62]:
      año deuda pop estatal del este
uno      2000 Ohio 1,5 NaN Verdadero
dos      2001 Ohio 1,7 -1,2          Verdadero
tres 2002 Ohio 3,6 NaN cuatro 2001 Nevada          Verdadero
2,4 -1,5 cinco 2002 Nevada 2,9 -1,7 seis 2003      FALSO
Nevada 3,2 NaN                                FALSO
                                                FALSO
```



No se pueden crear columnas nuevas con la sintaxis `frame2.eastern`.

El método del `del` se puede usar para eliminar esta columna:

```
En [63]: del frame2['oriental']

En [64]: marco2.columnas
Salida[64]: Índice(['año', 'estado', 'pop', 'deuda'], dtype='objeto')
```



La columna devuelta de indexar un DataFrame es una vista en el datos subyacentes, no una copia. Por lo tanto, cualquier modificación in situ al La serie se reflejará en el DataFrame. La columna puede ser copiado explícitamente con el método de copia de la Serie.

Otra forma común de datos es un dictado anidado de dictados:

```
En [65]: población = {'Nevada': {2001: 2.4, 2002: 2.9},
.....:               'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

Si el dict anidado se pasa al DataFrame, los pandas interpretarán las claves del dict externo como las columnas y las claves internas como los índices de las filas:

```
En [66]: frame3 = pd.DataFrame(pop)

En [67]: fotograma3
Fuera[67]:
      nevada ohio
2000      NaN 1,5
```

```

2001      2.4 1.7
2002      2.9 3.6

```

Puede transponer el DataFrame (intercambiar filas y columnas) con una sintaxis similar a un matriz NumPy:

```

En [68]: cuadro3.T
Fuera[68]:
          2000 2001 2002
Nevada NaN  2.4 2.9
Ohio      1.5 1.7 3.6

```

Las claves en los dictados internos se combinan y ordenan para formar el índice en el resultado.

Esto no es cierto si se especifica un índice explícito:

```

En [69]: pd.DataFrame(pop, índice=[2001, 2002, 2003])
Fuera[69]:
          nevada ohio
2001      2.4 1.7
2002      2.9 3.6
2003      NaN NaN

```

Los dictados de series se tratan de la misma manera:

```

En [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
.....:           'Nevada': frame3['Nevada'][:2]}

En [71]: pd.DataFrame(pdata)
Fuera[71]:
          nevada ohio
2000      NaN 1.5
2001      2.4 1.7

```

Para obtener una lista completa de las cosas que puede pasar al constructor de DataFrame, consulte [la Tabla 5-1](#).

Si el índice y las columnas de un DataFrame tienen sus atributos de nombre establecidos, estos también serán desplegado:

```

En [72]: frame3.index.name = 'año'; frame3.columns.name = 'estado'

En [73]: fotograma3
Fuera[73]:
estado Nevada Ohio
año
2000      NaN 1.5
2001      2.4 1.7
2002      2.9 3.6

```

Al igual que con Series, el atributo de valores devuelve los datos contenidos en el DataFrame como un ndarray bidimensional:

```

En [74]: frame3.values
Fuera[74]:
matriz([[ nan, 1.5],

```

```
[ 2.4, 1.7], [ 2.9, 3.6]])
```

Si las columnas de DataFrame son de diferentes tipos, se elegirá el tipo de la matriz de valores para acomodar todas las columnas:

```
In [75]: frame2.values Out[75]:
array([[2000,
'Ohio', 1.5, nan], [2001, 'Ohio', 1.7, -1.2],
[2002, 'Ohio', 3.6, nan], [2001,
'Nevada', 2.4, -1.5], [2002, 'Nevada',
2.9, -1.7], [2003, 'Nevada', 3.2, nan]],
dtype=objeto)
```

Tabla 5-1. Posibles entradas de datos al constructor de DataFrame

Tipo	notas
Dict 2D ndarray	Una matriz de datos, pasando etiquetas de fila y columna opcionales
de matrices, listas o tuplas	Cada secuencia se convierte en una columna en el DataFrame; todas las secuencias deben tener la misma longitud
NumPy matriz estructurada/registro	Tratado como el caso "dict of arrays"
dictado de la serie	Cada valor se convierte en una columna; los índices de cada serie se unen para formar el índice de fila del resultado si no se pasa un índice explícito
dictado de dictados	Cada dictado interno se convierte en una columna; las claves se unen para formar el índice de fila como en el "dict of Caso de la serie
Lista de dictados o Serie	Cada elemento se convierte en una fila en el DataFrame; unión de claves dict o índices de serie se convierten en el Etiquetas de columna de DataFrame
Lista de listas o tuplas	Tratado como el caso "2D ndarray"
Otro marco de datos	Los índices de DataFrame se utilizan a menos que se pasen otros diferentes
Matriz enmascarada NumPy	Como el caso "2D ndarray", excepto que los valores enmascarados se vuelven NA/faltan en el resultado de DataFrame

Objetos de índice

Los objetos de índice de pandas son responsables de contener las etiquetas de los ejes y otros metadatos (como el nombre o los nombres de los ejes). Cualquier matriz u otra secuencia de etiquetas que utilice al construir una Serie o Marco de datos se convierte internamente en un Índice:

```
En [76]: obj = pd.Series(rango(3), índice=['a', 'b', 'c'])

En [77]: índice = obj.index

En [78]: índice
Salida[78]: Índice(['a', 'b', 'c'], dtype='objeto')

En [79]: índice[1:]
Salida[79]: Índice(['b', 'c'], dtype='objeto')
```

Los objetos de índice son inmutables y, por lo tanto, el usuario no puede modificarlos:

```
índice[1] = 'd' # TypeError
```

La inmutabilidad hace que sea más seguro compartir objetos de índice entre estructuras de datos:

```
En [80]: etiquetas = pd.Index(np.arange(3))
```

```
En [81]: etiquetas
```

```
Salida[81]: Int64Index([0, 1, 2], dtype='int64')
```

```
En [82]: obj2 = pd.Series([1.5, -2.5, 0], index=etiquetas)
```

```
Entrada [83]:
```

```
obj2
```

```
Salida
```

```
[83]: 0 1.5 1 -2.5
```

```
2      0.0
```

```
tipo: float64
```

```
In [84]: obj2.index es etiquetas Out[84]: True
```



Algunos usuarios no suelen aprovechar las capacidades proporcionadas por los índices, pero debido a que algunas operaciones arrojarán resultados que contienen datos indexados, es importante comprender cómo funcionan.

Además de ser similar a una matriz, un índice también se comporta como un conjunto de tamaño fijo:

```
En [85]: fotograma3
```

```
Salida[85]:
```

```
estado Nevada Ohio
```

```
año
```

```
2000      NaN 1,5
```

```
2001      2.4 1.7
```

```
2002      2,9 3,6
```

```
In [86]: frame3.columns Out[86]:
```

```
Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

```
In [87]: 'Ohio' en frame3.columns Out [87]:
```

```
Verdadero
```

```
En [88]: 2003 en frame3.index Salida [88]:
```

```
Falso
```

A diferencia de los conjuntos de Python, un índice de pandas puede contener etiquetas duplicadas:

```
En [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
```

```
En [90]: dup_labels
```

```
Salida[90]: Índice(['foo', 'foo', 'bar', 'bar'], dtype='objeto')
```

Las selecciones con etiquetas duplicadas seleccionarán todas las apariciones de esa etiqueta.

Cada índice tiene una serie de métodos y propiedades para establecer la lógica, que responden a otras preguntas comunes sobre los datos que contiene. Algunos útiles se resumen en [la Tabla 5-2](#).

Tabla 5-2. Algunos métodos y propiedades de Index

Método	Descripción
adjuntar	Concatenar con objetos de índice adicionales, produciendo un nuevo índice
diferencia	Calcule la diferencia de conjuntos como un índice
intersección	Calcular conjunto de intersección
Unión	Unión de conjuntos de cómputos
es en	Calcule la matriz booleana que indica si cada valor está contenido en la colección pasada
borrar	Calcule el nuevo índice con el elemento en el índice que eliminé
gota	Calcule el nuevo índice eliminando los valores pasados
insertar	Calcule el nuevo índice insertando el elemento en el índice 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
único	Calcule la matriz de valores únicos en el índice

5.2 Funcionalidad esencial

Esta sección lo guiará a través de la mecánica fundamental de la interacción con los datos contenidos en una serie o marco de datos. En los próximos capítulos, profundizaremos en los temas de análisis y manipulación de datos utilizando pandas. Este libro no pretende servir como documentación exhaustiva para la biblioteca de pandas; en su lugar, nos centraremos en las características más importantes, dejando las cosas menos comunes (es decir, más esotéricas) para que las explore por su cuenta.

Reindexación Un

método importante en los objetos pandas es la reindexación, lo que significa crear un nuevo objeto con los datos conformes a un nuevo índice. Considere un ejemplo:

```
En [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

Entrada [92]: obj
Salida[92]:
d 4.5 b
      7.2
un -5.3
      3.6
c dtipo: float64
```

Llamar a `reindex` en esta serie reorganiza los datos de acuerdo con el nuevo índice, introduciendo valores faltantes si alguno de los valores del índice no estaba ya presente:

```
En [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

```
Entrada [94]: obj2
```

```
Salida
```

```
[94]: a -5.3
```

```
b 7.2
```

```
c 3.6
```

```
d 4.5
```

```
mi Yaya
```

```
tipo: float64
```

Para datos ordenados como series de tiempo, puede ser deseable hacer alguna interpolación o completar valores al reindexar. La opción de método nos permite hacer esto, usando un método como `ffill`, que reenvía los valores:

```
En [95]: obj3 = pd.Series(['azul', 'morado', 'amarillo'], index=[0, 2, 4])
```

```
En [96]: obj3
```

```
Out[96]: 0
```

```
azul 2 púrpura
```

```
amarillo dtype:
```

```
4 objeto
```

```
En [97]: obj3.reindex(rango(6), método='relleno')
```

```
Out[97]:
```

```
azul 0
```

```
azul violeta
```

```
2 violeta
```

```
amarillo
```

```
3 4 amarillo
```

```
5 dtype:
```

```
objeto
```

Con `DataFrame`, la reindexación puede modificar el índice (fila), las columnas o ambos. Cuando se pasa solo una secuencia, vuelve a indexar las filas en el resultado:

```
En [98]: marco = pd.DataFrame(np.arange(9).reshape((3, 3)), índice=['a', 'c', 'd'],
.....:                        columnas=['Ohio', 'Texas',
.....:                        'California'])
```

```
En [99]: cuadro
```

```
Fuera[99]:
```

```
Ohio Texas California
```

```
a 0 1 2
```

```
c 3 4 5
```

```
d 6 7 8
```

```
En [100]: cuadro2 = cuadro.reindex(['a', 'b', 'c', 'd'])
```

```
En [101]: fotograma2
Fuera[101]:
Ohio Texas California
a 0.0 b      1.0      2.0
NaN      NaN      NaN
c 3.0 d 6.0    4.0      5.0
      7.0      8.0
```

Las columnas se pueden volver a indexar con la palabra clave de columnas :

```
En [102]: estados = ['Texas', 'Utah', 'California']

En [103]: frame.reindex(columnas=estados)
Fuera[103]:
Texas Utah California
a      1 NaN      2
c      4 NaN      5
d      7 NaN      8
```

Consulte la [Tabla 5-3](#) para obtener más información sobre los argumentos para reindexar.

Como exploraremos con más detalle, puede reindexar de manera más sucinta mediante la indexación de etiquetas con loc, y muchos usuarios prefieren usarlo exclusivamente:

```
En [104]: frame.loc[['a', 'b', 'c', 'd'], estados]
Salida[104]:
Texas Utah California 1.0 NaN 2.0
a
b      NaN NaN      Yaya
c      4.0 NaN      5.0
d      7.0 NaN      8.0
```

Tabla 5-3. Argumentos de función de reindexación

Argumento	Descripción
índice	Nueva secuencia para usar como índice. Puede ser una instancia de índice o cualquier otra estructura de datos de Python similar a una secuencia. Un índice se utilizará exactamente como está sin ninguna copia.
método	Método de interpolación (relleno); 'ffill' se llena hacia adelante, mientras que 'bfill' se llena hacia atrás.
fill_value	Valor de sustitución que se utilizará al introducir datos faltantes mediante la reindexación.
límite	Cuando se rellene hacia delante o hacia atrás, espacio de tamaño máximo (en número de elementos) a rellenar.
Tolerancia	Cuando se rellene hacia delante o hacia atrás, la brecha de tamaño máximo (en distancia numérica absoluta) para rellenar las coincidencias inexactas.
nivel	Haga coincidir el índice simple en el nivel de MultiIndex; de lo contrario, seleccione un subconjunto de.
Copiar	Si es Verdadero, copie siempre los datos subyacentes incluso si el índice nuevo es equivalente al índice anterior; si es False, no copie los datos cuando los índices sean equivalentes.

Eliminar entradas de un eje Eliminar una o más

entradas de un eje es fácil si ya tiene una matriz o lista de índice sin esas entradas. Como eso puede requerir un poco de munging y lógica establecida, el

El método drop devolverá un nuevo objeto con el valor o valores indicados eliminados de un eje:

```
En [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
Entrada [106]: obj
```

```
Salida [106]:
```

```
..      0.0
...      1.0
c       2.0
d       3.0
mi      4.0
tipo: float64
```

```
En [107]: nuevo_obj = obj.drop('c')
```

```
Entrada [108]: nuevo_obj
```

```
Salida [108]:
```

```
a       0.0
        1.0
bd       3.0
mi      4.0
tipo: float64
```

```
En [109]: obj.drop(['d', 'c'])
```

```
Salida [109]:
```

```
a       0.0
b       1.0
mi      4.0
dtipo: float64
```

Con DataFrame, los valores de índice se pueden eliminar de cualquiera de los ejes. Para ilustrar esto, primero creamos un DataFrame de ejemplo:

```
En [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio',
.....:                                     'Colorado', 'Utah', 'New York'], column= ['uno', 'dos', 'tres', 'cuatro'])
.....:
```

```
En [111]: datos
```

```
Fuera[111]:
```

```
      uno dos tres cuatro 3
Ohio      0   1     2
Colorado   4   5     6   7
Utah 9     8     10   11
Nueva York 12 13     14   15
```

Al llamar a drop con una secuencia de etiquetas, se eliminarán los valores de las etiquetas de fila (eje 0):

```
En [112]: data.drop(['Colorado', 'Ohio'])
```

```
Fuera[112]:
```

```
      uno dos tres CUATRO
Utah      8   9     10   11
Nueva York 12 13     14   15
```

Puede eliminar valores de las columnas pasando `axis=1` o `axis='columns'`:

```
En [113]: data.drop('dos', eje=1)
Fuera[113]:
      uno tres cuatro 3
Ohio      0      2
Colorado  4      6      7
Utah 8      10     11
Nueva York 12     14     15

En [114]: data.drop(['dos', 'cuatro'], eje='columnas')
Fuera[114]:
      uno tres
Ohio      0      2
Colorado  4
Utah 8      6 10
Nueva York 12     14
```

Muchas funciones, como `soltar`, que modifican el tamaño o la forma de una serie o un marco de datos, pueden manipular un objeto en el lugar sin devolver un objeto nuevo:

```
En [115]: obj.drop('c', inplace=True)

Entrada [116]: obj
Salida [116]:
a      0.0
      1.0
bd     3.0
mi     4.0
tipo: float64
```

Tenga cuidado con el `inplace`, ya que destruye cualquier dato que se suelte.

Indexación, selección y filtrado

La indexación de series (`obj[...]`) funciona de manera análoga a la indexación de matrices NumPy, excepto que puede usar los valores de índice de Series en lugar de solo números enteros. Aquí hay algunos ejemplos de esto:

```
En [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

Entrada [118]: obj
Salida [118]:
a      0.0
b      1.0
c      2.0
d  3.0 dtipo:
float64

En [119]: objeto['b']
Salida[119]: 1.0
```

```

En [120]: objeto[1]
Salida[120]: 1.0

En [121]: obj[2:4]
Salida[121]:
c      2.0
      3.0
d tipo: float64

En [122]: obj[['b', 'a', 'd']]
Salida[122]:
b 1.0
a      0.0
d 3.0 d tipo:
float64

```

```

En [123]: objeto[[1, 3]]
Salida[123]:
b 1.0 d 3.0
d tipo:
float64

```

```

En [124]: obj[obj < 2]
Salida[124]:
a      0.0
      1.0
tipo b: float64

```

La segmentación con etiquetas se comporta de manera diferente a la segmentación normal de Python en que el punto final es inclusivo:

```

En [125]: obj['b':'c']
Salida[125]:
b 1.0
c      2.0
tipo: float64

```

La configuración con estos métodos modifica la sección correspondiente de la Serie:

```

En [126]: obj['b':'c'] = 5

Entrada [127]: obj
Salida [127]:
a      0.0
b      5.0
c      5.0
d 3.0 d tipo:
float64

```

La indexación en un DataFrame es para recuperar una o más columnas con un solo valor o secuencia:

```
En [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio',
.....:                                'Colorado', 'Utah', 'New York'], column= ['uno', 'dos', 'tres', 'cuatro'])
.....:
```

```
En [129]: datos
```

```
Fuera[129]:
```

	uno	dos	tres	CUATRO
Ohio	0	1	2	3
colorado 5	4			
Utah 9	8		6 10	7 11
Nueva York	12 13		14	15

```
En [130]: datos['dos']
```

```
Salida[130]:
```

Ohio	1
Colorado	5
Utah	9
Nueva York	13

Nombre: dos, dtype: int64

```
En [131]: datos[['tres', 'uno']]
```

```
Fuera[131]:
```

	tres	uno
Ohio	2	
Colorado	6	0 4
Utah	10	8
Nueva York	14 12	

La indexación como esta tiene algunos casos especiales. Primero, cortando o seleccionando datos con una matriz booleana:

```
En [132]: datos[:2]
```

```
Fuera[132]:
```

	uno	dos	tres	CUATRO
Ohio	0	1	2	3
Colorado	4	5	6	7

```
En [133]: datos[datos['tres'] > 5]
```

```
Fuera[133]:
```

	uno	dos	tres	CUATRO
Colorado	4	5	6	7
Utah 9	8		10	11
Nueva York	12 13		14	15

La sintaxis de selección de fila `data[:2]` se proporciona para su comodidad. Pasar un solo elemento o una lista al operador `[]` selecciona columnas.

Otro caso de uso es la indexación con un DataFrame booleano, como uno producido por una comparación escalar:

```

En [134]: datos < 5
Fuera[134]:
      uno      dos tres cuatro
Ohio      Verdadero Verdadero Verdadero Verdadero
Colorado Verdadero Falso Falso Falso
Utah      Falso Falso Falso Falso
Nueva York Falso Falso Falso Falso

```

```

En [135]: datos[datos < 5] = 0

```

```

En [136]: datos
Fuera[136]:
      uno dos tres CUATRO
Ohio      0      0      0      0
Colorado  0      5      6      7
Utah 9     8      10     11
Nueva York 12 13      14     15

```

Esto hace que DataFrame sea sintácticamente más como una matriz NumPy bidimensional en este caso particular.

Selección con loc e iloc

Para la indexación de etiquetas de DataFrame en las filas, introduzco los operadores especiales de indexación loc e iloc. Le permiten seleccionar un subconjunto de filas y columnas de un DataFrame con notación similar a NumPy utilizando etiquetas de eje (loc) o números enteros (iloc).

Como ejemplo preliminar, seleccionemos una sola fila y varias columnas por etiqueta:

```

En [137]: data.loc['Colorado', ['dos', 'tres']]
Fuera[137]:
dos      5
tres     6
Nombre: Colorado, tipo de d: int64

```

Luego realizaremos algunas selecciones similares con números enteros usando iloc:

```

En [138]: datos.iloc[2, [3, 0, 1]]
Fuera[138]:
cuatro 11 8
uno
dos      9
Nombre: Utah, tipo de d: int64

```

```

En [139]: datos.iloc[2]
Fuera[139]:
uno      8
dos      9
tres     10
cuatro   11
Nombre: Utah, tipo de d: int64

```

```
En [140]: datos.iloc[[1, 2], [3, 0, 1]]
Fuera[140]:
          cuatro uno dos 7 5
Colorado          0
Utah              11   8   9
```

Ambas funciones de indexación funcionan con sectores además de etiquetas individuales o listas de etiquetas:

```
En [141]: data.loc['Utah', 'dos']
Fuera[141]:
Ohio          0
Colorado      5
Utah          9
Nombre: dos, dtype: int64

En [142]: datos.iloc[:, :3][datos.tres > 5]
Fuera[142]:
          uno dos tres 5 6
Colorado          0
Utah 9            8    10
Nueva York 12 13    14
```

Entonces, hay muchas formas de seleccionar y reorganizar los datos contenidos en un objeto pandas. Para DataFrame, la Tabla 5-4 proporciona un breve resumen de muchos de ellos. Como verá más adelante, hay varias opciones adicionales para trabajar con índices jerárquicos.



Cuando diseñé pandas originalmente, sentí que tener que escribir `frame[:, col]` para seleccionar una columna era demasiado detallado (y propenso a errores), ya que la selección de columnas es una de las operaciones más comunes. Hice el compromiso de diseño para impulsar todo el comportamiento de indexación elegante (tanto etiquetas como números enteros) en el operador `ix`. En la práctica, esto condujo a muchos casos extremos en datos con etiquetas de eje de enteros, por lo que el equipo de pandas decidió crear los operadores `loc` e `iloc` para manejar la indexación estrictamente basada en etiquetas y en enteros, respectivamente.

El operador de indexación `ix` todavía existe, pero está obsoleto. No recomiendo usarlo.

Tabla 5-4. Opciones de indexación con DataFrame

	notas
<code>df[val]</code>	Seleccione una sola columna o secuencia de columnas del DataFrame; conveniencias de casos especiales: matriz booleana (filas de filtro), división (filas de división) o trama de datos booleana (valores establecidos según algún criterio)
<code>df.loc[valor]</code>	Selecciona una sola fila o un subconjunto de filas del DataFrame por etiqueta
<code>df.loc[:, val]</code>	Selecciona una sola columna o subconjunto de columnas por etiqueta
<code>df.loc[val1, val2]</code>	Seleccionar filas y columnas por etiqueta
<code>df.iloc[donde]</code>	Selecciona una sola fila o un subconjunto de filas del DataFrame por posición entera

notas	
<code>df.iloc[:, where]</code>	Selecciona una sola columna o subconjunto de columnas por posición entera
<code>df.iloc[where_i, where_j]</code>	Seleccione filas y columnas por posición de entero Seleccione un solo valor escalar por
<code>df.at[etiqueta_i, etiqueta_j]</code>	etiqueta de fila y columna Seleccione un solo valor escalar por
<code>df.iat[i, j]</code>	posición de fila y columna (enteros)
método de reindexación	Seleccionar filas o columnas por etiquetas
Métodos <code>get_value</code> , <code>set_value</code>	Seleccionar valor único por etiqueta de fila y columna

Índices enteros

Trabajar con objetos pandas indexados por enteros es algo que a menudo hace tropezar a los nuevos usuarios debido a algunas diferencias con la semántica de indexación en las estructuras de datos integradas de Python, como listas y tuplas. Por ejemplo, es posible que no espere que el siguiente código genere un error:

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

En este caso, los pandas podrían "recorrer" a la indexación de enteros, pero es difícil hacer esto en general sin introducir errores sutiles. Aquí tenemos un índice que contiene 0, 1, 2, pero inferir lo que quiere el usuario (indexación basada en etiquetas o basada en la posición) es difícil:

```
Entrada [144]: ser
Salida[144]:
0 0.0
1    1.0
2    2.0
tipo: float64
```

Por otro lado, con un índice no entero, no hay posibilidad de ambigüedad:

```
En [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

En [146]: ser2[-1]
Salida[146]: 2.0
```

Para mantener la coherencia, si tiene un índice de eje que contiene números enteros, la selección de datos siempre estará orientada a etiquetas. Para un manejo más preciso, use `loc` (para etiquetas) o `iloc` (para números enteros):

```
En [147]: ser[:1]
Salida[147]:
0 0.0
tipo: float64

En [148]: ser.loc[:1]
Salida[148]:
0 0.0
1    1.0
```

```
tipo: float64
```

```
En [149]: ser.iloc[:1]
```

```
Salida[149]:
```

```
0 0.0 dtype:
```

```
float64
```

Aritmética y alineación de datos Una

característica importante de pandas para algunas aplicaciones es el comportamiento de la aritmética entre objetos con diferentes índices. Cuando está sumando objetos, si algún par de índices no es el mismo, el índice respectivo en el resultado será la unión de los pares de índices. Para los usuarios con experiencia en bases de datos, esto es similar a una unión externa automática en las etiquetas de índice. Veamos un ejemplo:

```
En [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
```

```
En [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
```

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

```
En [152]: s1
```

```
Fuera[152]:
```

```
a      7.3
```

```
c -2.5 d 3.4
```

```
mi      1.5
```

```
tipo: float64
```

```
Entrada [153]:
```

```
s2 Salida
```

```
[153]: a -2.1
```

```
c      3.6
```

```
e -1.5
```

```
F      4.0
```

```
3.1
```

```
g tipo: float64
```

Sumando estos rendimientos juntos:

```
Entrada [154]: s1 + s2
```

```
Salida [154]:
```

```
a      5.2
```

```
c      1.1
```

```
d
```

```
mi      NaN 0.0
```

```
F      Yaya
```

```
Yaya
```

```
tipo g: float64
```

La alineación de datos internos introduce valores faltantes en las ubicaciones de las etiquetas que no se superponen. Los valores faltantes se propagarán luego en más cálculos aritméticos.

En el caso de DataFrame, la alineación se realiza tanto en las filas como en las columnas:

```
En [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=lista('bcd'),
.....:                      índice=['Ohio', 'Texas', 'Colorado'])

En [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), column=list('bde'), index=['Utah', 'Ohio', 'Texas ', 'Oregón'])
.....:

En [157]: df1
Fuera[157]:
```

	b	c	d
Ohio	0,0	1,0	2,0
Texas	3,0	4,0	5,0
Colorado	6,0	7,0	8,0

```
En [158]: df2
Fuera[158]:
```

	b	d
Utah	0,0	1,0
Ohio	3,0	4,0
Tejas	6,0	7,0
Oregón	9,0	10,0

Al sumar estos, se obtiene un DataFrame cuyo índice y columnas son las uniones de los de cada DataFrame:

```
En [159]: df1 + df2
Fuera[159]:
```

	b	c	Delaware
Colorado	NaN	NaN	NaN
Ohio	3,0	NaN	6,0
Oregón	NaN	NaN	NaN
Texas	9,0	NaN	12,0
Utah	NaN	NaN	NaN

Dado que las columnas 'c' y 'e' no se encuentran en ambos objetos DataFrame, aparecen como faltantes en el resultado. Lo mismo vale para las filas cuyas etiquetas no son comunes a ambos objetos.

Si agrega objetos DataFrame sin etiquetas de columna o fila en común, el resultado contendrá todos los valores nulos:

```
En [160]: df1 = pd.DataFrame({'A': [1, 2]})

En [161]: df2 = pd.DataFrame({'B': [3, 4]})

En [162]: df1
Fuera[162]:
```

A
0
1

```
En [163]: df2
```

```
Fuera[163]:
B
0 3
1 4
```

```
En [164]: df1 - df2
Fuera[164]:
AB
0 NaN NaN
1 NaN NaN
```

Métodos aritméticos con valores de relleno

En las operaciones aritméticas entre objetos indexados de manera diferente, es posible que desee completar con un valor especial, como 0, cuando se encuentra una etiqueta de eje en un objeto pero no en el otro:

```
En [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), column=list('abcd'))
.....:

En [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)), column=list('abcde'))
.....:

En [167]: df2.loc[1, 'b'] = np.nan

En [168]: df1
Fuera[168]:
   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

En [169]: df2
Fuera[169]:
   a    b    c    d    mi
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
```

La suma de estos da como resultado valores de NA en las ubicaciones que no se superponen:

```
En [170]: df1 + df2
Fuera[170]:
   a    b    c    d    mi
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
```

Usando el método add en df1, paso df2 y un argumento a fill_value:

```
En [171]: df1.add(df2, fill_value=0)
Fuera[171]:
```

```
      a      b      c      d      mi
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
```

Consulte la [Tabla 5-5](#) para obtener una lista de los métodos Series y DataFrame para la aritmética. Cada uno de ellos tiene una contraparte, comenzando con la letra r, que tiene argumentos invertidos. Así que estas dos declaraciones son equivalentes:

```
En [172]: 1 / df1
Fuera[172]:
      a      b      c
0      d inf 1.000000 0.500000 0.333333
1 0,250000 0,200000 0,166667 0,142857
2 0,125000 0,111111 0,100000 0,090909

En [173]: df1.rdiv(1)
Fuera[173]:
      a      b      c
0      d inf 1.000000 0.500000 0.333333
1 0,250000 0,200000 0,166667 0,142857 2 0,125000 0,111111
0,100000 0,090909
```

De manera relacionada, al volver a indexar una serie o un marco de datos, también puede especificar un valor de relleno diferente:

```
En [174]: df1.reindex(columns=df2.columns, fill_value=0)
Fuera[174]:
      a      b      c      de 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
agregar, agregar	Métodos para la suma (+)
sub, rsub	Métodos para restar (-)
div, rdiv	Métodos de división (/)
floordiv, rfloordiv	Métodos de división de piso (//)
mul, rmul	Métodos de multiplicación (*)
pow, pow	Métodos de exponenciación (**)

Operaciones entre DataFrame y Series Al igual que

con los arreglos NumPy de diferentes dimensiones, también se define la aritmética entre DataFrame y Series. Primero, como ejemplo motivador, considere la diferencia entre una matriz bidimensional y una de sus filas:

```
En [175]: arr = np.arange(12.).reshape((3, 4))
```

```
Entrada [176]:
```

```
matriz
```

```
Salida[176]:      1., 2., 3., 5., 6., 7., 9.,  
              10., 11.]])
```

```
matriz([[ 0., [ 4., [ 8.,
```

```
En [177]: arr[0]
```

```
Salida[177]: matriz([ 0., 1., 2., 3.]])
```

```
En [178]: arr - arr[0]
```

```
Salida[178]:
```

```
matriz([[ 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.

Esto se conoce como transmisión y se explica con más detalle en relación con las matrices NumPy generales en el **Apéndice A**. Las operaciones entre un DataFrame y una Serie son similares:

```
En [179]: marco = pd.DataFrame(np.arange(12.).reshape((4, 3)), column=list('bde'),
```

```
.....:                                index=['Utah', 'Ohio', 'Texas ',
```

```
.....:                                'Oregón'])
```

```
En [180]: serie = cuadro.iloc[0]
```

```
En [181]: cuadro
```

```
Fuera[181]:
```

```
      b      d      mi  
Utah    0,0 1,0 2,0  
Ohio     3,0 4,0 5,0  
Tejas 6,0 7,0 8,0  
Oregón 9,0 10,0 11,0
```

```
Entrada [182]: serie
```

```
Salida
```

```
[182]: b 0.0
```

```
d      1.0
```

```
mi      2.0
```

```
Nombre: Utah, dtype: float64
```

De forma predeterminada, la aritmética entre DataFrame y Series coincide con el índice de Series en las columnas de DataFrame, transmitiendo las filas:

```
En [183]: cuadro - serie
```

```
Fuera[183]:
```

```
      b      d      mi  
Utah    0.0 0.0 0.0  
Ohio     3.0 3.0 3.0  
Tejas 6,0 6,0 6,0  
Oregón 9,0 9,0 9,0
```

Si no se encuentra un valor de índice en las columnas de DataFrame o en el índice de Series, los objetos se reindexarán para formar la unión:

```
En [184]: serie2 = pd.Series(rango(3), índice=['b', 'e', 'f'])
```

```
En [185]: cuadro + serie2
```

```
Fuera[185]:
```

	bd	ef
Utah	0,0 NaN	3,0 NaN
Ohio	3,0 NaN	6,0 NaN
Texas	6,0 NaN	9,0 NaN
Oregón	9,0 NaN	12,0 NaN

Si, en cambio, desea transmitir sobre las columnas, haciendo coincidir las filas, debe usar uno de los métodos aritméticos. Por ejemplo:

```
En [186]: serie3 = fotograma['d']
```

```
En [187]: cuadro
```

```
Fuera[187]:
```

	b	d	mi
Utah	0,0	1,0	2,0
Ohio	3,0	4,0	5,0
Texas	6,0	7,0	8,0
Oregón	9,0	10,0	11,0

```
En [188]: serie3
```

```
Fuera[188]:
```

Utah	1.0
Ohio	4.0
Texas	7.0
Oregón	10.0

Nombre: d, dtype: float64

```
En [189]: frame.sub(serie3, eje='índice')
```

```
Fuera[189]:
```

	bd	Utah	mi
-1,0	0,0	1,0	1,0
Ohio	-1,0	0,0	1,0
Texas	-1,0	0,0	0,0
Oregón	-1,0	0,0	1,0

El número de eje que pasa es el eje en el que debe coincidir. En este caso, nos referimos a hacer coincidir el índice de fila de DataFrame (axis = 'index' o axis = 0) y transmitirlo.

Aplicación y mapeo de funciones

NumPy ufuncs (métodos de matriz de elementos) también funcionan con objetos pandas:

```
En [190]: marco = pd.DataFrame(np.random.randn(4, 3), column=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```

En [191]: cuadro
Fuera[191]:
           b           d           mi
Utah -0,204708 0,478943 -0,519439 Ohio
      -0,555730 1,965781 1,393406
Texas 0.092908 0.281746 0.769023
Oregón 1,246435 1,007189 -1,296221

```

```

En [192]: np.abs(fotograma)
Fuera[192]:
           b           d           mi
Utah      0.204708 0.478943 0.519439
Ohio      0.555730 1.965781 1.393406
Texas 0.092908 0.281746 0.769023 Oregón
1.246435 1.007189 1.296221

```

Otra operación frecuente es aplicar una función en arreglos unidimensionales a cada columna o fila. El método de aplicación de DataFrame hace exactamente esto:

```

En [193]: f = lambda x: x.max() - x.min()

En [194]: cuadro.aplicar(f)
Salida[194]:
b 1.802165 d
1.684034
mi      2.689627
tipo: float64

```

Aquí, la función `f`, que calcula la diferencia entre el máximo y el mínimo de una serie, se invoca una vez en cada columna del cuadro. El resultado es una Serie que tiene como índice las columnas del marco .

Si pasa `axis='columns'` para aplicar, la función se invocará una vez por fila en su lugar:

```

En [195]: frame.apply(f, eje='columnas')
Fuera[195]:
Utah      0.998382
Ohio      2.521511
Texas 0.676115 Oregón
2.542656 dtype: float64

```

Muchas de las estadísticas de matriz más comunes (como la suma y la media) son métodos de DataFrame, por lo que no es necesario usar `apply` .

La función pasada para aplicar no necesita devolver un valor escalar; también puede devolver una Serie con múltiples valores:

```

En [196]: def f(x): return
.....:         pd.Series([x.min(), x.max()], index=['min', 'max'])

En [197]: cuadro.aplicar(f)

```

```
Fuera[197]:
           b           d           mi
min -0.555730 0.281746 -1.296221
máx. 1.246435 1.965781 1.393406
```

También se pueden usar funciones de Python basadas en elementos. Suponga que desea calcular una cadena formateada a partir de cada valor de punto flotante en el marco. Puedes hacer esto con aplicar mapa:

```
En [198]: formato = lambda x: '%.2f' % x

En [199]: frame.applymap(formato)
Fuera[199]:
           ser -0.20 d 0.48 -0.52

Utah
Ohio   -0,56 1,97 1,39
Texas   0,09 0,28 0,77
Oregón  1,25 1,01 -1,30
```

El motivo del nombre applymap es que Series tiene un método de mapa para aplicar una función por elementos:

```
En [200]: marco['e'].mapa(formato)
Fuera[200]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregón    -1.30
Nombre: e, dtype: objeto
```

Clasificación y clasificación La

clasificación de un conjunto de datos por algún criterio es otra operación incorporada importante. Para ordenar lexicográficamente por índice de fila o columna, use el método `sort_index`, que devuelve un nuevo objeto ordenado:

```
En [201]: obj = pd.Series(rango(4), índice=['d', 'a', 'b', 'c'])

En [202]: obj.sort_index()
Salida[202]:
a      1
b      2
c      3
d      0
tipo: int64
```

Con un DataFrame, puede ordenar por índice en cualquiera de los ejes:

```
En [203]: cuadro = pd.DataFrame(np.arange(8).reshape((2, 4)),
.....:                          índice=['tres', 'uno'], columnas=['d',
.....:                          'a', 'b', 'c'])

En [204]: frame.sort_index()
```

```

Fuera[204]:
           dabc
uno      4 5 6 7
tres 0 1 2 3

En [205]: frame.sort_index(eje=1)
Fuera[205]:
           a B C D
tres 1 2 3 0
uno   5 6 7 4

```

Los datos se ordenan en orden ascendente de forma predeterminada, pero se pueden ordenar en orden descendente ordenar, también:

```

En [206]: frame.sort_index(eje=1, ascendente=Falso)
Fuera[206]:
           dcba
tres 0 3 2 1
uno   4 7 6 5

```

Para ordenar una Serie por sus valores, use su método `sort_values` :

```

En [207]: obj = pd.Series([4, 7, -3, 2])

En [208]: obj.sort_values()
Fuera[208]:
2 -3
3    2
0    4
1    7
tipo: int64

```

Los valores que faltan se ordenan al final de la serie de forma predeterminada:

```

En [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

En [210]: obj.sort_values()
Salida[210]:
4 -3.0
5 2.0
0    4.0
2    7.0
1    Yaya
3    Yaya
tipo: float64

```

Al ordenar un DataFrame, puede usar los datos en una o más columnas como orden llaves. Para hacerlo, pase uno o más nombres de columna a la opción `by` de `sort_values`:

```

En [211]: cuadro = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

En [212]: marco
Fuera[212]:
           abdominales

```

```
0 0 4
1 1 7
2 0 -3
3 1 2
```

```
En [213]: frame.sort_values(by='b')
```

```
Fuera[213]:
```

```
abdominales
2 0 -3
3 1 2
0 0 4
1 1 7
```

Para ordenar por varias columnas, pase una lista de nombres:

```
En [214]: frame.sort_values(by=['a', 'b'])
```

```
Fuera[214]:
```

```
abdominales
2 0 -3
0 0 4
3 1 2
1 1 7
```

Clasificación asigna rangos desde uno hasta el número de puntos de datos válidos en una matriz.

Los métodos de rango para Series y DataFrame son el lugar para buscar; por rango predeterminado rompe los empates asignando a cada grupo el rango medio:

```
En [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
```

```
En [216]: obj.rango()
```

```
Fuera[216]:
```

```
0 6.5
1 1.0
2    6.5
3    4.5
4    3.0
5 2.0
6 4.5
```

```
tipo: float64
```

Los rangos también se pueden asignar de acuerdo con el orden en que se observan en el datos:

```
En [217]: obj.rank(método='primero')
```

```
Fuera[217]:
```

```
0 6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
```

```
tipo: float64
```

Aquí, en lugar de usar el rango promedio 6.5 para las entradas 0 y 2, en su lugar tienen se ha establecido en 6 y 7 porque la etiqueta 0 precede a la etiqueta 2 en los datos.

También puede clasificar en orden descendente:

```
# Asigne valores de empate al rango máximo en el grupo
En [218]: obj.rank(ascending=False, method='max')
Fuera[218]:
0 2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6 4.0
tipo: float64
```

Consulte la [Tabla 5-6](#) para obtener una lista de los métodos de desempate disponibles.

DataFrame puede calcular rangos sobre las filas o las columnas:

```
En [219]: cuadro = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
.....:                          'c': [-2, 5, 8, -2.5]})

En [220]: marco
Fuera[220]:
   a    b    c
0 0 4,3 -2,0
1 1 7,0 5,0
2 0 -3,0 8,0
3 1 2,0 -2,5

En [221]: frame.rank(axis='columns')
Fuera[221]:
   a    b    c
0 2,0 3,0 1,0
1 1,0 3,0 2,0
2 2,0 1,0 3,0
3 2,0 3,0 1,0
```

Tabla 5-6. Métodos de desempate con rango

Método	Descripción
'promedio'	Predeterminado: asigna el rango promedio a cada entrada en el grupo igual
'min'	Use el rango mínimo para todo el grupo
'máximo'	Usar el rango máximo para todo el grupo
'primero'	Asigne rangos en el orden en que aparecen los valores en los datos
'denso'	Me gusta method='min', pero los rangos siempre aumentan en 1 entre grupos en lugar del número de iguales elementos en un grupo

Índices de ejes con etiquetas duplicadas

Hasta ahora, todos los ejemplos que hemos visto tenían etiquetas de eje únicas (índice valores). Si bien muchas funciones de pandas (como reindexar) requieren que las etiquetas sean únicas, no es obligatorio. Consideremos una serie pequeña con índices duplicados:

```
En [222]: obj = pd.Series(rango(5), índice=['a', 'a', 'b', 'b', 'c'])
```

```
En [223]: objeto
```

```
Fuera[223]:
```

```
a      0
a      1
...     2
...     3
c      4
tipo: int64
```

La propiedad `is_unique` del índice puede decirle si sus etiquetas son únicas o no:

```
En [224]: obj.index.is_unique
```

```
Salida[224]: Falso
```

La selección de datos es una de las cosas principales que se comporta de manera diferente con los duplicados.

Indexar una etiqueta con múltiples entradas devuelve una Serie, mientras que las entradas individuales devuelven un valor escalar:

```
En [225]: objeto['a']
```

```
Fuera[225]:
```

```
a      0
a      1
tipo: int64
```

```
En [226]: objeto['c']
```

```
Salida[226]: 4
```

Esto puede hacer que su código sea más complicado, ya que el tipo de salida de la indexación puede variar en función de si una etiqueta se repite o no.

La misma lógica se extiende a la indexación de filas en un DataFrame:

```
En [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
```

```
En [228]: d.f.
```

```
Fuera[228]:
```

```
      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
```

```
En [229]: df.loc['b']
```

```
Fuera[229]:
```

```
      0      1      2
```

```
b 1.669025 -0.438570 -0.539741 b
0.476985 3.248944 -1.021228
```

5.3 Resumen y cálculo de estadísticas descriptivas

Los objetos pandas están equipados con un conjunto de métodos matemáticos y estadísticos comunes.

La mayoría de estos entran en la categoría de reducciones o estadísticas de resumen, métodos que extraen un valor único (como la suma o la media) de una Serie o una Serie de valores de las filas o columnas de un DataFrame. En comparación con los métodos similares que se encuentran en las matrices NumPy, tienen un manejo integrado para los datos faltantes. Considere un pequeño DataFrame:

```
En [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
.....:                      [np.nan, np.nan], [0.75, -1.3]],
.....:                      índice=['a', 'b', 'c', 'd'], columnas=['uno',
.....:                      'dos'])

En [231]: df
Fuera[231]:
      uno dos
a  1.40 NaN b
7,10 -4,5
c  NaN NaN
d  0,75 -1,3
```

Llamar al método `sum` de DataFrame devuelve una serie que contiene sumas de columnas:

```
En [232]: df.sum()
Fuera[232]:
      uno      dos
dtype: float64
```

Pasando `axis='columns'` o `axis=1` sumas a través de las columnas en su lugar:

```
En [233]: df.sum(axis='columns')
Salida[233]:
      1,40
      2,60
c
NaN d -0.55
dtype: float64
```

Los valores NA se excluyen a menos que todo el segmento (fila o columna en este caso) sea NA.

Esto se puede desactivar con la opción `skipna` :

```
En [234]: df.mean(axis='columns', skipna=False)
Salida[234]:
      a      b
b      NaN  1.300
c      Yaya
```

```
d -0.275 dtipo:
float64
```

Consulte la [Tabla 5-7](#) para obtener una lista de opciones comunes para cada método de reducción.

Tabla 5-7. Opciones para métodos de reducción

Método	Descripción
eje	Eje a reducir; 0 para filas de DataFrame y 1 para columnas
skipna	Excluye valores faltantes; Verdadero por nivel
predeterminado	Reducir agrupados por nivel si el eje está indexado jerárquicamente (MultiIndex)

Algunos métodos, como `idxmin` e `idxmax`, devuelven estadísticas indirectas como el valor del índice donde se alcanzan los valores mínimo o máximo:

```
En [235]: df.idxmax()
Salida[235]:
uno      b
dos      d
dtype: objeto
```

Otros métodos son las acumulaciones:

```
En [236]: df.cumsum()
Fuera[236]:
      uno dos
a 1,40 NaN b
8,50 -4,5
c NaN NaN d
9,25 -5,8
```

Otro tipo de método no es ni una reducción ni una acumulación. describe es uno de esos ejemplos, que produce múltiples estadísticas de resumen de una sola vez:

```
En [237]: df.describe()
Fuera[237]:
      uno      dos
cuenta 3.000000 2.000000
media 3,083333 -2,900000 estándar
3,493685 2,262742 min
      0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
máximo 7.100000 -1.300000
```

En datos no numéricos, describe produce estadísticas de resumen alternativas:

```
En [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

En [239]: obj.describe()
Fuera[239]:
cuenta      4
unique      4
```

único tipo 3
de a
8
frecuencia superior: objeto

Consulte la [Tabla 5-8](#) para obtener una lista completa de estadísticas de resumen y métodos relacionados.

Tabla 5-8. Estadísticas descriptivas y resumidas

Método	Descripción
contar	Número de valores no NA
describir	Calcule un conjunto de estadísticas de resumen para Series o cada columna de DataFrame
min, max	Calcular valores mínimos y máximos
argmin, argmax	Calcular ubicaciones de índice (enteros) en las que se obtuvo el valor mínimo o máximo, respectivamente
idxmin, idxmax	Calcular las etiquetas de índice en las que se obtuvo el valor mínimo o máximo, respectivamente
cuantil	Calcule el cuantil de muestra que va de 0 a 1
suma	Suma de valores
significar	Media de valores
mediana	Mediana aritmética (50% cuantil) de valores
enojado	Desviación absoluta media del valor medio
pinchar	Producto de todos los valores
variable	Ejemplo de varianza de valores
estándar	Ejemplo de desviación estándar de valores
sesgar	Sesgo muestral (tercer momento) de los valores
kurt	Ejemplo de curtosis (cuarto momento) de valores
cumsum	Suma acumulada de valores
cummin, cummax	Mínimo o máximo acumulativo de valores, respectivamente
Cumprod	Producto acumulativo de valores
diferencia	Calcular la primera diferencia aritmética (útil para series de tiempo)
cambio_pct	Calcular cambios porcentuales

Correlación y Covarianza

Algunas estadísticas de resumen, como la correlación y la covarianza, se calculan a partir de pares de argumentos. Consideremos algunos DataFrames de precios de acciones y volúmenes obtenidos de Yahoo! Finanzas utilizando el paquete adicional pandas-datareader . si no lo haces tenerlo ya instalado, se puede obtener a través de conda o pip:

```
conda instalar pandas-datareader
```

Utilizo el módulo pandas_datareader para descargar algunos datos de algunas cotizaciones bursátiles:

```
importar pandas_datareader.data como web
all_data = {marca: web.get_data_yahoo(marca)
            para ticker en ['AAPL', 'IBM', 'MSFT', 'GOOG']}
```

```

precio = pd.DataFrame({ticker: data[Adj Close]
                        para teletipo, datos en all_data.items()}) volumen =
pd.DataFrame({ticker: data[Volume] para teletipo, datos en
              all_data.items()})

```



Es posible en el momento en que estés leyendo esto que Yahoo! Finance ya no existe desde que Yahoo! fue adquirida por Verizon en 2017. Consulte la documentación en línea de pandas-datareader para obtener la última funcionalidad.

Ahora calculo los cambios porcentuales de los precios, una operación de series de tiempo que se explorará más adelante en el **Capítulo 11**:

```

En [242]: devuelve = precio.pct_change()

En [243]: devuelve.tail()
Fuera[243]:

```

	AAPL	GOOG	IBM	MSFT
Fecha				
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` de Series calcula la correlación de los valores superpuestos, no NA, alineados por índice en dos Series. De manera relacionada, `cov` calcula la covarianza:

```

En [244]: devuelve["MSFT"].corr(devuelve["IBM"])
Salida[244]: 0.49976361144151144

En [245]: devuelve["MSFT"].cov(devuelve["IBM"])
Salida[245]: 8.8706554797035462e-05

```

Dado que MSFT es un atributo válido de Python, también podemos seleccionar estas columnas usando una sintaxis más concisa:

```

En [246]: devoluciones.MSFT.corr(devoluciones.IBM)
Salida[246]: 0.49976361144151144

```

Los métodos `corr` y `cov` de DataFrame, por otro lado, devuelven una matriz de correlación o covarianza completa como un DataFrame, respectivamente:

```

En [247]: devuelve.corr()
Fuera[247]:

```

	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

```

En [248]: devuelve.cov()
Fuera[248]:
      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

```

Con el método `corrwith` de `DataFrame`, puede calcular las correlaciones por pares entre las columnas o filas de un `DataFrame` con otra Serie o `DataFrame`. Pasar una Serie devuelve una Serie con el valor de correlación calculado para cada columna:

```

En [249]: devuelve.corrwith(devuelve.IBM)
Salida[249]:
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. Aquí calculo las correlaciones de los cambios porcentuales con el volumen:

```

En [250]: devuelve.corrwith(volumen)
Salida[250]:
AAPL -0.075565
GOOG -0.007067
IBM -0.204849
MSFT -0.092950
tipo: float64

```

Pasar `axis='columns'` hace las cosas fila por fila en su lugar. En todos los casos, los puntos de datos se alinean por etiqueta antes de calcular la correlación.

Valores únicos, recuentos de valor y membresía

Otra clase de métodos relacionados extrae información sobre los valores contenidos en una Serie unidimensional. Para ilustrar esto, considere este ejemplo:

```

En [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'b', 'b', 'c', 'c'])

```

La primera función es `unique`, lo que le brinda una matriz de valores únicos en una Serie:

```

En [252]: únicos = obj.unique()

En [253]: únicos
Salida[253]: matriz(['c', 'a', 'd', 'b'], dtype=objeto)

```

Los valores únicos no se devuelven necesariamente en orden ordenado, pero se pueden ordenar después del hecho si es necesario (`uniques.sort()`). De manera relacionada, `value_counts` calcula una serie que contiene frecuencias de valor:

```

En [254]: obj.value_counts()
Fuera[254]:
c      3
      3
      2
      1
abd tipo: int64

```

La serie se ordena por valor en orden descendente para mayor comodidad. `value_counts` es también disponible como un método de pandas de nivel superior que se puede usar con cualquier matriz o secuencia:

```

En [255]: pd.value_counts(obj.values, sort=False)
Fuera[255]:
a      3
b      2
      3

disco 1
tipo: int64

```

`isin` realiza una verificación de membresía de conjunto vectorizado y puede ser útil para filtrar un conjunto de datos hasta un subconjunto de valores en una serie o columna en un marco de datos:

```

En [256]: objeto
Fuera[256]:
0      c
1      a
2      d
3      a
4      a
5      b
6      b
7      c
8      c
tipo: objeto

```

```

En [257]: máscara = obj.isin(['b', 'c'])

```

```

En [258]: máscara
Fuera[258]:
0      Verdadero
1      FALSO
2      FALSO
3      FALSO
4      FALSO
5      Verdadero
6      Verdadero
7      Verdadero
8      Verdadero
dtipo: booleano

```

```

En [259]: obj[máscara]
Fuera[259]:

```

```
0    c
5    b
6    b
    c
    c
7 8 dtipo: objeto
```

Relacionado con `isin` está el método `Index.get_indexer`, que le brinda una matriz de índice de una matriz de valores posiblemente no distintos a otra matriz de valores distintos:

```
En [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

En [261]: valores_únicos = pd.Series(['c', 'b', 'a'])

En [262]: pd.Index(unique_vals).get_indexer(to_match)
Salida[262]: matriz([0, 2, 1, 1, 0, 2])
```

Consulte la [Tabla 5-9](#) para obtener una referencia sobre estos métodos.

Tabla 5-9. Únicos, recuentos de valores y métodos de membresía establecidos

Método	Descripción
<code>es en</code>	Calcule la matriz booleana que indica si cada valor de Serie está contenido en la secuencia pasada de valores
<code>fósforo</code>	Calcule índices enteros para cada valor en una matriz en otra matriz de valores distintos; útil para los datos operaciones de alineación y tipo unión
<code>unique</code>	Calcule una matriz de valores únicos en una serie, devueltos en el orden observado
<code>value_counts</code>	Devuelve una serie que contiene valores únicos como su índice y frecuencias como sus valores, conteo ordenado en orden descendiente

En algunos casos, es posible que desee calcular un histograma en varias columnas relacionadas en una trama de datos. Aquí hay un ejemplo:

```
En [263]: datos = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....:                        'Qu2': [2, 3, 1, 2, 3],
.....:                        'Qu3': [1, 5, 2, 4, 4]})

En [264]: datos
Fuera[264]:
   Qu1  Qu2  Qu3
0     1     2     1
1     3     3     5
2     4     1     2
3     3     2     4
4     4     3     4
```

Pasar `pandas.value_counts` a la función de aplicación de este `DataFrame` da:

```
En [265]: resultado = data.apply(pd.value_counts).fillna(0)

En [266]: resultado
Fuera[266]:
```

```

      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

```

Aquí, las etiquetas de las filas en el resultado son los distintos valores que ocurren en todas las columnas. Los valores son los recuentos respectivos de estos valores en cada columna.

5.4 Conclusión

En el próximo capítulo, discutiremos las herramientas para leer (o cargar) y escribir conjuntos de datos con pandas. Después de eso, profundizaremos en las herramientas de limpieza, disputa, análisis y visualización de datos usando pandas.

Carga de datos, almacenamiento y formatos de archivo

Acceder a los datos es un primer paso necesario para utilizar la mayoría de las herramientas de este libro. Soy se centrará en la entrada y salida de datos usando pandas, aunque hay numerosos Otras herramientas en otras bibliotecas para ayudar con la lectura y escritura de datos en varios formatos.

La entrada y la salida generalmente se dividen en algunas categorías principales: lectura de archivos de texto y otros formatos en disco más eficientes, cargando datos desde bases de datos e interactuando con redes fuentes de trabajo como las API web.

6.1 Leer y escribir datos en formato de texto

pandas presenta una serie de funciones para leer datos tabulares como un DataFrame objeto. La Tabla 6-1 resume algunos de ellos, aunque read_csv y read_table son probablemente los que más usará.

Tabla 6-1. Funciones de análisis en pandas

Función	Descripción
read_csv	Cargue datos delimitados desde un archivo, URL u objeto similar a un archivo; usar coma como delimitador predeterminado
read_table read_fwf	Cargue datos delimitados desde un archivo, URL u objeto similar a un archivo; use la pestaña ('\t') como delimitador predeterminado
read_clipboard	Leer datos en formato de columna de ancho fijo (es decir, sin delimitadores)
Versión de read_table que lee datos del portapapeles; útil para convertir tablas de la web	
	paginas
read_excel read_hdf	Leer datos tabulares de un archivo Excel XLS o XLSX
read_html	Leer archivos HDF5 escritos por pandas
read_json	Leer todas las tablas que se encuentran en el documento HTML dado
read_msgpack	Leer datos de una representación de cadena JSON (Notación de objetos de JavaScript)
Leer datos de pandas codificados usando el formato binario MessagePack	
leer_pickle	Lea un objeto arbitrario almacenado en formato pickle de Python

Función	Descripción
<code>leer_sas</code>	Lea un conjunto de datos SAS almacenado en uno de los formatos de almacenamiento personalizados del sistema SAS
<code>leer_sql</code>	Lea los resultados de una consulta SQL (usando SQLAlchemy) como un marco de datos de pandas
<code>read_stata</code>	Leer un conjunto de datos del formato de archivo Stata
<code>read_feather</code>	Leer el formato de archivo binario Feather

Daré una descripción general de la mecánica de estas funciones, que están destinadas a convertir datos de texto en un DataFrame. Los argumentos opcionales para estas funciones pueden caer en algunas categorías:

Indexación

Puede tratar una o más columnas como el DataFrame devuelto y si obtener los nombres de las columnas del archivo, el usuario o no obtenerlos.

Inferencia de tipos y conversión de datos

Esto incluye las conversiones de valores definidas por el usuario y la lista personalizada de marcadores de valores perdidos.

Análisis de fecha y

hora Incluye capacidad de combinación, incluida la combinación de información de fecha y hora distribuida en varias columnas en una sola columna en el resultado.

Iteración

Soporte para iterar sobre fragmentos de archivos muy grandes.

Problemas de datos

sucios Omitir filas o un pie de página, comentarios u otras cosas menores como datos numéricos con miles separados por comas.

Debido a lo desordenados que pueden ser los datos en el mundo real, algunas de las funciones de carga de datos (especialmente `read_csv`) se han vuelto muy complejas en sus opciones con el tiempo. Es normal sentirse abrumado por la cantidad de parámetros diferentes (`read_csv` tiene más de 50 al momento de escribir este artículo). La documentación de pandas en línea tiene muchos ejemplos sobre cómo funciona cada uno de ellos, por lo que si tiene dificultades para leer un archivo en particular, puede haber un ejemplo lo suficientemente similar para ayudarlo a encontrar los parámetros correctos.

Algunas de estas funciones, como `pandas.read_csv`, realizan inferencias de tipos, porque los tipos de datos de las columnas no forman parte del formato de datos. Eso significa que no necesariamente tiene que especificar qué columnas son numéricas, enteras, booleanas o de cadena. Otros formatos de datos, como HDF5, Feather y msgpack, tienen los tipos de datos almacenados en el formato.

El manejo de fechas y otros tipos personalizados puede requerir un esfuerzo adicional. Comencemos con un pequeño archivo de texto separado por comas (CSV):

```
En [8]: !cat ejemplos/ex1.csv
a,b,c,d,mensaje
1,2,3,4,hola
```

```
5,6,7,8,mundo
9,10,11,12,foo
```



Aquí usé el comando Unix cat shell para imprimir el contenido sin procesar del archivo a la pantalla. Si está en Windows, puede usar el tipo en lugar de gato para lograr el mismo efecto.

Dado que está delimitado por comas, podemos usar `read_csv` para leerlo en un DataFrame:

```
En [9]: df = pd.read_csv('ejemplos/ex1.csv')
```

```
En [10]: d.f.
```

```
Fuera[10]:
```

```
   a  b  c  mensaje
0  1   2  3    4 hola
15   6  7    8 mundo
2  9 10 11 12      Foo
```

También podríamos haber usado `read_table` y especificado el delimitador:

```
En [11]: pd.read_table('ejemplos/ex1.csv', sep=',')
```

```
Fuera[11]:
```

```
   a  b  c  mensaje
0  1   2  3    4 hola
1  5  6  2  9    7    8 mundo
10 11 12      Foo
```

Un archivo no siempre tendrá una fila de encabezado. Considere este archivo:

```
En [12]: !cat ejemplos/ex2.csv
```

```
1,2,3,4, hola
5,6,7,8,mundo
9,10,11,12,foo
```

Para leer este archivo, tiene un par de opciones. Puede permitir que los pandas asignen valores predeterminados nombres de columna, o puede especificar nombres usted mismo:

```
En [13]: pd.read_csv('ejemplos/ex2.csv', encabezado=Ninguno)
```

```
Fuera[13]:
```

```
   0 1      2  3 4
0  1   2  3    4 hola
15   6  7    8 mundo
2  9 10 11 12      Foo
```

```
En [14]: pd.read_csv('ejemplos/ex2.csv', nombres=['a', 'b', 'c', 'd', 'mensaje'])
```

```
Fuera[14]:
```

```
   a  b  c  mensaje
0  1   2  3    4 hola
1  5   6  7    8 mundo
2  9 10 11 12      Foo
```

Suponga que desea que la columna del mensaje sea el índice del DataFrame devuelto. Puede indicar que desea la columna en el índice 4 o llamarla 'mensaje' usando el argumento `index_col` :

```
En [15]: nombres = ['a', 'b', 'c', 'd', 'mensaje']

En [16]: pd.read_csv('ejemplos/ex2.csv', nombres=nombres, index_col='mensaje')
Fuera[16]:
```

	a	b	c	d
mensaje				
hola	1	2	3	4
mundo	5	6	7	8
foo	9	10	11	12

En el caso de que desee formar un índice jerárquico a partir de varias columnas, pase un lista de números de columna o nombres:

```
En [17]: !cat ejemplos/csv_mindex.csv
clave1,clave2,valor1,valor2
uno,a,1,2
uno,b,3,4
uno,c,5,6
uno,d,7,8
dos,a,9,10
dos,b,11,12
dos,c,13,14
dos, d, 15, 16

En [18]: analizado = pd.read_csv('ejemplos/csv_mindex.csv',
.....:                             index_col=['clave1', 'clave2'])
```

```
En [19]: analizado
Fuera[19]:
```

		valor1	valor2
clave1	clave2		
uno	a	1	2
	b	3	4
		5	6
	cd	7	8
dos	abdominales	9	10
		11	12
	C	13	14
	d	15	decada

En algunos casos, una tabla puede no tener un delimitador fijo, usando espacios en blanco o algún otro patrón para separar campos. Considere un archivo de texto que se ve así:

```
En [20]: list(open('ejemplos/ex3.txt'))
Fuera[20]:
```

```
[
    '          A          B          C\n',
    'aaa -0.264438 -1.026059 -0.619500\n',
    'bbb 0.927272 0.302904 -0.032399\n',
```

```
'ccc -0.264273 -0.386314 -0.217601\n',
'ddd -0.871858 -0.348382 1.100491\n']
```

Si bien podría hacer algunos cambios a mano, los campos aquí están separados por una variedad. cantidad capaz de espacios en blanco. En estos casos, puede pasar una expresión regular como delimitador para `read_table`. Esto se puede expresar mediante la expresión regular `\s+`, por lo que tener entonces:

```
En [21]: resultado = pd.read_table('ejemplos/ex3.txt', sep='\s+')
```

```
En [22]: resultado
```

```
Fuera[22]:
```

```
      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
```

Debido a que había un nombre de columna menos que el número de filas de datos,

`read_table` infiere que la primera columna debe ser el índice de DataFrame en este caso especial.

Las funciones del analizador tienen muchos argumentos adicionales para ayudarlo a manejar la amplia variedad de formatos de archivo de excepción que ocurren (vea una lista parcial en [la Tabla 6-2](#)). Para ejemplo, puede omitir la primera, tercera y cuarta fila de un archivo con `skiprows`:

```
En [23]: !cat ejemplos/ex4.csv
# ¡ey!
a,b,c,d,mensaje
# solo quería ponerte las cosas más difíciles
# ¿Quién lee los archivos CSV con las computadoras?
1,2,3,4, hola
5,6,7,8,mundo
9,10,11,12,foo
En [24]: pd.read_csv('ejemplos/ex4.csv', skiprows=[0, 2, 3])
Fuera[24]:
```

	a	b	c	mensaje
0	1	2	3	4 hola
15	6	7	8	mundo
29	10	11	12	Foo

El manejo de los valores faltantes es una parte importante y frecuentemente matizada de los parámetros del archivo. proceso de ing Los datos que faltan generalmente no están presentes (cadena vacía) o están marcados por algún valor centinela. Por defecto, pandas usa un conjunto de centinelas comunes, como NA y NULL:

```
En [25]: !cat ejemplos/ex5.csv
algo,a,b,c,d,mensaje
uno, 1, 2, 3, 4, NA
dos, 5, 6, 8, mundo
tres,9,10,11,12,foo
En [26]: resultado = pd.read_csv('ejemplos/ex5.csv')
```

```
En [27]: resultado
Fuera[27]:
    algo abd mensaje          C
0      uno 1 2 3.0 4 NaN
1      dos 5 6 NaN 8 mundo
2      tres 9 10 11.0 12 pies

En [28]: pd.isnull(resultado)
Fuera[28]:
    algo d mensaje      a      b      c
0      Falso Falso Falso Falso Falso Verdadero
1      Falso Falso Falso Verdadero Falso Falso
2      Falso Falso Falso Falso Falso          FALSO
```

La opción `na_values` puede tomar una lista o un conjunto de cadenas para considerar que faltan valores:

```
En [29]: resultado = pd.read_csv('ejemplos/ex5.csv', na_values=['NULL'])

En [30]: resultado
Fuera[30]:
    algo abd mensaje          C
    uno 1 2 3.0 4 NaN
0 1      dos 5 6 NaN 8 mundo
2      tres 9 10 11.0 12 pies
```

Se pueden especificar diferentes centinelas de NA para cada columna en un dictado:

```
En [31]: centinelas = {'mensaje': ['foo', 'NA'], 'algo': ['dos']}

En [32]: pd.read_csv('ejemplos/ex5.csv', na_values=centinelas)
Fuera[32]:
    algo abd mensaje          C
0      uno 1 2 3.0 4 NaN
1      NaN 5 6 NaN 8 mundo
2      tres 9 10 11.0 12          Yaya
```

La **Tabla 6-2** enumera algunas opciones de uso frecuente en `pandas.read_csv` y `pd.read_table`.

Tabla 6-2. Algunos argumentos de la función `read_csv/read_table`

Argumento	Descripción
<code>ruta</code>	Cadena que indica la ubicación del sistema de archivos, URL u objeto similar a un archivo
<code>separador</code> o <code>delimitador</code>	Secuencia de caracteres o expresión regular que se usará para dividir campos en cada fila
<code>encabezamiento</code>	Número de fila para usar como nombres de columna; el valor predeterminado es 0 (primera fila), pero debe ser Ninguno si no hay fila de encabezado
<code>index_col</code>	Números de columna o nombres para usar como índice de fila en el resultado; puede ser un solo nombre/número o un lista de ellos para un índice jerárquico
<code>nombres</code>	Lista de nombres de columna para resultado, combinar con <code>encabezado</code> = Ninguno

Argumento	Descripción
salteadores	Número de filas al principio del archivo para ignorar o lista de números de fila (a partir de 0) para omitir.
valores_na	Secuencia de valores para reemplazar con NA.
comentario	Carácter(es) para dividir los comentarios al final de las líneas.
parse_dates	Intente analizar los datos hasta la fecha y hora; Falso por defecto. Si es Verdadero, intentará analizar todas las columnas. De lo contrario, puede especificar una lista de números de columna o nombre para analizar. Si el elemento de la lista es una tupla o una lista, combine varias columnas y analice hasta la fecha (por ejemplo, si la fecha/hora se divide en dos columnas).
keep_date_col	Si une columnas para analizar la fecha, mantenga las columnas unidas; Falso por defecto.
convertidores	Dict que contiene el número de columna del mapeo de nombres a funciones (por ejemplo, {'foo': f} aplicaría el función f a todos los valores en la columna 'foo').
primer dia	Al analizar fechas potencialmente ambiguas, trátelas como formato internacional (p. ej., 7/6/2012 -> 7 de junio de 2012); Falso por defecto.
analizador_de_fechas	Función a usar para analizar fechas.
filas	Número de filas para leer desde el principio del archivo.
iterador	Devuelve un objeto TextParser para leer el archivo por partes.
tamaño de porción	Para la iteración, el tamaño de los fragmentos de archivo.
saltar_pie de página	Número de líneas a ignorar al final del archivo.
verboso	Imprima diversa información de salida del analizador, como la cantidad de valores faltantes colocados en no numéricos columnas
codificación	Codificación de texto para Unicode (p. ej., 'utf-8' para texto codificado en UTF-8).
estrujar	Si los datos analizados solo contienen una columna, devuelva una Serie.
miles	Separador de miles (p. ej., ',' o '.').

Lectura de archivos de texto en partes

Al procesar archivos muy grandes o descubrir el conjunto correcto de argumentos para procesar correctamente un archivo grande, es posible que solo desee leer una pequeña parte de un archivo o iterar a través de trozos más pequeños del archivo.

Antes de mirar un archivo grande, hacemos que la configuración de visualización de pandas sea más compacta:

```
En [33]: pd.options.display.max_rows = 10
```

Ahora tenemos:

```
En [34]: resultado = pd.read_csv('ejemplos/ex6.csv')
```

```
En [35]: resultado
Fuera[35]:
```

	uno	dos	tres	cuatro	teclas
	0.467976	-0.038649	-0.295344	-1.824726	L
0 1	-0,358893	1,404453	0,704965	-0,200638	B
2	-0,501840	0,659254	-0,421691	-0,057688	G
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q
...	...	...	...	...	...
9995	2.311896	-0.417070	-1.409599	-0.515821	L

```

9996 -0.479893 -0.650419 0.745152 -0.646038 E
9997 0.523331 0.787112 0.486066 1.093156 K
9998 -0.362559 0.598894 -1.843201 0.887292 G
9999 -0.096376 -1.012999 -0.657431 -0.573315 0
[10000 filas x 5 columnas]

```

Si solo desea leer una pequeña cantidad de filas (evitando leer todo el archivo),
especifica eso con `nrows`:

```

En [36]: pd.read_csv('ejemplos/ex6.csv', nrows=5)
Fuera[36]:
      uno      dos      tres      cuatro teclas
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 en partes, especifique un tamaño de fragmento como un número de filas:

```

En [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

En [38]: fragmentador
Salida[38]: <pandas.io.parsers.TextFileReader en 0x7f6b1e2672e8>

```

El objeto `TextParser` devuelto por `read_csv` le permite iterar sobre las partes de
el archivo de acuerdo con el tamaño de fragmento. Por ejemplo, podemos iterar sobre `ex6.csv`, agregar
controlar el valor cuenta en la columna 'clave' de la siguiente manera:

```

chunker = pd.read_csv('ejemplos/ex6.csv', tamaño de fragmento=1000)

tot = pd.Series({})
por pieza en chunker:
    tot = tot.add(pieza['clave'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)

```

Tenemos entonces:

```

En [40]: total[:10]
Fuera[40]:
mi      368.0
X       364.0
L       346.0
O       343.0
q       340.0
—       338.0
j       337.0
F       335.0
k       334.0
H       330.0
tipo: float64

```

TextParser también está equipado con un método `get_chunk` que le permite leer piezas de un tamaño arbitrario.

Escritura de datos en formato de texto Los

datos también se pueden exportar a un formato delimitado. Consideremos uno de los archivos CSV leídos antes:

```
En [41]: datos = pd.read_csv('ejemplos/ex5.csv')

In [42]: data
Out[42]:
      algo abd mensaje uno 1 2 3 4 NaN dos 5 6
0      NaN 8 mundo tres 9 10 11.0 12 foo
1
2
```

Usando el método `to_csv` de `DataFrame`, podemos escribir los datos en un archivo separado por comas:

```
En [43]: data.to_csv('ejemplos/out.csv')

En [44]: !cat ejemplos/
out.csv ,algo,a,b,c,d,mensaje
0,uno,1,2,3.0,4,
1,dos,5,6,,8,mundo 2,
tres,9,10,11.0,12,foo
```

Por supuesto, se pueden usar otros delimitadores (escribiendo en `sys.stdout` para que imprima el resultado del texto en la consola):

```
En [45]: importar sistema

En [46]: data.to_csv(sys.stdout, sep='|') |algo|a|b|c|d|
mensaje 0|uno|1|2|3.0|4| 1|dos|5|6|
8|mundo 2|tres|9|10|
11.0|12|foo
```

Los valores que faltan aparecen como cadenas vacías en la salida. Es posible que desee indicarlos mediante algún otro valor centinela:

```
En [47]: data.to_csv(sys.stdout,
na_rep='NULL') ,algo,a,b,c,d,mensaje
0,uno,1,2,3.0,4,NULL
1,dos,5,6,NULO,8,mundo
2,tres,9,10,11.0,12,foo
```

Sin otras opciones especificadas, se escriben las etiquetas de fila y columna. Ambos se pueden deshabilitar:

```
En [48]: 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 puede escribir solo un subconjunto de las columnas y en el orden que elija:

```
En [49]: data.to_csv(sys.stdout, índice=False, columnas=['a', 'b', 'c']) a,b,c 1,2,3.0 5,6,
9,10
,11.0
```

Series también tiene un método `to_csv` :

```
En [50]: fechas = pd.date_range('1/1/2000', periodos=7)
```

```
En [51]: ts = pd.Series(np.arange(7), index=fechas)
```

```
En [52]: ts.to_csv('ejemplos/tseries.csv')
```

```
En [53]: !cat ejemplos/tseries.csv
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

Trabajar con formatos delimitados Es posible cargar

la mayoría de las formas de datos tabulares desde el disco usando funciones como `pd.read_table`. En algunos casos, sin embargo, puede ser necesario algún procesamiento manual.

No es raro recibir un archivo con una o más líneas mal formadas que tropiezan con `read_table`. Para ilustrar las herramientas básicas, considere un pequeño archivo CSV:

```
En [54]: !cat ejemplos/ex7.csv
"a","b","c"
1","2","3"
"1","2","3"
```

Para cualquier archivo con un delimitador de un solo carácter, puede usar el módulo `csv` incorporado de Python .

Para usarlo, pase cualquier archivo abierto u objeto similar a un archivo a `csv.reader`:

```
importar csv
f = open('ejemplos/ex7.csv')

lector = csv.lector(f)
```

Iterar a través del lector como un archivo produce tuplas de valores con cualquier carácter de comillas eliminado:

```

En [56]: para línea en lector:
.....:         print(línea)
['a', 'b', 'c'] ['1', '2',
'3'] ['1', '2', '3']

```

A partir de ahí, depende de usted hacer las gestiones necesarias para poner los datos en la forma en que los necesita. Vamos a tomar esto paso a paso. Primero, leemos el archivo en una lista de líneas:

```

En [57]: with open('examples/ex7.csv') as f: líneas =
.....:         list(csv.reader(f))

```

Luego, dividimos las líneas en la línea de encabezado y las líneas de datos:

```

En [58]: encabezado, valores = líneas[0], líneas[1:]

```

Luego, podemos crear un diccionario de columnas de datos usando una comprensión de diccionario y la expresión `zip(*valores)`, que transpone filas a columnas:

```

En [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}

```

```

En [60]: data_dict
Salida[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}

```

Los archivos CSV vienen en muchos sabores diferentes. Para definir un nuevo formato con un delimitador diferente, una convención de comillas de cadena o un terminador de línea, definimos una subclase simple de `csv.Dialect`:

```

class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimitador = ';'
    quotechar = '"'
    citando = csv.QUOTE_MINIMAL

lector = csv.reader(f, dialect=mi_dialect)

```

También podemos dar parámetros de dialecto CSV individuales como palabras clave a `csv.reader` sin tener que definir una subclase:

```

lector = csv.reader(f, delimitador=';')

```

Las opciones posibles (atributos de `csv.Dialect`) y lo que hacen se pueden encontrar en la [Tabla 6-3](#).

Tabla 6-3. Opciones de dialecto CSV

Argumento	Descripción
delimitador	Cadena de un carácter para separar campos; el valor predeterminado es <code>','</code> .
terminador de línea	Terminador de línea para escribir; por defecto es <code>'\n'</code> . Reader ignora esto y reconoce los terminadores de línea multiplataforma.
cotizar	Carácter de comillas para campos con caracteres especiales (como un delimitador); el valor predeterminado es <code>'"'</code> .

Argumento	Descripción
citando	Convención de citas. Las opciones incluyen csv.QUOTE_ALL (comillar todos los campos), csv.QUOTE_MINI MAL (solo campos con caracteres especiales como el delimitador), csv.QUOTE_NONNUMERIC y csv.QUOTE_NONE (sin comillas). Consulte la documentación de Python para obtener detalles completos. El valor predeterminado es QUOTE_MINIMAL.
skipinitialspace	Ignora los espacios en blanco después de cada delimitador; el valor predeterminado es
Falso. comillas dobles	Cómo manejar el carácter de comillas dentro de un campo; si es True, se duplica (consulte la documentación en línea para conocer todos los detalles y el comportamiento).
escapechar	Cadena para escapar del delimitador si las comillas se establecen en csv.QUOTE_NONE; deshabilitado por defecto.



Para archivos con delimitadores de varios caracteres fijos o más complicados, no podrá utilizar el módulo csv . En esos casos, deberá realizar la división de líneas y otras tareas de limpieza mediante el método de división de cadenas o el método de expresión regular re.split.

Para escribir archivos delimitados manualmente, puede usar csv.writer. Acepta un objeto de archivo abierto y grabable y las mismas opciones de formato y dialecto que csv.reader:

```
con open('misdatos.csv', 'w') como f:
    escritor = csv.escritor(f, dialecto=mi_dialecto)
    escritor.escritor(('uno', 'dos', 'tres'))
    escritor.escritor( ('1', '2', '3'))
    escritor.escritor(('4', '5', '6'))
    escritor.escritor(('7', '8', '9'))
```

Datos JSON

JSON (abreviatura de JavaScript Object Notation) se ha convertido en uno de los formatos estándar para enviar datos mediante solicitud HTTP entre navegadores web y otras aplicaciones. Es un formato de datos mucho más libre que un formulario de texto tabular como CSV. Aquí hay un ejemplo:

```
obj =
{
  "nombre": "Wes",
  "lugares_vivieron": ["Estados Unidos", "España", "Alemania"],
  "mascota":
  null,
  "hermanos": [
    {"nombre": "Scott", "edad": 30, "mascotas": ["Zeus", "Zuko"]},
    {"nombre": "Katie", "edad": 38,
     "mascotas": ["Sixes", "Stache", " Cisco"]}
  ]
}
```

JSON es un código de Python casi válido con la excepción de su valor nulo nulo y algunos otros matices (como no permitir las comas finales al final de las listas). Los tipos básicos son objetos (dicts), arreglos (listas), cadenas, números, valores booleanos y nulos. Todas las claves de un objeto deben ser cadenas. Hay varias bibliotecas de Python para leer

y escribir datos JSON. Usaré json aquí, ya que está integrado en la biblioteca estándar de Python. Para convertir una cadena JSON a forma de Python, use json.loads:

```
En [62]: importar json
```

```
En [63]: resultado = json.loads(obj)
```

```
En [64]: resultado
```

```
Fuera[64]:
```

```
{'nombre': 'Wes',  
  'mascota':  
    Ninguno, 'lugares_vividos': ['Estados Unidos', 'España', 'Alemania'],  
  'hermanos': [{'edad': 30, 'nombre': 'Scott', 'mascotas': ['Zeus', 'Zuko']}, {'edad': 38,  
    'nombre': 'Katie', 'mascotas': ['Sixes', 'Stache', 'Cisco']}]}
```

json.dumps, por otro lado, convierte un objeto de Python de nuevo a JSON:

```
En [65]: asjson = json.dumps(resultado)
```

Usted decide cómo convierte un objeto JSON o una lista de objetos en un DataFrame o alguna otra estructura de datos para el análisis. Convenientemente, puede pasar una lista de dictados (que anteriormente eran objetos JSON) al constructor de DataFrame y seleccionar un subconjunto de los campos de datos:

```
En [66]: hermanos = pd.DataFrame(resultado['hermanos'], columns=['nombre', 'edad'])
```

```
En [67]: hermanos
```

```
Fuera[67]:
```

```
      nombre Edad  
0 Scott    30  
1 Katie    38
```

pandas.read_json puede convertir automáticamente conjuntos de datos JSON en arreglos específicos en una serie o marco de datos. Por ejemplo:

```
En [68]: !cat ejemplos/ejemplo.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 asumen que cada objeto en la matriz JSON es una fila en la tabla:

```
En [69]: datos = pd.read_json('ejemplos/ejemplo.json')
```

```
En [70]: datos
```

```
Salida[70]:
```

```
      abc  
0 1 2 3  
1 4 5 6  
2 7 8 9
```

Para obtener un ejemplo ampliado de lectura y manipulación de datos JSON (incluidos los registros anidados), consulte el ejemplo de la base de datos de alimentos del USDA en el [Capítulo 7](#).

Si necesita exportar datos de pandas a JSON, una forma es usar los métodos `to_json` en Series y DataFrame:

```
En [71]: imprimir (data.to_json()) {"a":
{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}

En [72]: print(data.to_json(orient='registros')) [{"a":1,"b":2,"c":3},
{"a":4,"b":5,"c":6},{a":7,"b":8,"c":9}]
```

XML y HTML: Web Scraping Python tiene

muchas bibliotecas para leer y escribir datos en los omnipresentes formatos HTML y XML. Los ejemplos incluyen `lxml`, `Beautiful Soup` y `html5lib`. Si bien `lxml` es comparativamente mucho más rápido en general, las otras bibliotecas pueden manejar mejor los archivos HTML o XML con formato incorrecto.

`pandas` tiene una función integrada, `read_html`, que utiliza bibliotecas como `lxml` y `Beautiful Soup` para analizar automáticamente las tablas de los archivos HTML como objetos DataFrame. Para mostrar cómo funciona esto, descargué un archivo HTML (usado en la documentación de `pandas`) de la agencia gubernamental FDIC de los Estados Unidos que muestra fallas bancarias.¹ Primero, debe instalar algunas bibliotecas adicionales usadas por `read_html`:

```
conda instalar lxml
pip instalar beautifulsoup4 html5lib
```

Si no está utilizando `conda`, es probable que `pip` instale `lxml` también funcione.

La función `pandas.read_html` tiene varias opciones, pero de forma predeterminada busca e intenta analizar todos los datos tabulares contenidos en las etiquetas `<table>`. El resultado es una lista de objetos DataFrame:

```
En [73]: tablas = pd.read_html('examples/fdic_failed_bank_list.html')

En [74]: len(tablas)
Salida[74]: 1

En [75]: fallas = tablas [0]

En [76]: fallas.head()
Fuera[76]:
```

	Nombre del banco	Ciudad ST CERT \
0	banco aliado	Mora AR 91
1	La empresa bancaria de Woodbury 2	Woodbury Georgia 11297
	Primer Banco CornerStone Rey de Prusia PA	35312

¹ Para ver la lista completa, consulte <https://www.fdic.gov/bank/individual/failed/banklist.html>.

3	Banco de la compañía fiduciaria	Menfis TN 9956
4	Banco Estatal del Norte de Milwaukee	Milwaukee WI 20364
	Institución adquirente 0	Fecha de cierre
	Banco de hoy 23 de septiembre de 2016	17 de noviembre de 2016 1
	banco unido	19 de agosto de 2016 17 de noviembre de 2016
2	First-Citizens Bank & Trust Company	6 de mayo de 2016 6 de septiembre de 2016
	El banco del condado de Fayette	29 de abril de 2016 6 de septiembre de 2016 3
4	First-Citizens Bank & Trust Company	11 de marzo de 2016 16 de junio de 2016

Debido a que fails tiene muchas columnas, pandas inserta un carácter de salto de línea \.

Como aprenderá en capítulos posteriores, desde aquí podríamos proceder a realizar una limpieza y análisis de datos, como calcular el número de quiebras bancarias por año:

```
En [77]: close_timestamps = pd.to_datetime(failures['Closing Date'])
```

```
En [78]: close_timestamps.dt.year.value_counts()
```

```
Salida[78]:
```

```
2010 157
2009   140
2011   92
2012   51
2008   25
...
2004    4
2001    4
2007    3
2003 3 2000 2
```

```
Nombre: Fecha de cierre, Longitud: 15, dtype: int64
```

Analizar XML con lxml.objectify

XML (lenguaje de marcado extensible) es otro formato de datos estructurados común que admite datos anidados jerárquicos con metadatos. El libro que está leyendo actualmente se creó a partir de una serie de documentos XML de gran tamaño.

Anteriormente, mostré la función `pandas.read_html`, que usa lxml o BeautifulSoup bajo el capó para analizar datos de HTML. XML y HTML son estructuralmente similares, pero XML es más general. Aquí, mostraré un ejemplo de cómo usar lxml para analizar datos de un formato XML más general.

La Autoridad de Transporte Metropolitano de Nueva York (MTA) publica una **serie de datos sobre sus servicios de autobús y tren**. Aquí veremos los datos de rendimiento, que se encuentran en un conjunto de archivos XML. Cada servicio de tren o autobús tiene un archivo diferente (como

Performance_MNR.xml para Metro-North Railroad) que contiene datos mensuales como una serie de registros XML que se ven así:

```
<INDICADOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
```

```

<AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
<INDICATOR_NAME> Disponibilidad de escaleras mecánicas</
INDICATOR_NAME> <DESCRIPTION>Porcentaje de tiempo que las escaleras
mecánicas están operativas en todo el sistema. La tasa de disponibilidad se basa en las observaciones
físicas realizadas únicamente en la mañana de los días hábiles normales. Este es un nuevo indicador que
la agencia comenzó a informar en 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_LUGARES>1</
DECIMAL_LUGARES>
<YTD_TARGET>97.00</
YTD_TARGET> <YTD_ACTUAL></YTD_ACTUAL>
<MONTHLY_TARGET>97.00</
MONTHLY_TARGET> <MONTHLY_ACTUAL>< /MENSUAL_ACTUAL> </INDICADOR>

```

Usando `lxml.objectify`, analizamos el archivo y obtenemos una referencia al nodo raíz del archivo XML con `getroot()`:

```

desde lxml import objectify

ruta = 'ejemplos/mta_perf/Performance_MNR.xml' analizado =
objetificar.parse(abrir(ruta)) root = analizado.getroot()

```

`root.INDICATOR` devuelve un generador que produce cada elemento XML `<INDICATOR>`. Para cada registro, podemos completar un dictado de nombres de etiquetas (como `YTD_ACTUAL`) a valores de datos (excluyendo algunas etiquetas):

```

datos = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'CAMBIO_DESEADO', 'LUGARES_DECIMALES']

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, convierta esta lista de dictados en un `DataFrame`:

```

En [81]: rendimiento = pd.DataFrame(datos)

En [82]: perf.head()
Out[82]:
Trama de datos vacía

```

Columnas: []

Índice: []

Los datos XML pueden volverse mucho más complicados que este ejemplo. Cada etiqueta también puede tener metadatos. Considere una etiqueta de enlace HTML, que también es XML válido:

```
from io import StringIO tag = '<a
href="http://www.google.com">Google</a>' root =
objectify.parse(StringIO(tag)).getroot()
```

Ahora puede acceder a cualquiera de los campos (como href) en la etiqueta o el texto del enlace:

En [84]: raíz

Salida[84]: <Elemento a en 0x7f6b15817748>

En [85]: root.get('href')

Salida[85]: 'http://www.google.com'

Entrada [86]: root.text

Salida [86]: 'Google'

6.2 Formatos de datos binarios

Una de las formas más fáciles de almacenar datos (también conocida como serialización) de manera eficiente en formato binario es usar la serialización pickle integrada de Python . Todos los objetos pandas tienen un método `to_pickle` que escribe los datos en el disco en formato pickle:

En [87]: `cuadro = pd.read_csv('ejemplos/ex1.csv')`

En [88]: `marco`

Salida[88]:

	a	b	c	d mensaje 4
0	1	2	3	hola
1	5	6	2	9
			7	8 mundo foo
10	11	12		

En [89]: `frame.to_pickle('ejemplos/frame_pickle')`

Puede leer cualquier objeto "decapado" almacenado en un archivo usando el pickle incorporado directamente, o incluso más convenientemente usando `pandas.read_pickle`:

En [90]: `pd.read_pickle('ejemplos/frame_pickle')`

Salida[90]:

	a	b	c	d mensaje 4
0	1	2	3	hola
1	5	6	2	9
			7	8 mundo foo
10	11	12		



pickle solo se recomienda como formato de almacenamiento a corto plazo. El problema es que es difícil garantizar que el formato sea estable en el tiempo; es posible que un objeto decapado hoy no se deshaga con una versión posterior de una biblioteca. Hemos tratado de mantener la compatibilidad con versiones anteriores siempre que sea posible, pero en algún momento en el futuro puede ser necesario "romper" el formato pickle.

pandas tiene soporte integrado para dos formatos de datos binarios más: HDF5 y Message-Pack. Daré algunos ejemplos de HDF5 en la siguiente sección, pero lo animo a explorar diferentes formatos de archivo para ver qué tan rápidos son y qué tan bien funcionan para su análisis. Algunos otros formatos de almacenamiento para pandas o datos NumPy incluyen:

bcolz

Un formato binario comprimible orientado a columnas basado en la biblioteca de compresión Blosc.

Feather

Un formato de archivo multilenguaje orientado a columnas que diseñó con [Hadley Wickham](#) de la comunidad de programación R. Feather utiliza el formato de memoria en columnas [Apache Arrow](#).

Uso del formato HDF5

es un formato de archivo bien considerado diseñado para almacenar grandes cantidades de datos de matrices científicas. Está disponible como biblioteca C y tiene interfaces disponibles en muchos otros lenguajes, incluidos Java, Julia, MATLAB y Python. El "HDF" en HDF5 significa formato de datos jerárquicos. Cada archivo HDF5 puede almacenar múltiples conjuntos de datos y metadatos de apoyo. En comparación con formatos más simples, HDF5 admite la compresión sobre la marcha con una variedad de modos de compresión, lo que permite que los datos con patrones repetidos se almacenen de manera más eficiente. HDF5 puede ser una buena opción para trabajar con conjuntos de datos muy grandes que no caben en la memoria, ya que puede leer y escribir eficientemente pequeñas secciones de arreglos mucho más grandes.

Si bien es posible acceder directamente a los archivos HDF5 utilizando las bibliotecas PyTables o h5py, pandas proporciona una interfaz de alto nivel que simplifica el almacenamiento de objetos Series y DataFrame. La clase HDFStore funciona como un dict y maneja los detalles de bajo nivel:

```
En [92]: cuadro = pd.DataFrame({'a': np.random.randn(100)})
```

```
En [93]: tienda = pd.HDFStore('misdatos.h5')
```

```
En [94]: tienda['obj1'] = marco
```

```
En [95]: tienda['obj1_col'] = marco['a']
```

```
En [96]: almacenar
```

Salida[96]:

<clase 'pandas.io.pytables.HDFStore'> Ruta del
archivo: mydata.h5 /obj1

marco (forma->[100,1])

/obj1_col serie (forma->[100])

/obj2 frame_table (typ->appendable,nrows->100,ncols->1,indexers->
[indice]) /
obj3 frame_table (typ->appendable,nrows->100,ncols->1,indexers->
[indice])

Los objetos contenidos en el archivo HDF5 se pueden recuperar con la misma API similar a dict:

En [97]: almacenar['obj1']

Fuera[97]:

```
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 filas x 1 columna]
```

HDFStore admite dos esquemas de almacenamiento, 'fijo' y 'tabla'. Este último es generalmente más lento, pero admite operaciones de consulta utilizando una sintaxis especial:

En [98]: store.put('obj2', cuadro, formato='tabla')

En [99]: store.select('obj2', where=['index >= 10 and index <= 15'])

Fuera[99]:

```
a
10 1.007189 11
-1.296221
12 0.274992
13 0.228913
14 1.352917
15 0.886429
```

En [100]: tienda.cerrar()

El 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 le brinda un acceso directo a estas herramientas:

En [101]: frame.to_hdf('mydata.h5', 'obj3', format='table')

```
En [102]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
```

```
Fuera[102]:
```

```
a
0 -0.204708
1 0.478943
2 -0.519439
3 -0.555730
4 1.965781
```



Si está procesando datos almacenados en servidores remotos, como Amazon S3 o HDFS, utilizando un formato binario diferente diseñado para el almacenamiento distribuido como **Apache Parquet** puede ser más adecuado. Python para Parquet y otros formatos de almacenamiento similares aún se están desarrollando, por lo que no escribo sobre ellos en este libro.

Si trabaja con grandes cantidades de datos localmente, le animo a explorar

PyTables y h5py para ver cómo pueden adaptarse a sus necesidades. Dado que muchos análisis de datos

los problemas están vinculados a E/S (en lugar de vinculados a la CPU), el uso de una herramienta como HDF5 puede combinar acelere sus aplicaciones de manera significativa.



HDF5 no es una base de datos. Es más adecuado para escribir una vez, leer muchas conjuntos de datos. Si bien se pueden agregar datos a un archivo en cualquier momento, si hay varios los escritores lo hacen simultáneamente, el archivo puede corromperse.

Lectura de archivos de Microsoft Excel

pandas también admite la lectura de datos tabulares almacenados en archivos de Excel 2003 (y superior)

utilizando la clase `ExcelFile` o la función `pandas.read_excel`. Internamente estos

Las herramientas utilizan los paquetes complementarios `xlrd` y `openpyxl` para leer archivos XLS y XLSX, respectivamente. Es posible que deba instalarlos manualmente con `pip` o `conda`.

Para usar `ExcelFile`, cree una instancia pasando una ruta a un archivo xls o xlsx :

```
En [104]: xlsx = pd.ExcelFile('ejemplos/ex1.xlsx')
```

Los datos almacenados en una hoja se pueden leer en `DataFrame` con `parse`:

```
En [105]: pd.read_excel(xlsx, 'Hoja1')
```

```
Fuera[105]:
```

```
mensaje
un 0 1      do 3  4 hola
15   6   7   8 mundo
2 9 10 11 12      Foo
```

Si está leyendo varias hojas en un archivo, entonces es más rápido crear el `ExcelFile`, pero también puede simplemente pasar el nombre del archivo a `pandas.read_excel`:

```
En [106]: marco = pd.read_excel('ejemplos/ex1.xlsx', 'Hoja1')
```

```
En [107]: cuadro
```

```
Fuera[107]:
```

```
   a  b  c  mensaje
0  1   2   3     hola
15  6   7     mundo
2  9 10 11 12      Foo
```

Para escribir datos de pandas en formato Excel, primero debe crear un ExcelWriter, luego escriba datos en él usando el método to_excel de pandas objects :

```
En [108]: escritor = pd.ExcelWriter('ejemplos/ex2.xlsx')
```

```
En [109]: frame.to_excel(escritor, 'Hoja1')
```

```
En [110]: escritor.guardar()
```

También puede pasar una ruta de archivo a to_excel y evitar el ExcelWriter:

```
En [111]: frame.to_excel('ejemplos/ex2.xlsx')
```

6.3 Interacción con API web

Muchos sitios web tienen API públicas que proporcionan fuentes de datos a través de JSON o algún otro formato.

Hay varias formas de acceder a estas API desde Python; uno fácil de usar

El método que recomiendo es el paquete `solicitudes` .

Para encontrar los últimos 30 problemas de GitHub para pandas en GitHub, podemos hacer un GET HTTP solicitud usando la biblioteca de solicitudes de complementos :

```
In [113]: solicitudes de importación
```

```
En [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'
```

```
En [115]: resp = solicitudes.get(url)
```

```
En [116]: resp.
```

```
Salida[116]: <Respuesta [200]>
```

El método json del objeto Response devolverá un diccionario que contiene JSON analizado en objetos Python nativos:

```
En [117]: datos = resp.json()
```

```
En [118]: datos[0]['titulo']
```

```
Out[118]: 'El período no se redondea hacia abajo para frecuencias menores a 1 hora'
```

Cada elemento de los datos es un diccionario que contiene todos los datos que se encuentran en un GitHub página del problema (excepto los comentarios). Podemos pasar datos directamente a DataFrame y extraer campos de interés:

```

En [119]: problemas = pd.DataFrame (datos, columnas = ['número', 'título',
.....:                                             'etiquetas', 'estado'])

En [120]: problemas
Salida[120]:
título del número \
0      17666 El periodo no se redondea hacia abajo para frecuencias menos...
1      donde DOC: mejore la cadena de documentación de la función
2      17665 17664 COMPAT: omite la prueba de 32 bits en int repr 17662 implemente
3      la clase Delegator 17654 ERROR: corrija el cambio de nombre de la serie
4      llamado con str alterin ...

..      ...      ...

25 17603 ERROR: Localizar correctamente cadenas de fecha y hora
ingenuas... 26 17599 core.dtypes.generic --> cython 27 17596 Fusionar la
funcionalidad cdate_range en bdate_range 28 17587 Corrección de errores
en el agrupador de tiempo cuando se aplica para list gro...
29 17583 ERROR: corregir tz-aware DatetimeIndex + TimedeltaInd...

estado de las etiquetas

0 [] abrir 1 [{'id': 134699, 'url': 'https://api.github.com... abrir 2 [{'id': 563047854,
'url': 'https://api.github.com... abierto [] abierto 3 4 [{'id': 76811, 'url': 'https://
api.github.com/... abierto

..      ...      ...

25 [{'id': 76811, 'url': 'https://api.github.com/... abierto 26 [{'id': 49094459, 'url':
'https://api.github.com/... abrir 27 [{'id': 35818298, 'url': 'https://api.github.com/...
28 [{'id': 233160, 'url': 'https://api.github.com/... abrir 29 [{'id': 76811, 'url': 'https://
api.github.com/... abrir [30 filas x 4 columnas]

```

Con un poco de esfuerzo, puede crear algunas interfaces de nivel superior para API web que devuelven objetos DataFrame para facilitar el análisis.

6.4 Interactuando con bases de datos

En un entorno empresarial, es posible que la mayoría de los datos no se almacenen en archivos de texto o de Excel. basado en SQL Las bases de datos relacionales (como SQL Server, PostgreSQL y MySQL) se utilizan ampliamente, y muchas bases de datos alternativas se han vuelto muy populares. La elección de la base de datos es generalmente depende del rendimiento, la integridad de los datos y las necesidades de escalabilidad de un solicitud.

Cargar datos de SQL en un DataFrame es bastante sencillo y pandas tiene algunas funciones para simplificar el proceso. Como ejemplo, crearé una base de datos SQLite usando el controlador sqlite3 incorporado de Python :

```

En [121]: importar sqlite3

En [122]: consulta
          = .....: prueba CREAR TABLA

```

```
.....: (a VARCHAR(20), b VARCHAR(20), d
ENTERO .....: c
REAL, .....: );"""
```

```
En [123]: con = sqlite3.connect('misdatos.sqlite')
```

```
En [124]: con.ejecutar(consultar)
```

```
Salida[124]: <sqlite3.Cursor en 0x7f6b12a50f10>
```

```
En [125]: con.commit()
```

Luego, inserte algunas filas de datos:

```
En [126]: datos = [('Atlanta', 'Georgia', 1.25, 6),
.....:             ('Tallahassee', 'Florida', 2.6, 3), ('Sacramento',
.....:             'California', 1.7, 5)]
```

```
En [127]: sentencia = "INSERTAR EN VALORES de prueba (?, ?, ?, ?)"
```

```
En [128]: con.executemany(stmt, data)
```

```
Salida[128]: <sqlite3.Cursor en 0x7f6b15c66ce0>
```

```
En [129]: con.commit()
```

La mayoría de los controladores Python SQL (PyODBC, pycopg2, MySQLdb, pymssql, etc.) devuelven una lista de tuplas al seleccionar datos de una tabla:

```
En [130]: cursor = con.execute('seleccionar * de la prueba')
```

```
En [131]: filas = cursor.fetchall()
```

```
Entrada [132]:
```

```
filas
```

```
Salida[132]: [('Atlanta', 'Georgia', 1.25, 6),
('Tallahassee', 'Florida', 2.6, 3), ('Sacramento',
'California', 1.7, 5)]
```

Puede pasar la lista de tuplas al constructor de DataFrame, pero también necesita los nombres de las columnas, contenidos en el atributo de descripción del cursor :

```
In [133]: cursor.description Out[133]:
```

```
(( 'a',
```

```
Ninguno, Ninguno, Ninguno, Ninguno, Ninguno, Ninguno),
```

```
( 'b', Ninguno, Ninguno, Ninguno, Ninguno, Ninguno,
```

```
Ninguno), ( 'c', Ninguno, Ninguno, Ninguno, Ninguno,
```

```
Ninguno, Ninguno), ( 'd', Ninguno, Ninguno, Ninguno, Ninguno, Ninguno, Ninguno))
```

```
En [134]: pd.DataFrame(filas, columnas=[x[0] para x en cursor.descripcion])
```

```
Fuera[134]:
```

```
          a          b          cd
0  atlanta  Georgia  1.25  6  1
Tallahassee Florida  2.60  3  2  Sacramento
California  1.70  5
```

Esto es bastante complicado y preferiría no repetirlo cada vez que consulta la base de datos. El **proyecto SQLAlchemy** es un popular conjunto de herramientas Python SQL que abstrae muchas de las diferencias comunes entre las bases de datos SQL. pandas tiene una función `read_sql` que le permite leer datos fácilmente desde una conexión SQLAlchemy general. Aquí, nos conectaremos a la misma base de datos SQLite con SQLAlchemy y leeremos los datos de la tabla creada antes:

```
En [135]: importar sqlalchemy como sqla

En [136]: db = sqla.create_engine('sqlite:///mydata.sqlite')

En [137]: pd.read_sql('seleccionar * de prueba', db)
Fuera[137]:
```

	a	b	cd
0	Atlanta Georgia	1,25	6 1 Tallahassee
Florida	2,60	3 2 Sacramento	California 1,70
5			

6.5 Conclusión

Con frecuencia, obtener acceso a los datos es el primer paso en el proceso de análisis de datos. Hemos visto una serie de herramientas útiles en este capítulo que deberían ayudarlo a comenzar. En los próximos capítulos, profundizaremos en la disputa de datos, la visualización de datos, el análisis de series temporales y otros temas.

Limpieza y preparación de datos

Durante el transcurso del análisis y el modelado de datos, se dedica una cantidad significativa de tiempo a la preparación de datos: carga, limpieza, transformación y reorganización. A menudo se informa que tales tareas ocupan el 80% o más del tiempo de un analista. A veces, la forma en que se almacenan los datos en archivos o bases de datos no tiene el formato correcto para una tarea en particular. Muchos investigadores optan por realizar un procesamiento ad hoc de datos de un formulario a otro utilizando un lenguaje de programación de propósito general, como Python, Perl, R o Java, o herramientas de procesamiento de texto de Unix como sed o awk. Afortunadamente, pandas, junto con las características integradas del lenguaje Python, le proporciona un conjunto de herramientas rápido, flexible y de alto nivel que le permite manipular los datos en la forma correcta.

Si identifica un tipo de manipulación de datos que no se encuentra en este libro ni en ninguna otra parte de la biblioteca de pandas, siéntase libre de compartir su caso de uso en una de las listas de correo de Python o en el sitio de pandas en GitHub. De hecho, gran parte del diseño y la implementación de pandas ha sido impulsado por las necesidades de las aplicaciones del mundo real.

En este capítulo analizo herramientas para datos faltantes, datos duplicados, manipulación de cadenas y algunas otras transformaciones de datos analíticos. En el próximo capítulo, me enfoco en combinar y reorganizar conjuntos de datos de varias maneras.

7.1 Manejo de datos faltantes

La falta de datos ocurre comúnmente en muchas aplicaciones de análisis de datos. Uno de los objetivos de pandas es hacer que trabajar con datos faltantes sea lo menos doloroso posible. Por ejemplo, todas las estadísticas descriptivas de los objetos pandas excluyen los datos faltantes de forma predeterminada.

La forma en que se representan los datos faltantes en los objetos pandas es algo imperfecta, pero es funcional para muchos usuarios. Para datos numéricos, pandas usa el valor de coma flotante NaN (No es un número) para representar los datos que faltan. Llamamos a esto un valor centinela que se puede detectar fácilmente:

```
En [10]: string_data = pd.Series(['aardvark', 'alcachofa', np.nan, 'aguacate'])

En [11]: cadena_datos
Fuera[11]:
0 oso hormiguero
1 alcachofa
2 Yaya
3 palta
tipo: objeto

En [12]: string_data.isnull()
Fuera[12]:
0 FALSO
1 FALSO
2 Verdadero
3 FALSO
dtipo: booleano
```

En pandas, hemos adoptado una convención utilizada en el lenguaje de programación R por referencia llamar a los datos faltantes como NA, que significa no disponible. En las aplicaciones de estadísticas, los datos NA pueden ser datos que no existen o que existen pero no se observaron (a través de problemas con la recopilación de datos, por ejemplo). Al limpiar los datos de análisis, a menudo es importante hacer un análisis de los datos que faltan para identificar los datos problemas de recopilación o posibles sesgos en los datos causados por datos faltantes.

El valor integrado de Python Ninguno también se trata como NA en las matrices de objetos:

```
En [13]: string_data[0] = Ninguno

En [14]: string_data.isnull()
Fuera[14]:
0 Verdadero
1 FALSO
2 Verdadero
3 FALSO
dtipo: booleano
```

Se está trabajando en el proyecto pandas para mejorar los detalles internos de cómo los datos faltantes se manejan, pero las funciones de la API del usuario, como pandas.isnull, abstraen muchos de los detalles molestos. Consulte la [Tabla 7-1](#) para obtener una lista de algunas funciones relacionadas al manejo de datos perdidos.

Tabla 7-1. Métodos de manejo NA

Argumento	Descripción
dropna	Filtre las etiquetas de los ejes en función de si los valores de cada etiqueta tienen datos faltantes, con diferentes umbrales de cómo muchos datos perdidos que tolerar.
fillna	Rellene los datos faltantes con algún valor o utilizando un método de interpolación como 'ffill' o 'bfill'.
isnull	Devuelve valores booleanos que indican qué valores faltan/NA.
notnull	Negación de isnull.

Filtrado de datos faltantes

Hay algunas formas de filtrar los datos que faltan. Si bien siempre tiene la opción de hacerlo a mano usando `pandas.isnull` y la indexación booleana, el `dropna` puede ser útil.

En una Serie, devuelve la Serie con solo los datos no nulos y los valores de índice:

```
En [15]: from numpy import nan as NA

En [16]: datos = pd.Series([1, NA, 3.5, NA, 7])

En [17]: data.dropna()
Fuera[17]:
0 1.0
2 3.5
      7.0
4 tipo: float64
```

Esto es equivalente a:

```
En [18]: datos[datos.notnull()]
Fuera[18]:
0 1.0
2    3.5
4 7.0
tipo: float64
```

Con los objetos `DataFrame`, las cosas son un poco más complejas. Es posible que desee soltar filas o columnas que son todas NA o solo aquellas que contienen NA. `dropna` por defecto gotas cualquier fila que contenga un valor faltante:

```
En [19]: datos = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                               ...:                                [NA, NA, NA], [NA, 6.5, 3.]])

En [20]: limpiado = data.dropna()

En [21]: datos
Fuera[21]:
   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

En [22]: limpiado
Fuera[22]:
   0    1    2
0 1,0 6,5 3,0
```

Pasar `how='all'` solo eliminará las filas que son todas NA:

```
En [23]: data.dropna(how='all')
Fuera[23]:
   0    1    2
```

```
0 1,0 6,5 3,0
1 1,0 NaN NaN
3 NaN 6,5 3,0
```

Para soltar columnas de la misma manera, pase axis=1:

```
En [24]: datos[4] = NA
```

```
En [25]: datos
```

```
Fuera[25]:
           1      2      4
0 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
```

```
En [26]: data.dropna(axis=1, how='all')
```

```
Fuera[26]:
           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
```

Una forma relacionada de filtrar filas de DataFrame tiende a referirse a datos de series temporales. Suponer desea mantener solo filas que contengan un cierto número de observaciones. Puede indicar esto con el argumento de umbral :

```
En [27]: df = pd.DataFrame(np.random.randn(7, 3))
```

```
En [28]: df.iloc[:4, 1] = NA
```

```
En [29]: df.iloc[:2, 2] = NA
```

```
En [30]: d.f.
```

```
Fuera[30]:
           0           1           2
0 -0.204708           Yaya           Yaya
1 -0.555730 2           NaN NaN
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
```

```
En [31]: df.dropna()
```

```
Fuera[31]:
           0           1           2
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

```
En [32]: df.dropna(umbral=2)
```

```
Fuera[32]:
      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
```

Rellenar los datos que faltan

En lugar de filtrar los datos que faltan (y posiblemente descartar otros datos junto con ellos), es posible que desee rellenar los "agujeros" de varias maneras. Para la mayoría de los propósitos, el método `fillna` es la función caballo de batalla a utilizar. Llamar a `fillna` con una constante reemplaza los valores faltantes con ese valor:

```
En [33]: df.fillna(0)
Fuera[33]:
      0      1      2
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
```

Al llamar a `fillna` con un dict, puede usar un valor de relleno diferente para cada columna:

```
En [34]: df.fillna({'1': 0.5, '2': 0})
Fuera[34]:
      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
```

`fillna` devuelve un nuevo objeto, pero puede modificar el objeto existente en el lugar:

```
En [35]: _ = df.fillna(0, inplace=True)

En [36]: d.f.
Fuera[36]:
      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
```

Los mismos métodos de interpolación disponibles para la reindexación se pueden utilizar con `fillna`:

```
En [37]: df = pd.DataFrame(np.random.randn(6, 3))
```

```
En [38]: df.iloc[2:, 1] = NA
```

```
En [39]: df.iloc[4:, 2] = NA
```

```
En [40]: d.f.
```

```
Fuera[40]:
```

	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	Yaya	Yaya

```
En [41]: df.fillna(método='rellenar')
```

```
Fuera[41]:
```

	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

```
En [42]: df.fillna(método='rellenar', límite=2)
```

```
Fuera[42]:
```

	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` puedes hacer muchas otras cosas con un poco de creatividad. Por ejemplo tu podría pasar el valor medio o mediano de una Serie:

```
En [43]: datos = pd.Series([1., NA, 3.5, NA, 7])
```

```
En [44]: data.fillna(data.mean())
```

```
Fuera[44]:
```

0	1.000000
1	3.833333
2	3.500000
3	3.833333
4	7.000000

tipo: float64

Consulte la [Tabla 7-2](#) para obtener una referencia sobre `fillna`.

Tabla 7-2. argumentos de la función fillna

Argumento	Descripción
valor	Valor escalar u objeto similar a un dictado para usar para completar los valores que faltan
método	Interpolación; por defecto 'rellenar' si se llama a la función sin otros argumentos
eje	Eje para rellenar; eje por defecto =0
inplace	Modificar el objeto que llama sin producir una copia
límite	Para llenado hacia adelante y hacia atrás, número máximo de periodos consecutivos para llenar

7.2 Transformación de datos

Hasta ahora, en este capítulo nos hemos preocupado por reorganizar los datos. Filtrado, limpieza, y otras transformaciones son otra clase de operaciones importantes.

Eliminación de duplicados

Se pueden encontrar filas duplicadas en un DataFrame por varias razones. Aquí hay un ejemplo:

```
En [45]: datos = pd.DataFrame({'k1': ['uno', 'dos'] * 3 + ['dos'],
.....:                        'k2': [1, 1, 2, 3, 3, 4, 4]})

En [46]: datos
Fuera[46]:
   k1 k2
0  uno  1
1  dos  1
2  uno  2
3  dos  3
4  uno  3
5  dos  4
6  dos  4
```

El método DataFrame.duplicated devuelve una serie booleana que indica si cada fila es un duplicado (se ha observado en una fila anterior) o no:

```
En [47]: datos.duplicated()
Fuera[47]:
0 falso
1  FALSE
2  FALSE
3  FALSE
4  FALSE
5  FALSE
Verdadero
6 dtipo: bool
```

De manera relacionada, drop_duplicates devuelve un DataFrame donde se encuentra la matriz duplicada .
FALSE:

```

En [48]: data.drop_duplicates()
Salida[48]:
   k1 k2
0 uno 1 1
1 dos 1
2 uno 2
3 dos 3
4 uno 3
5 dos 4

```

Ambos métodos por defecto consideran todas las columnas; como alternativa, puede especificar cualquier subconjunto de ellos para detectar duplicados. Supongamos que tuviéramos una columna adicional de valores y quisiéramos filtrar los duplicados solo en función de la columna 'k1' :

```

En [49]: datos['v1'] = rango(7)

En [50]: datos.drop_duplicates(['k1'])
Salida[50]:
   k1 k2 v1
0 uno 1 0
1 dos 1    1

```

deduplicated y drop_duplicates por defecto mantienen la primera combinación de valores observada. Pasar keep='last' devolverá el último:

```

En [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Salida[51]:
   k1 k2 v1
0 uno 1 0
1 dos 1    1
2 uno 2 2
3 dos 3 3
4 uno 3 4
5 dos 4 6

```

Transformación de datos usando una función o mapeo Para muchos

conjuntos de datos, es posible que desee realizar alguna transformación basada en los valores en una matriz, Serie o columna en un DataFrame. Considere los siguientes datos hipotéticos recopilados sobre varios tipos de carne:

```

En [52]: datos = pd.DataFrame({'comida': ['tocino', 'carne de cerdo desmenuzada', 'tocino',
.....:                                'Pastrami', 'corned beef', 'Bacon', 'pastrami',
.....:                                'jamón a la miel', 'nova lox'],
.....:                        'onzas': [4, 3, 12, 6, 7, 5, 8, 3, 5, 6]})

En [53]: datos
Fuera[53]:
           comida onzas de
0 1 tocino de puerco    3.0
desmenuzado 2    12.0

```

	Pastrami	3	6.0
4	corned beef		7.5
5	Tocino		8.0
6	pastrami		3.0
7	jamón miel		5.0
8	nova lox		6.0

Suponga que desea agregar una columna que indique el tipo de animal que cada alimento vino de. Escribamos un mapeo de cada tipo de carne distinto al tipo de animal:

```
carne_a_animal = {
    'tocino': 'cerdo',
    'carne de cerdo desmenuzada': 'cerdo',
    'pastrami': 'vaca',
    'corned beef': 'vaca',
    'jamón de miel': 'cerdo',
    'nova lox': 'salmón'
}
```

El método de mapa en una serie acepta una función o un objeto similar a un dictado que contiene un mapa. ping, pero aquí tenemos un pequeño problema en que algunas de las carnes están en mayúsculas y otros no lo son. Por lo tanto, necesitamos convertir cada valor a minúsculas usando `str.lower` Método de la serie:

```
En [55]: minúsculas = data['comida'].str.lower()
```

```
En [56]: minúsculas
```

```
Fuera[56]:
```

```
0          tocino
1   cerdo desmenuzado
2          tocino
          pastrami
3 4   carne en conserva
5          tocino
6          pastrami
7   jamón de miel
8 nova lox
```

```
Nombre: alimento, dtype: objeto
```

```
En [57]: datos['animal'] = minúsculas.map(carne_a_animal)
```

```
En [58]: datos
```

```
Fuera[58]:
```

```
comida onzas animal
0          tocino de cerdo   4.0
1 puerco desmenuzado       3.0
2          tocino de cerdo  12.0
3   Pastrami 4             6.0   vaca
carne en conserva       7.5   vaca
5          Tocino         8.0   cerdo
6   pastrami             3.0   vaca
```

```

7    jamón de      5,0      cerdo
8    miel nova lox 6,0 salmón

```

También podríamos haber pasado una función que hiciera todo el trabajo:

```
En [59]: data["food"].map(lambda x: meat_to_animal[x.lower()])
```

```
Fuera[59]:
```

```

0 cerdo
1      cerdo
2      cerdo
3      vaca
4      vaca
5      cerdo
6      vaca
7      cerdo
8    salmón

```

```
Nombre: alimento, dtype: objeto
```

El uso del mapa es una forma conveniente de realizar transformaciones por elementos y otras operaciones relacionadas con la limpieza de datos.

Sustitución de valores

Rellenar los datos que faltan con el método `fillna` es un caso especial de valor más general reemplazo. Como ya ha visto, el mapa se puede utilizar para modificar un subconjunto de valores en un objeto pero `replace` proporciona una forma más simple y flexible de hacerlo. Vamos a con Considere esta serie:

```
En [60]: datos = pd.Series([1., -999., 2., -999., -1000., 3.])
```

```
En [61]: datos
```

```
Fuera[61]:
```

```

0      1.0
1    -999.0
2      2.0
3 -999.0
4 -1000.0
5      3.0

```

```
tipo: float64
```

Los valores -999 pueden ser valores centinela para datos faltantes. Para reemplazar estos con NA valores que pandas entiende, podemos usar `replace`, produciendo una nueva serie (a menos que `pasas inplace=True`):

```
En [62]: data.replace(-999, np.nan)
```

```
Fuera[62]:
```

```

0      1.0
1      Yaya
2      2.0
3      Yaya
4 -1000.0

```

```
5      3.0
tipo: float64
```

Si desea reemplazar varios valores a la vez, pase una lista y luego el valor sustitutivo:

```
En [63]: datos.reemplazar([-999, -1000], np.nan)
Fuera[63]:
0 1.0
1      Yaya
2      2.0
      Yaya
3 4      Yaya
5      3.0
tipo: float64
```

Para usar un reemplazo diferente para cada valor, pase una lista de sustitutos:

```
En [64]: datos.reemplazar([-999, -1000], [np.nan, 0])
Fuera[64]:
0 1.0
1      Yaya
2      2.0
3      Yaya
4      0.0
5      3.0
tipo: float64
```

El argumento pasado también puede ser un dictado:

```
En [65]: datos.reemplazar({-999: np.nan, -1000: 0})
Fuera[65]:
0 1.0
1      Yaya
2      2.0
3      Yaya
4 0.0
5 3.0
tipo: float64
```



El método `data.replace` es distinto de `data.str.replace`, que realiza la sustitución de cadenas por elementos. Nos fijamos en estos métodos de cadena en Series más adelante en el capítulo.

Cambio de nombre de índices de eje

Al igual que los valores de una serie, las etiquetas de los ejes se pueden transformar de manera similar mediante una función o un mapa. ping de alguna forma para producir objetos nuevos, etiquetados de manera diferente. También puedes modificar los ejes en el lugar sin crear una nueva estructura de datos. Aquí hay un ejemplo simple:

```
En [66]: datos = pd.DataFrame(np.arange(12).reshape((3, 4)),
.....:                        index=['Ohio', 'Colorado', 'Nueva York'],
.....:                        columns = ['uno', 'dos', 'tres', 'cuatro'])
```

Al igual que una serie, los índices de los ejes tienen un método de mapa :

```
En [67]: transformar = lambda x: x[:4].upper()

En [68]: data.index.map(transformar)
Salida[68]: Índice(['OHIO', 'COLO', 'NUEVO'], dtype='objeto')
```

Puede asignar al índice, modificando el DataFrame en el lugar:

```
En [69]: data.index = data.index.map(transformar)

En [70]: datos
Fuera[70]:
```

	uno	dos	tres	CUATRO
OHIO	0	1	2	3
COLO	4	5	6	7
NUEVO	8	9	10	11

Si desea crear una versión transformada de un conjunto de datos sin modificar el original, un método útil es renombrar:

```
En [71]: data.rename(index=str.title, column=str.upper)
Fuera[71]:
```

	UNO	DOS	TRES	CUATRO
Ohio	0	1	2	3
Colo	4	5	6	7
Nuevo	8	9	10	11

En particular, el cambio de nombre se puede usar junto con un objeto similar a un dictado que proporciona un nuevo valor. Es útil para un subconjunto de las etiquetas de los ejes:

```
En [72]: data.rename(index={'OHIO': 'INDIANA'},
.....:                columns = {'tres': 'cucú'})
Fuera[72]:
```

	uno	dos	peekaboo	cuatro
INDIANA	0	1	2	3
COLO	4	5	6	7
NUEVO	8	9	10	11

renombrar le ahorra la tarea de copiar el DataFrame manualmente y asignar a sus atributos de índice y columnas. Si desea modificar un conjunto de datos en el lugar, pase en el lugar = Verdadero:

```
En [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

En [74]: datos
Fuera[74]:
```

	uno	dos	tres	CUATRO
INDIANA	0	1	2	3

COLO	4	5	6	7
NUEVO	8	9	10	11

Discretización y Binning

Los datos continuos a menudo se discretizan o se separan en "contenedores" para su análisis.

Suponga que tiene datos sobre un grupo de personas en un estudio y desea agrupar en cubos de edad discretos:

```
En [75]: edades = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Dividámoslos en grupos de 18 a 25, 26 a 35, 36 a 60 y finalmente 61 y más. A para hacerlo, tienes que usar cut, una función en pandas:

```
En [76]: contenedores = [18, 25, 35, 60, 100]
```

```
En [77]: gatos = pd.cut(edades, bins)
```

```
En [78]: gatos
```

```
Fuera[78]:
```

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]]
```

```
Longitud: 12
```

```
Categorías (4, intervalo[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

El objeto que devuelve pandas es un objeto categórico especial . La salida que ves

describe los contenedores calculados por pandas.cut. Puedes tratarlo como una matriz de cadenas.

indicando el nombre del contenedor; internamente contiene una matriz de categorías que especifica la distribución

Nombres de categoría de tinte junto con un etiquetado para los datos de edades en el atributo de códigos :

```
En [79]: códigos.gatos
```

```
Salida[79]: matriz([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

```
En [80]: gatos.categorías
```

```
Fuera[80]:
```

```
Índice de intervalo [(18, 25], (25, 35], (35, 60], (60, 100]]
```

```
cerrado = 'derecha',
```

```
dtype='intervalo[int64]')
```

```
En [81]: pd.value_counts(gatos)
```

```
Fuera[81]:
```

```
(18, 25] 5
```

```
(35, 60] 3
```

```
(25, 35] 3
```

```
(60, 100] 1
```

```
tipo: int64
```

Tenga en cuenta que pd.value_counts(cats) son los recuentos de contenedores para el resultado de pandas.cut.

De acuerdo con la notación matemática para intervalos, un paréntesis significa que el lado está abierto, mientras que el corchete significa que está cerrado (inclusive). Puedes cambiar cual el lado se cierra pasando right=False:

```
En [82]: pd.cut(edades, [18, 26, 36, 61, 100], derecha=False)
Fuera[82]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100 ), [36, 61), [36, 61 ), [26, 36)]
```

```
Longitud:
12 Categorías (4, intervalo[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

También puede pasar sus propios nombres de contenedores pasando una lista o matriz a la opción de etiquetas :

```
En [83]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

En [84]: pd.cut(edades, bins, etiquetas=nombres_de_grupo)
Fuera[84]:
[Joven, Joven, Joven, Adulto joven, Joven, ..., Adulto joven, Mayor, Edad media, Edad media,
Adulto joven]
Longitud:
12 Categorías (4, objeto): [Youth < YoungAdult < MiddleAged < Senior]
```

Si pasa un número entero de contenedores para cortar en lugar de bordes de contenedores explícitos, calculará contenedores de igual longitud en función de los valores mínimo y máximo de los datos.

Considere el caso de algunos datos uniformemente distribuidos divididos en cuartos:

```
En [85]: datos = np.random.rand(20)

En [86]: pd.cut(data, 4, precision=2)
Salida[86]:
[(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]]
Longitud:
20 Categorías (4, intervalo[float64]): [(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, `qcut`, agrupa los datos en función de cuantiles de muestra. Dependiendo de la distribución de los datos, el uso de corte generalmente no dará como resultado que cada contenedor tenga la misma cantidad de puntos de datos. Dado que `qcut` usa cuantiles de muestra en su lugar, por definición obtendrá contenedores de aproximadamente el mismo tamaño:

```
En [87]: data = np.random.randn(1000) # Normalmente distribuida

En [88]: cats = pd.qcut(data, 4) # Cortar en cuantiles

Entrada [89]:
gatos
Salida[89]: [(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62] , ..., (-0.68,
-0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.928], (-0.68, -0.0265]]

Longitud: 1000
Categorías (4, intervalo[float64]): [(-2.95, -0.68] < (-0.68, -0.0265] < (-0.0265, 0.62] <
(0.62, 3.928]]
```

```
En [90]: pd.value_counts(gatos)
Salida[90]:
(0.62, 3.928]          250
(-0.0265, 0.62]       250
(-0.68, -0.0265] (-2.95,
-0.68] dtype: int64    250
```

Similar a cortar, puede pasar sus propios cuantiles (números entre 0 y 1, inclusive):

```
En [91]: pd.qcut(datos, [0, 0.1, 0.5, 0.9, 1.])
Salida[91]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.0265, 1.286], ..., (-1.187, -0.0265],
(-1.187, -0.0265], (-2.95, -1.187], (-0.0265, 1.286], (-1.187, -0.0265]]

Longitud: 1000
Categorías (4, intervalo[float64]): [(-2.95, -1.187] < (-1.187, -0.0265] < (-0.0265, 1.286] <
(1.286, 3.928]]
```

Regresaremos a cut y qcut más adelante en el capítulo durante nuestra discusión sobre las operaciones de agregación y de grupo, ya que estas funciones de discretización son especialmente útiles para el análisis de cuantiles y de grupos.

Detección y filtrado de valores atípicos

Filtrado o la transformación de valores atípicos es en gran medida una cuestión de aplicar operaciones de matriz. Considere un DataFrame con algunos datos normalmente distribuidos:

```
En [92]: datos = pd.DataFrame(np.random.randn(1000, 4))

En [93]: datos.describe()
Fuera[93]:
```

	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

Suponga que desea encontrar valores en una de las columnas que excedan 3 en valor absoluto:

```
En [94]: columna = datos[2]

En [95]: col[np.abs(col) > 3]
Salida[95]:
41      -3.399312
136    -3.745356
Nombre: 2, tipo de d: float64
```

Para seleccionar todas las filas que tengan un valor superior a 3 o -3, puede utilizar cualquier método en un DataFrame booleano:

```
En [96]: datos[np.abs(datos) > 3].cualquiera(1)
Fuera[96]:
```

	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 valores se pueden establecer en función de estos criterios. Aquí hay un código para limitar los valores fuera del intervalo -3 a 3:

```
En [97]: datos[np.abs(datos) > 3] = np.sign(datos) * 3
En [98]: datos.describe()
Fuera[98]:
```

	0	1	2	3
cuenta	1000.000000	1000.000000	1000.000000	0.050286
	0.025567			
significar	-0.051765		-0.001399	
estándar	0.992920	1.004214	0.991414	0.995761
mínimo	-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
máximo	2.653656	3.000000	2.735527	3.000000

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

```
En [99]: np.sign(datos).head()
Salida[99]:
```

	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	3 -1,0 -1,0 1,0
	-1,0			
4	-1,0	1,0	-1,0	-1,0

Permutación y muestreo aleatorio Permutar (reordenar

aleatoriamente) una serie o las filas en un marco de datos es fácil de hacer usando la función

`numpy.random.permutation`. Llamar a la permutación con la longitud del eje que desea permutar produce una matriz de números enteros que indican el nuevo orden:

```
En [100]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
```

```
En [101]: muestra = np.random.permutation(5)
```

```
En [102]: mestreador
```

```
Salida[102]: matriz([3, 1, 4, 2, 0])
```

Esa matriz se puede usar en la indexación basada en iloc o en la función de toma equivalente :

```
En [103]: d.f.
```

```
Salida[103]:
```

```

    0 1      2      3
0   0  1      2      3
1   4  5      6      7
2   8  9 10 11
3  12 13 14 15 16 17
18 19
```

```
En [104]: df.take(mestreador)
```

```
Salida[104]:
```

```

    0 3 3 12 13 14 15
1   4  5      6      7
4  16 17 18 19
2   8  9 10 11
0   0  1      2      3
```

Para seleccionar un subconjunto aleatorio sin reemplazo, puede usar el método de muestra en Serie y marco de datos:

```
En [105]: df.muestra(n=3)
```

```
Salida[105]:
```

```

    0 1      2      3
3  12 13 14 15
4  16 17 18 19
2   8  9 10 11
```

Para generar una muestra con reemplazo (para permitir opciones repetidas), pase `replace=True` a la muestra:

```
En [106]: opciones = pd.Series([5, 7, -1, 6, 4])
```

```
En [107]: sorteos = opciones.muestra(n = 10, reemplazar = Verdadero)
```

```
En [108]: sorteos
```

```
Salida[108]:
```

```

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

```

4      4
0      5
4      4
tipo: int64

```

Indicador de computación/variables ficticias Otro tipo

de transformación para aplicaciones de modelado estadístico o aprendizaje automático es convertir una variable categórica en una matriz "ficticia" o "indicadora". Si una columna en un DataFrame tiene k valores distintos, usted derivaría una matriz o DataFrame con k columnas que contienen todos 1 y 0. pandas tiene una función `get_dummies` para hacer esto, aunque diseñar una no es difícil. Volvamos a un ejemplo anterior de DataFrame:

```

En [109]: df = pd.DataFrame({'clave': ['b', 'b', 'a', 'c', 'a', 'b'], 'datos1': rango(6) })
.....:

En [110]: pd.get_dummies(df['clave'])
Salida[110]:
  abc
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, es posible que desee agregar un prefijo a las columnas en el marco de datos del indicador, que luego se puede fusionar con los otros datos. `get_dummies` tiene un argumento de prefijo para hacer esto:

```

En [111]: maniqués = pd.get_dummies(df['clave'], prefijo='clave')

En [112]: df_with_dummy = df[['data1']].join(dummies)

En [113]: df_with_dummy
Salida[113]:
  dato1 clave_a clave_b clave_c 0 0
0      1      0      1
1      1      0      1
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0

```

Si una fila en un DataFrame pertenece a varias categorías, las cosas son un poco más complicadas. Veamos el conjunto de datos MovieLens 1M, que se investiga con más detalle en [el Capítulo 14](#):

```
En [114]: mnames = ['movie_id', 'title', 'genres']
```

```
En [115]: películas = pd.read_table('conjuntos de datos/lentes de película/películas.dat', sep='::',
.....:                               encabezado=Ninguno, nombres=mnombres)
```

```
En [116]: películas[:10]
```

```
Salida[116]:
```

	movie_id	título	géneros
		Toy Story (1995)	Animación Infantil Comedia
0 1	1 2	Jumanji (1995)	Aventura Infantil Fantasía
2	3	Viejos gruñones (1995)	Comedia Romance
3	4	Esperando para exhalar (1995)	Comedia Drama
4	5	Padre de la novia Parte II (1995)	Comedia
5		Calor (1995)	Acción Crimen Suspense
6		sabina (1995)	Comedia Romance
7	8	Tom y Huck (1995)	Aventura Infantil
8	9	Muerte súbita (1995)	Acción
9	10	Ojo dorado (1995)	Acción Aventura Suspense

Agregar variables indicadoras para cada género requiere un poco de discusión. Primero, extraemos la lista de géneros únicos en el conjunto de datos:

```
En [117]: todos los géneros = []
```

```
En [118]: para x en películas.géneros:
.....:         all_genres.extend(x.split('|'))
```

```
En [119]: géneros = pd.unique(all_genres)
```

Ahora tenemos:

```
En [120]: géneros
```

```
Out[120]:
```

```
array(['Animación', 'Infantil', 'Comedia', 'Aventura', 'Fantasía',
      'Romance', 'Drama', 'Acción', 'Crimen', 'Suspense', 'Terror',
      'Ciencia ficción', 'Documental', 'Guerra', 'Musical', 'Misterio', 'Cine negro',
      'Occidental'], dtype=objeto)
```

Una forma de construir el DataFrame del indicador es comenzar con un DataFrame de todos ceros:

```
En [121]: zero_matrix = np.zeros((len(películas), len(géneros)))
```

```
En [122]: dummies = pd.DataFrame(zero_matrix, column=genres)
```

Ahora, itere a través de cada película y configure las entradas en cada fila de dummies en 1. Para hacer esto, usamos `dummies.columns` para calcular los índices de columna para cada género:

```
En [123]: gen = películas.géneros[0]

En [124]: gen.split('|')
Out[124]: ['Animación', 'Infantil', 'Comedia']

En [125]: dummies.columns.get_indexer(gen.split('|'))
Salida[125]: matriz([0, 1, 2])
```

Luego, podemos usar .iloc para establecer valores basados en estos índices:

```
En [126]: para i, gen en enumerate(movies.genres):
.....:     índices = maniquíes.columns.get_indexer(gen.split('|')) maniquíes.iloc[i,
.....:     índices] = 1
.....:
```

Entonces, como antes, puedes combinar esto con películas:

```
En [127]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

En [128]: películas_windic.iloc[0]
Salida[128]:
movie_id                                1
título                                Historia del juguete (1995)
géneros                                Animación|Infantil|Comedia 1
Género_animación                                1
Género_Infantil                                1
Género_Comedia                                1
género_aventura                                0
Género_Fantasia                                0
Género_Romance                                0
Género_Drama                                0
...
Género_Crimen                                0
género_suspenso                                0
género_terror                                0
Género_Ciencia ficción                        0
Género_Documental                            0
Género_Guerra                                0
Genero_Musical                                0
género_misterio                                0
Género_Cine-Noir                              0
género_occidental                             0
Nombre: 0, Longitud: 21, tipo de d: objeto
```



Para datos mucho más grandes, este método de construcción de variables indicadoras con pertenencia múltiple no es especialmente rápido. Sería mejor escribir una función de nivel inferior que escriba directamente en una matriz NumPy y luego envolver el resultado en un DataFrame.

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

```

En [129]: np.random.seed(12345)

En [130]: valores = np.random.rand(10)

In [131]: valores
Out[131]:
array([ 0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532, 0.7489, 0.6536])

En [132]: contenedores = [0, 0.2, 0.4, 0.6, 0.8, 1]

En [133]: pd.get_dummies(pd.cut(valores, bins))
Salida[133]:
      (0,0, 0,2] (0,2, 0,4] (0,4, 0,6] (0,6, 0,8] (0,8, 1,0] 1
0              0          0          0          0          0
1              0          1          0          0          0
2              1          0          0          0          0
3              0          1          0          0          0
4              0          0          0          0          0
5              0          0          1 1          0          0
6              0          0          0          0          0
7              0          0          0          1          0
8              0          0          0          1          0
9              0          0          0          1          0

```

Configuramos la semilla aleatoria con `numpy.random.seed` para que el ejemplo sea determinista. Veremos nuevamente `pandas.get_dummies` más adelante en el libro.

7.3 Manipulación de cadenas

Python ha sido durante mucho tiempo un lenguaje popular de manipulación de datos sin procesar, en parte debido a su facilidad de uso para el procesamiento de cadenas y texto. La mayoría de las operaciones de texto se simplifican con los métodos integrados del objeto de cadena. Para combinaciones de patrones y manipulaciones de texto más complejas, es posible que se necesiten expresiones regulares. `pandas` se suma a la mezcla al permitirle aplicar cadenas y expresiones regulares de manera concisa en conjuntos completos de datos, además de manejar la molestia de los datos faltantes.

Métodos de objetos de cadena En

muchas aplicaciones de secuencias de comandos y secuencias de comandos, los métodos de cadena integrados son suficientes. Como ejemplo, una cadena separada por comas se puede dividir en partes con `split`:

```

En [134]: val = 'a, b, guido'

En [135]: val.split(',')
Fuera[135]: ['a', 'b', 'guido']

```

`split` a menudo se combina con `strip` para recortar espacios en blanco (incluidos los saltos de línea):

```
En [136]: piezas = [x.strip() for x in val.split(',')]
```

```
En [137]: piezas
```

```
Fuera[137]: ['a', 'b', 'guido']
```

Estas subcadenas se pueden concatenar junto con un delimitador de dos puntos usando la suma:

```
En [138]: primero, segundo, tercero = piezas
```

```
En [139]: primero + '::' + segundo + '::' + tercero
```

```
Salida[139]: 'a::b::guido'
```

Pero este no es un método genérico práctico. Una forma más rápida y más pitónica es pasar una lista o tupla al método de unión en la cadena '::':

```
En [140]: '::'.join(piezas)
```

```
Salida[140]: 'a::b::guido'
```

Otros métodos se ocupan de localizar subcadenas. Usar la palabra clave `in` de Python es la mejor manera de detectar una subcadena, aunque también se pueden usar `index` y `find`:

```
En [141]: 'guido' in val
```

```
Salida[141]: Verdadero
```

```
En [142]: val.index(',')
```

```
Salida[142]: 1
```

```
En [143]: val.find(',')
```

```
Salida[143]: -1
```

Tenga en cuenta que la diferencia entre buscar e indexar es que `index` genera una excepción si no se encuentra la cadena (en lugar de devolver `-1`):

```
En [144]: val.index(',')
```

```
ValueError
```

Rastreo (última llamada más reciente)

```
<python-input-144-280f8b2856ce> en <módulo>() ----> 1
```

```
val.index(',')
```

```
ValueError: subcadena no encontrada
```

De manera relacionada, `count` devuelve el número de ocurrencias de una subcadena en particular:

```
En [145]: val.cuenta(',')
```

```
Salida[145]: 2
```

`replace` sustituirá las ocurrencias de un patrón por otro. También se usa comúnmente para eliminar patrones, pasando una cadena vacía:

```
En [146]: val.replace(',', '::')
```

```
Salida[146]: 'a::b::guido'
```

```
En [147]: val.replace(',', '')
```

```
Fuera[147]: 'ab guido'
```

Consulte la [Tabla 7-3](#) para obtener una lista de algunos de los métodos de cadena de Python.

Las expresiones regulares también se pueden usar con muchas de estas operaciones, como verá.

Tabla 7-3. Métodos de cadena integrados de Python

Argumento	Descripción
contar	Devuelve el número de ocurrencias no superpuestas de subcadena en la cadena.
termina con	Devuelve True si la cadena termina con sufijo.
comienza con	Devuelve verdadero si la cadena comienza con un prefijo.
unirse	Use una cadena como delimitador para concatenar una secuencia de otras cadenas.
índice	Devuelve la posición del primer carácter en la subcadena si se encuentra en la cadena; genera ValueError si no se encuentra.
encontrar	Devuelve la posición del primer carácter de la primera aparición de subcadena en la cadena; como índice, pero devuelve -1 si no se encuentra.
encontrar	Devuelve la posición del primer carácter de la última aparición de subcadena en la cadena; devuelve -1 si no se encuentra.
reemplazar	Reemplace las ocurrencias de cadena con otra cadena.
tira, rstrip, lstrip	Recorte los espacios en blanco, incluidos los saltos de línea; equivalente a x.strip() (y rstrip, lstrip, respectivamente) para cada elemento.
dividir	Divide la cadena en una lista de subcadenas utilizando el delimitador pasado.
más bajo	Convierte los caracteres del alfabeto a minúsculas.
superior	Convierte los caracteres del alfabeto a mayúsculas.
plegable	Convierta los caracteres a minúsculas y convierta cualquier combinación de caracteres variables específica de la región a una forma comparable común.
solo, solo	Justificación a la izquierda o justificación a la derecha, respectivamente; rellenar el lado opuesto de la cadena con espacios (o algún otro carácter de relleno) para devolver una cadena con un ancho mínimo.

Expresiones regulares Las

expresiones regulares proporcionan una forma flexible de buscar o hacer coincidir patrones de cadenas (a menudo más complejos) en el texto. Una sola expresión, comúnmente llamada expresión regular, es una cadena formada de acuerdo con el lenguaje de expresión regular. El módulo re incorporado de Python es responsable de aplicar expresiones regulares a las cadenas; Voy a dar una serie de ejemplos de su uso aquí.



El arte de escribir expresiones regulares podría ser un capítulo propio y, por lo tanto, está fuera del alcance del libro. Hay muchos tutoriales y referencias excelentes disponibles en Internet y en otros libros.

Las funciones del módulo re se dividen en tres categorías: coincidencia de patrones, sustitución y división. Naturalmente, todos estos están relacionados; una expresión regular describe un patrón para ubicar en el texto, que luego se puede usar para muchos propósitos. Veamos un ejemplo sencillo:

Supongamos que queremos dividir una cadena con un número variable de caracteres de espacio en blanco (tabulaciones, espacios y saltos de línea). La expresión regular que describe uno o más caracteres de espacio en blanco es `\s+`:

```
En [148]: importar re
```

```
En [149]: texto = "foo      bar\t baz \tqux"
```

```
En [150]: re.split("\s+", texto)
```

```
Salida[150]: ['foo', 'bar', 'baz', 'qux']
```

Cuando llama a `re.split("\s+", texto)`, primero se compila la expresión regular y luego se llama a su método de división en el texto pasado. Puede compilar la expresión regular usted mismo con `re.compile`, formando un objeto de expresión regular reutilizable:

```
En [151]: expresión regular = re.compile("\s+")
```

```
En [152]: regex.split(texto)
```

```
Salida[152]: ['foo', 'bar', 'baz', 'qux']
```

Si, en cambio, desea obtener una lista de todos los patrones que coinciden con la expresión regular, puede usar el método `findall`:

```
En [153]: regex.findall(texto)
```

```
Salida[153]: [' ', '\t', '\t']
```



Para evitar escapes no deseados con `\` en una expresión regular, use literales de cadena sin procesar como `'C\x'` en lugar del equivalente `'C\\x'`.

Se recomienda encarecidamente crear un objeto `regex` con `re.compile` si tiene la intención de aplicar la misma expresión a muchas cadenas; hacerlo ahorrará ciclos de CPU.

`match` y `search` están estrechamente relacionados con `findall`. Mientras que `findall` devuelve todas las coincidencias en una cadena, la búsqueda devuelve solo la primera coincidencia. De manera más rígida, haga coincidir solo las coincidencias al comienzo de la cadena. Como un ejemplo menos trivial, consideremos 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
```

```
patrón = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[AZ]{2,4}'
```

```
# re.IGNORECASE hace que la expresión regular no distinga entre
```

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

El uso de `findall` en el texto produce una lista de las direcciones de correo electrónico:

```
En [155]: regex.findall(texto)
Salida[155]:
['dave@google.com',
'steve@gmail.com',
'rob@gmail.com',
'ryan@yahoo.com']
```

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

```
En [156]: m = expresión regular.buscar(texto)

In [157]: m
Out[157]: <_sre.SRE_Match object; span=(5, 20), partido='dave@google.com'>

En [158]: texto[m.start():m.end()]
Salida[158]: 'dave@google.com'
```

regex.match devuelve Ninguno, ya que solo coincidirá si el patrón ocurre al comienzo de la cadena:

```
En [159]: imprimir (regex.match (texto))
Ninguno
```

De manera relacionada, sub devolverá una nueva cadena con ocurrencias del patrón reemplazadas por una nueva cadena:

```
En [160]: print(regex.sub('ELIMINADO', texto))
Dave ELIMINADO
Steve ELIMINADO
Rob ELIMINADO
Ryan ELIMINADO
```

Suponga que desea encontrar direcciones de correo electrónico y, al mismo tiempo, segmentar cada dirección en sus tres componentes: nombre de usuario, nombre de dominio y sufijo de dominio. Para ello, pon entre paréntesis las partes del patrón a segmentar:

```
En [161]: patrón = r'([A-Z0-9._%+-.]+)@([A-Z0-9.-]+\.[A-Z]{2,4})'

En [162]: expresión regular = re.compile(patrón, banderas=re.IGNORECASE)
```

Un objeto de coincidencia producido por esta expresión regular modificada devuelve una tupla de los componentes del patrón con su método de grupos :

```
En [163]: m = regex.match('wesm@bright.net')

En [164]: m.groups()
Out[164]: ('wesm', 'bright', 'net')
```

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

```
En [165]: regex.findall(texto)
Fuera[165]:
```

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

sub también tiene acceso a grupos en cada partido usando símbolos especiales como \1 y \2. El símbolo \1 corresponde al primer grupo emparejado, \2 corresponde al segundo, y así sucesivamente:

```
En [166]: print(regex.sub(r'Nombre de usuario: \1, Dominio: \2, Sufijo: \3', texto))  
Dave Nombre de usuario: dave, Dominio: google, Sufijo: com Steve  
Nombre de usuario: steve, Dominio: gmail, Sufijo: com Rob Nombre  
de usuario: rob, Dominio: gmail, Sufijo: com Ryan Nombre de  
usuario: ryan, Dominio: yahoo, Sufijo: com
```

Hay mucho más en las expresiones regulares en Python, la mayoría de las cuales están fuera del alcance del libro. La Tabla 7-4 proporciona un breve resumen.

Tabla 7-4. Métodos de expresiones regulares

Argumento	Descripción
encuentra todos	Devuelve todos los patrones coincidentes que no se superponen en una cadena como una lista
finditer	Como findall, pero devuelve un iterador
fósforo	Haga coincidir el patrón al comienzo de la cadena y, opcionalmente, segmente los componentes del patrón en grupos; si el patrón coincide, devuelve un objeto de coincidencia y, de lo contrario, Ninguno
buscar	Escanee la cadena en busca de coincidencias con el patrón; devolver un objeto de coincidencia si es así; a diferencia de la coincidencia, la coincidencia puede estar en cualquier parte de la cadena en lugar de solo al principio
dividir	Rompe la cuerda en pedazos en cada ocurrencia del patrón
sub, subn	Reemplaza todas (sub) o las primeras n apariciones (subn) del patrón en la cadena con la expresión de reemplazo; use los símbolos \1, \2, ... para hacer referenciaa los elementos del grupo de coincidencias en la cadena de reemplazo

Funciones de cadena vectorizadas en pandas La limpieza

de un conjunto de datos desordenado para el análisis a menudo requiere una gran cantidad de manipulación y regularización de cadenas. Para complicar las cosas, una columna que contiene cadenas a veces tendrá datos faltantes:

```
En [167]: datos = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com', 'Rob': 'rob@gmail.com',  
.....:           'Wes': np. yaya}  
  
En [168]: datos = pd.Series(datos)  
  
Entrada [169]:  
datos  
Salida[169]:  
dave@google.com Dave Rob  
Steve          rob@gmail.com  
steve@gmail.com NaN  
  
Wes dtype: objeto
```

```

En [170]: datos.isnull()
Fuera[170]:
           Falso Dave
Robar      FALSO
Steve      FALSO
Wes        Verdadero
dtype: booleano

```

Puede aplicar cadenas y métodos de expresiones regulares (pasar una lambda u otra función) a cada valor usando `data.map`, pero fallará en los valores NA (nulos). Para hacer frente a esto, Series tiene métodos orientados a arreglos para operaciones de cadenas que omiten valores NA. Se accede a estos a través del atributo `str` de Series ; por ejemplo, podríamos verificar si cada dirección de correo electrónico tiene 'gmail' con `str.contains`:

```

En [171]: data.str.contains('gmail')
Fuera[171]:
Dave Falso Rob
Verdadero
Steve      Verdadero
Wes        Yaya
tipo: objeto

```

También se pueden usar expresiones regulares, junto con cualquier opción `re` como `IGNORECASE`:

```

En [172]: patrón Salida[172]:
'([A-Z0-9_%-]+)@([A-Z0-9-]+)\.([AZ]{2, 4})'

En [173]: data.str.findall(patrón, banderas=re.IGNORECASE)
Fuera[173]:
Dave [(dave, google, com)]
robar [(robar, gmail, com)] [(steve, gmail, com)]
Steve
Wes                                     Yaya
tipo: objeto

```

Hay un par de formas de recuperar elementos vectorizados. Utilice `str.get` o `index` en el atributo `str` :

```

En [174]: coincidencias = data.str.match(patrón, banderas=re.IGNORECASE)

En [175]: coincidencias
Fuera[175]:
dave      Verdadero
Robar     Verdadero
Steve     Verdadero
Wes       NaN
dtype: objeto

```

Para acceder a los elementos de las listas incrustadas, podemos pasar un índice a cualquiera de estas funciones:

```

En [176]: partidos.str.get(1)
Fuera[176]:

```

```
dave      Yaya
Robar     Yaya
Steve NaN
Wes       Tipo
de NaN: float64

En [177]: partidos.str[0]
Fuera[177]:
Dave NaN Rob
NaN
Steve NaN
Wes       Yaya
tipo: float64
```

De manera similar, puede dividir cadenas usando esta sintaxis:

```
En [178]: datos.str[:5]
Fuera[178]:
Dave dave@ Rob
rob@g steve
Steve
Wes       Yaya
tipo: objeto
```

Consulte la [Tabla 7-5](#) para obtener más métodos de cadena de pandas.

Tabla 7-5. Listado parcial de métodos de cadenas vectorizadas

Método	Descripción
gato	Concatenar cadenas elemento-sabio con delimitador opcional
contiene	Devolver una matriz booleana si cada cadena contiene patrón/
contar	regex Contar ocurrencias de
extracto	patrón Usar una expresión regular con grupos para extraer una o más cadenas de una Serie de cadenas; el resultado será un DataFrame con una columna por
termina con	grupo Equivalente a x.endswith(patrón) para cada elemento
comienza con	Equivalente a x.startswith(patrón) para cada elemento Calcular la
encuentra todos	lista de todas las apariciones de patrón/regex para cada cadena
conseguir	Índice en cada elemento (recuperar i-ésimo elemento)
isalnum	Equivalente a str.alnum integrado
isalfa	Equivalente a str.isalpha integrado
isdecimal	Equivalente a str.isdecimal integrado
esdigito	Equivalente a str.isdigit incorporado
es bajo	Equivalente a str.islower incorporado
isnumeric	Equivalente a str.isnumeric incorporado
essuperior	Equivalente a str.isupper incorporado
unirse	Unir cadenas en cada elemento de la serie con el separador pasado
Len	Calcular la longitud de cada cadena
mayúsculas y minúsculas	Convertir mayúsculas y minúsculas; equivalente a x.lower() o x.upper() para cada elemento

Método	Descripción
fósforo	Use re.match con la expresión regular pasada en cada elemento, devolviendo los grupos coincidentes como una lista Agregue
elimina	espacios en blanco a la izquierda, a la derecha o a ambos lados
centro	de las cadenas Equivalente a pad(side='both')
repetir	Valores duplicados (p. ej., s.str.repeat(3) es equivalente a x * 3 para cada cadena)
reemplazar	Reemplazar ocurrencias de patrón/regex con alguna otra cadena Cortar cada
rebanada	cadena en la Serie Dividir cadenas
dividir	en delimitador o expresión regular Recortar espacios en
banda	blanco de ambos lados, incluyendo líneas nuevas Recortar espacios
rstrip lstrip	en blanco en el lado derecho Recortar espacios en blanco en el lado izquierdo

7.4 Conclusión

La preparación eficaz de datos puede mejorar significativamente la productividad al permitirle dedicar más tiempo a analizar datos y menos tiempo a prepararlos para el análisis. Hemos explorado una serie de herramientas en este capítulo, pero la cobertura aquí no es exhaustiva. En el próximo capítulo, exploraremos la funcionalidad de unión y agrupación de pandas.

Gestión de datos: unir, combinar y remodelar

En muchas aplicaciones, los datos pueden distribuirse entre varios archivos o bases de datos o pueden organizarse en una forma que no es fácil de analizar. Este capítulo se centra en las herramientas para ayudar a combinar, unir y reorganizar datos.

Primero, presento el concepto de indexación jerárquica en pandas, que se usa ampliamente en algunas de estas operaciones. Luego profundizo en las manipulaciones de datos particulares.

Puede ver varios usos aplicados de estas herramientas en el [Capítulo 14](#).

8.1 Indexación jerárquica

La indexación jerárquica es una característica importante de pandas que le permite tener múltiples (dos o más) niveles de índice en un eje. De manera un tanto abstracta, proporciona una forma de trabajar con datos de dimensiones superiores en una forma de dimensiones inferiores. Comencemos con un ejemplo simple; cree una serie con una lista de listas (o matrices) como índice:

```
En [9]: datos = pd.Series(np.random.randn(9),
...:                      índice=[['a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
...:                              [1, 2, 3, 1, 3, 1, 2, 2, 3]])
```

Entrada [10]:

datos

Salida[10]: a 1 -0.204708

2 0.478943

3 -0.519439 b 1

-0.555730 3 1.965781

do 1 1.393406

2 0.092908

re 2 0.281746

```

3      0.769023
tipo: float64

```

Lo que está viendo es una vista embellecida de una serie con un índice múltiple como índice. Los "espacios en blanco" en la visualización del índice significan "usar la etiqueta directamente arriba":

```

Entrada [11]: data.index
Salida[11]:
MultiIndex(niveles=[['a', 'b', 'c', 'd'], [1, 2, 3]],
            etiquetas=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])

```

Con un objeto indexado jerárquicamente, es posible la llamada indexación parcial, lo que le permite seleccionar subconjuntos de datos de manera concisa:

```

En [12]: datos['b']
Salida[12]:
1 -0.555730 3
1.965781 tipo de
d: float64

```

```

En [13]: datos['b':'c']
Salida[13]:
b 1 -0.555730
   3      1.965781
do 1      1.393406
   2      0.092908
tipo: float64

```

```

En [14]: data.loc[['b', 'd']]
Salida[14]:
b 1 -0.555730
   3      1.965781
3 días 2      0.281746
   3      0.769023
tipo: float64

```

La selección es incluso posible desde un nivel "interior":

```

En [15]: data.loc[:, 2]
Fuera[15]:
a      0.478943
      0.092908
c d 0.281746
dtipo: float64

```

La indexación jerárquica juega un papel importante en la remodelación de datos y operaciones basadas en grupos, como la formación de una tabla dinámica. Por ejemplo, podría reorganizar los datos en un DataFrame usando su método de desapilar :

```

En [16]: datos.unstack()
Fuera[16]:
      1      2      3
a -0.204708 0.478943 -0.519439 b
   -0.555730      NaN 1.965781

```

```
c 1.393406 0.092908 d Yaya
NaN 0,281746 0,769023
```

La operación inversa de desapilar es apilar:

```
En [17]: datos.unstack().stack()
Salida[17]:
a 1 -0.204708
   2    0.478943
   3 -0.519439
b 1 -0.555730 3 1.965781

do 1    1.393406
    0.092908
2 días 2    0.281746
    3 0.769023 dtipo:
float64
```

apilar y desapilar se explorará con más detalle más adelante en este capítulo.

Con un DataFrame, cualquiera de los ejes puede tener un índice jerárquico:

```
En [18]: marco = pd.DataFrame(np.arange(12).reshape((4, 3)), index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]], columns=[['Ohio', 'Ohio', 'Colorado'], ['Verde', 'Rojo', 'Verde']])

En [19]: marco
Fuera[19]:
      Ohio Colorado
Verde rojo Verde
un 1 2      0      1      2
      3      4      5
segundo 1    6      7      8
      2    9 10     11
```

Los niveles jerárquicos pueden tener nombres (como cadenas o cualquier objeto de Python). Si es así, aparecerán en la salida de la consola:

```
En [20]: frame.index.names = ['clave1', 'clave2']

En [21]: frame.columns.names = ['estado', 'color']

En [22]: cuadro
Out[22]:
color      Ohio Colorado
de Verde rojo Verde
estado tecla 1 tecla 2
a      1      0 1      2
      2      3 4 6 7      5
b      1      8
      2      9 10     11
```



Tenga cuidado de distinguir los nombres de índice 'estado' y 'color' de las etiquetas de las filas.

Con la indexación de columnas parciales, puede seleccionar grupos de columnas de manera similar:

```
En [23]: cuadro['Ohio']
Salida[23]:
color           Verde rojo
clave1 clave2
a      1          0      1
      2          3      4
b      1          6      7
      2          9 10
```

Un MultiIndex puede crearse por sí mismo y luego reutilizarse; las columnas de la anterior DataFrame con nombres de nivel podría crearse así:

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Verde', 'Rojo', 'Verde']],
                        nombres=['estado', 'color'])
```

Reordenación y clasificación de niveles

A veces necesitará reorganizar el orden de los niveles en un eje o clasificar los datos por los valores en un nivel específico. El nivel de intercambio toma dos números o nombres de nivel y devuelve un nuevo objeto con los niveles intercambiados (pero los datos no se modifican):

```
En [24]: frame.swaplevel('key1', 'key2')
Out[24]:
color           Ohio           Colorado
de           Verde rojo           Verde
estado key2 key1
1      a      0 1 3 4      2
2      a      5
1      6 7      8
2      bb  9 10      11
```

`sort_index`, por otro lado, ordena los datos usando solo los valores en un solo nivel. Al intercambiar niveles, no es raro usar también `sort_index` para que el resultado se ordene lexicográficamente por el nivel indicado:

```
En [25]: frame.sort_index(nivel=1)
Out[25]:
color           Ohio           Colorado
de           Verde rojo           Verde
estado tecla 1 tecla 2
a      1      0 1      2
b      1      6 7 3 4      8
a      2      5
```

```
b      2      9 10      11
```

```
En [26]: frame.swaplevel(0, 1).sort_index(level=0)
```

```
Out[26]:
```

```
color      Ohio      Colorado
de      Verde rojo      Verde
```

```
estado key2 key1
```

```
1      a      0 1      2
      b      6 7 3 4      8
2      a      5
      b      9 10      11
```



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

Estadísticas de resumen por nivel

Muchas estadísticas descriptivas y de resumen en `DataFrame` y `Series` tienen una opción de nivel en la que puede especificar el nivel por el que desea agregar en un eje en particular. Considere el `DataFrame` anterior; podemos agregar por nivel en las filas o columnas de esta manera:

```
En [27]: frame.sum(level='key2')
```

```
Salida [27]:
```

```
estado de Ohio      Colorado
color verde rojo key2      Verde
```

```
1      6 8 12      10
2      14      16
```

```
En [28]: frame.sum(nivel='color', eje=1)
```

```
Salida[28]:
```

```
tecla      Verde rojo
de color 1 tecla 2
a      1      2      1
      2      8      4
b      1      14      7
      2      20 10
```

Debajo del capó, esto utiliza la maquinaria `groupby` de los pandas, que se discutirá con más detalle más adelante en el libro.

Indexación con las columnas de un `DataFrame`

No es inusual querer usar una o más columnas de un `DataFrame` como índice de fila; como alternativa, es posible que desee mover el índice de la fila a las columnas del `DataFrame`. Aquí hay un marco de datos de ejemplo:

```
En [29]: cuadro = pd.DataFrame({'a': rango(7), 'b': rango(7, 0, -1), 'c': ['uno', 'uno', 'uno ',
.....:                                'dos', 'dos', 'dos', 'dos'], 'd': [0, 1, 2, 0, 1, 2, 3]})
.....:
.....:
```

```
En [30]: cuadro
```

```
Salida[30]:
      a b      cd
0 0 7 uno 0 1 1 6
uno 1
2 2 5 uno 2
3 3 4 dos 0
4 4 3 dos 1
5 5 2 dos 2 6 6 1
dos 3
```

La función `set_index` de `DataFrame` creará un nuevo `DataFrame` utilizando una o más de sus columnas como índice:

```
En [31]: cuadro2 = cuadro.set_index(['c', 'd'])
```

```
En [32]: fotograma2
```

```
Salida[32]:
      ab
c      d
uno 0 0 7
      1 1 6
      2 2 5
dos 0 3 4 1 4
      3
      2 5 2
      3 6 1
```

De forma predeterminada, las columnas se eliminan del `DataFrame`, aunque puede dejarlas en:

```
En [33]: frame.set_index(['c', 'd'], soltar=False)
```

```
Salida[33]:
      ab      cd
c      d
uno 0 0 7 uno 0
      1 1 6 uno 1 2 2 5
      uno 2
dos 0 3 4 dos 0
      1 4 3 dos 1
      2 5 2 dos 2
      3 6 1 dos 3
```

`reset_index`, por otro lado, hace lo contrario de `set_index`; los niveles de índice jerárquico se mueven a las columnas:

```

En [34]: frame2.reset_index()
Salida[34]:
      cdab
0 uno 0 0 7 1 uno 1
1 6
2 uno 2 2 5
3 dos 0 3 4
4 dos 1 4 3
5 dos 2 5 2 6 dos 3
6 1

```

8.2 Combinación y fusión de conjuntos de datos

Los datos contenidos en los objetos pandas se pueden combinar de varias maneras:

- `pandas.merge` conecta filas en DataFrames en función de una o más claves. Esto resultará familiar para los usuarios de SQL u otras bases de datos relacionales, ya que implementa operaciones de combinación de bases de datos.
- `pandas.concat` concatena o “apila” objetos a lo largo de un eje.
- El método de instancia `combine_first` permite empalmar datos superpuestos para completar los valores que faltan en un objeto con valores de otro.

Abordaré cada uno de ellos y daré una serie de ejemplos. Se utilizarán en ejemplos a lo largo del resto del libro.

Uniones de marcos de datos de estilo de base de datos

Las operaciones de combinación o combinación combinan conjuntos de datos al vincular filas usando una o más claves. Estas operaciones son fundamentales para las bases de datos relacionales (por ejemplo, basadas en SQL). La función de combinación en pandas es el principal punto de entrada para usar estos algoritmos en sus datos.

Comencemos con un ejemplo simple:

```

En [35]: df1 = pd.DataFrame({'clave': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
.....:                      'datos1': rango(7)})

En [36]: df2 = pd.DataFrame({'clave': ['a', 'b', 'd'], 'datos2': rango(3)})
.....:

En [37]: df1
Salida[37]:
      tecla data1 b
0
1      0 b
2      2  a
3      3  c
4      4  a
5      5  a

```

```
6      6      b
```

```
En [38]: df2
```

```
Salida[38]:
```

```
      tecla data2 0
0          a
1          1
2          2      bd
```

Este es un ejemplo de una unión de muchos a uno; los datos en df1 tienen múltiples filas etiquetadas como a y b, mientras que df2 tiene solo una fila para cada valor en la columna clave. Llamando a merge con estos objetos obtenemos:

```
En [39]: pd.merge(df1, df2)
```

```
Salida[39]:
```

```
      datos1 clave datos2 b
0          0      1      b
1          1          1
2          6      b      1
3          2      a      0
4          4      a      0
5          5      a      0
```

Tenga en cuenta que no especificué en qué columna unirse. Si no se especifica esa información, merge utiliza los nombres de las columnas superpuestas como claves. Sin embargo, es una buena práctica especificar explícitamente:

```
En [40]: pd.merge(df1, df2, on='clave')
```

```
Salida[40]:
```

```
      datos1 clave datos2 b
0          0          1
1          1          1
2          6      bb      1
3          2      a      0
4          4      a      0
5          5      a      0
```

Si los nombres de las columnas son diferentes en cada objeto, puede especificarlos por separado:

```
En [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
.....:                      'datos1': range(7)})
```

```
En [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'], 'data2': range(3)})
.....:
```

```
En [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

```
Salida[43]:
```

```
      datos1 lkey datos2 rkey b 1 bb
0          0
1          1          1
2          6      bb b      1
3          2      a      0      a
```

```
4      4      a      0      a
5      5      a      0      a
```

Puede notar que los valores 'c' y 'd' y los datos asociados faltan en el resultado. Por defecto, merge hace una unión 'interna' ; las claves en el resultado son la intersección, o el conjunto común que se encuentra en ambas tablas. Otras opciones posibles son 'izquierda', 'derecha' y 'exterior'. La unión externa toma la unión de las claves, combinando el efecto de aplicar las uniones izquierda y derecha:

```
En [44]: pd.merge(df1, df2, how='outer')
Salida[44]:
  datos1 clave datos2 0.0
0      b 1.0
1      1,0      1,0
2      segundo      1,0
3      6,0 segundo 2,0 un0
4      4.0 un      0.0
5      5.0 un      0.0
6      3.0 c NaN      NaN
7      d      2.0
```

Consulte la [Tabla 8-1](#) para ver un resumen de las opciones de cómo hacerlo.

Tabla 8-1. Diferentes tipos de unión con argumento how

Opción	Comportamiento
'interior'	Usar solo las combinaciones de teclas observadas en ambas tablas
'izquierda'	Usar todas las combinaciones de teclas que se encuentran en la tabla de la izquierda
'derecha'	Usar todas las combinaciones de teclas que se encuentran en la tabla de la derecha
'salida'	Usar todas las combinaciones de teclas observadas en ambas tablas juntas

Las fusiones de muchos a muchos tienen un comportamiento bien definido, aunque no necesariamente intuitivo.

Aquí hay un ejemplo:

```
En [45]: df1 = pd.DataFrame({'clave': ['b', 'b', 'a', 'c', 'a', 'b'], 'datos1': rango(6) })
.....:

En [46]: df2 = pd.DataFrame({'clave': ['a', 'b', 'a', 'b', 'd'], 'datos2': rango(5)})
.....:

En [47]: df1
Salida[47]:
  tecla data1 b
0      0
1      1      b
2      2      a
3      3      c
4      4      a
5      5      b
```

```
En [48]: df2
```

```
Salida[48]:
```

```
tecla data2 0
          1
0 1          —
2    2      a
3    3
4    4      bd
```

```
En [49]: pd.merge(df1, df2, on='key', how='left')
```

```
Salida[49]:
```

```
datos1 clave datos2 b
0      0          1.0
1          3.0
2      0 1          1.0
3      1      bb b  3.0
4      2      a    0.0
5      2      a    2.0
6          c    NaN
7      3 4      a    0.0
8      4      a    2.0
9      5          1.0
10     5      bb    3.0
```

Las uniones de muchos a muchos forman el producto cartesiano de las filas. Dado que había tres filas 'b' en el DataFrame izquierdo y dos en el derecho, hay seis filas 'b' en el resultado. El método de combinación solo afecta a los distintos valores clave que aparecen en el resultado:

```
En [50]: pd.merge(df1, df2, how='inner')
```

```
Salida[50]:
```

```
datos1 clave datos2 0
      b 0 bbb      1
0 1      b      3
2      1      1
3      1      3
4      5      1
5      5      b
6      2      a    3 0
7      2      a    2
8      4      a    0
9      4      a    2
```

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

```
En [51]: izquierda = pd.DataFrame({'key1': ['foo', 'foo', 'bar'], 'key2': ['one', 'two', 'one'],
.....:                             'lval': [1, 2, 3]})
.....:
```

```
En [52]: derecha = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'], 'key2': ['one', 'one', 'one', 'dos'],
.....:                             'rval': [4, 5, 6, 7]})
.....:
```

```
En [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

```
Salida[53]:
   tecla1 tecla2 lval rval 0 foo
one 1,0 4,0
1 foo uno 1,0 5,0 2 foo dos 2,0
NaN 3 bar uno 3,0 6,0 4 bar dos
NaN 7,0
```

Para determinar qué combinaciones de teclas aparecerán en el resultado dependiendo de la elección del método de fusión, piense en las claves múltiples como si formaran una matriz de tuplas que se utilizarán como una sola tecla de unión (aunque en realidad no se implemente de esa manera).



Cuando una columnas sobre columnas, los índices de los objetos DataFrame pasados se descartan.

Una última cuestión a tener en cuenta en las operaciones de combinación es el tratamiento de nombres de columnas superpuestos. Si bien puede abordar la superposición manualmente (consulte la sección anterior sobre cómo cambiar el nombre de las etiquetas de los ejes), la combinación tiene una opción de sufijos para especificar cadenas para agregar a los nombres superpuestos en los objetos DataFrame izquierdo y derecho:

```
En [54]: pd.merge(izquierda, derecha, on='key1')
Salida[54]:
   clave1 clave2_x lval clave2_y rval 0 foo 4
1 foo      uno      1      uno
      uno      1      uno      5
2 foo 3     dos      2      uno      4
foo 4     dos      2      uno      5
bar 5     uno      3      uno      6
bar      uno      3      dos      7

En [55]: pd.merge(izquierda, derecha, on='key1', sufijos=('_left', '_right'))
Salida[55]:
   tecla1 tecla2_izquierda lval tecla2_derecha rval 0
foo 4 1 foo      uno      1      uno
      uno      1      uno      5
2 foo 3     dos      2      uno      4
foo 4     dos      2      uno      5
bar 5     uno      3      uno      6
bar      uno      3      dos      7
```

Consulte la [Tabla 8-2](#) para obtener una referencia de argumento sobre la combinación. Unirse usando el índice de fila de DataFrame es el tema de la siguiente sección.

Tabla 8-2. fusionar argumentos de función

Argumento	Descripción
izquierda	DataFrame que se fusionará en el lado izquierdo.
bien	DataFrame que se fusionará en el lado derecho.
cómo	Uno de 'interior', 'exterior', 'izquierda' o 'derecha'; por defecto es 'interno'.
en	Nombres de columna para unirse. Debe encontrarse en ambos objetos DataFrame. Si no se especifica y no se proporcionan otras claves de combinación, se utilizará la intersección de los nombres de las columnas a la izquierda y a la derecha como claves de combinación.
left_on	Columnas en el marco de datos izquierdo para usar como claves de combinación.
right_on	Análogo a left_on para el DataFrame izquierdo .
left_index	Usa el índice de fila a la izquierda como su clave de combinación (o claves, si es un índice múltiple).
right_index	Análogo a left_index.
clasificar	Ordenar datos combinados lexicográficamente por claves de combinación; Verdadero de forma predeterminada (deshabilitar para obtener un mejor rendimiento en algunos casos en grandes conjuntos de datos).
sufijos	Tupla de valores de cadena para agregar a los nombres de columna en caso de superposición; el valor predeterminado es ('_x', '_y') (por ejemplo, si 'datos' en ambos objetos DataFrame, aparecería como 'datos_x' y 'datos_y' en el resultado).
Copiar	Si es False, evite copiar datos en la estructura de datos resultante en algunos casos excepcionales; por defecto siempre copia.
indicador	Añade una columna especial _merge que indica el origen de cada fila; los valores serán 'solo_izquierda', 'solo_derecha' o 'ambos' según el origen de los datos unidos en cada fila.

Fusionar en el índice

En algunos casos, las claves de fusión en un DataFrame se encontrarán en su índice. En este caso, puede pasar left_index=True o right_index=True (o ambos) para indicar que el índice debe usarse como clave de combinación:

```
En [56]: izquierda1 = pd.DataFrame({'clave': ['a', 'b', 'a', 'a', 'b', 'c'],
.....:                               'valor': rango (6)})
```

```
En [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

```
En [58]: izquierda1
```

```
Salida[58]:
```

	valor clave 0	
0	a	
1	b	1
2	a	2
	-	
3 4		3 4
5	C	5

```
Entrada [59]: right1
```

```
Salida [59]:
```

	group_val 3.5	
a		
b	7.0	

```
En [60]: pd.merge(left1, right1, left_on='key', right_index=True)
```

```
Out[60]:
```

```
valor clave group_val 0 3.5
0 a
2 a 2 3.5
3 a 3 3.5
1 b 1 7.0
4 b 4 7.0
```

Dado que el método de fusión predeterminado es intersectar las claves de combinación, en su lugar puede formar la unión de ellas con una combinación externa:

```
En [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

```
Out[61]:
```

```
valor clave group_val 0 3.5
0 a 2 3.5
2 a
3 a 3 3.5
1 - 1 7.0
4 - 4 7.0
5 C 5 Yaya
```

Con datos indexados jerárquicamente, las cosas son más complicadas, ya que unirse en el índice es implícitamente una combinación de varias claves:

```
En [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
```

```
.....:                                'Nevada', 'Nevada'],
.....:                                'clave2': [2000, 2001, 2002, 2001, 2002], 'datos':
.....:                                np.arange(5.)})
```

```
En [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)), index=[['Nevada',
```

```
.....:                        'Nevada', 'Ohio', 'Ohio',
.....:                        'Ohio', 'Ohio'], [2001,
.....:                        2000, 2000, 2000, 2001, 2002]], columns=['evento1',
.....:                        'evento2'])
```

```
Entrada [64]:
```

```
izquierda
```

```
Salida[64]: datos clave1
```

```
clave2 0 0,0 Ohio 2000 1 1,0
```

```
Ohio 2001 2 2,0 Ohio 2002
```

```
3 3,0 Nevada 2001
```

```
4 4,0 Nevada 2002
```

```
En [65]: derecha
```

```
Fuera[65]:
```

```
evento1 evento2
Nevada 2001 0 1
         2000 2 3
Ohio 2000 4 5
         2000 6 7
```

2001	8	9
2002	10	11

En este caso, debe indicar varias columnas para fusionarlas como una lista (tenga en cuenta el manejo de valores de índice duplicados con `how='outer'`):

```
En [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Salida[66]:
      datos clave1 clave2 evento1 evento2 0 0,0 Ohio
2000 5 0 0,0 Ohio 2000 1 1,0 Ohio 2001 2 2,0 Ohio 2002
                                     6      7
                                     8      9
                                     10     11
3 3.0 Nevada 2001                0      1
```

```
En [67]: pd.merge(izquierda, derecha, izquierda_activada=['tecla1', 'tecla2'],
      ....:                right_index=Verdadero, cómo='exterior')
Salida[67]:
      datos clave1 clave2 evento1 evento2 0 0,0 Ohio
2000 5,0 0 0,0 Ohio 2000 1 1,0 Ohio 2000 2 2,0 Ohio
2002
                                     6.0    7.0
                                     8.0    9.0
                                     10.0   11.0
3 3.0 Nevada 2001 4 4.0                0.0    1.0
Nevada 2002 4 NaN Nevada                NaN    NaN
2000
                                     2.0    3.0
```

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

```
En [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'], columnas=['Ohio',
      ....:                                     'Nevada'])
      ....:

En [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]], index=['b', 'c', 'd', 'e'], columnas=['Missouri',
      ....:                                     'Alabama'])
      ....:

En [70]: izquierda2
Fuera[70]:
      ohio nevada
un 1.0          2.0
c3.0 _          4.0
5.0 _          6.0

En [71]: derecha2
Fuera[71]:
      misuri alabama
b          7.0      8.0
          9.0     10.0
cd         11.0     12.0
mi         13.0     14.0

En [72]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

```
Fuera[72]:
Ohio Nevada Misuri Alabama
un 1.0      2.0      Yaya      Yaya
b NaN c     NaN      7,0      8,0
3,0 d NaN   4.0      9,0      10,0
      Yaya      11.0      12.0
5.0 _       6.0      13.0      14.0
```

DataFrame tiene una instancia de combinación conveniente para fusionar por índice. También se puede usar para combinar muchos objetos DataFrame que tienen índices iguales o similares pero columnas que no se superponen. En el ejemplo anterior, podríamos haber escrito:

```
En [73]: left2.join(right2, how='outer')
Fuera[73]:
Ohio Nevada Misuri Alabama
un 1.0      2.0      Yaya      Yaya
b NaN c     NaN      7,0      8,0
3,0 d NaN   4.0      9,0      10,0
      Yaya      11.0      12.0
5.0 _       6.0      13.0      14.0
```

En parte por motivos heredados (es decir, versiones mucho más antiguas de pandas), el método de combinación de DataFrame realiza una combinación izquierda en las teclas de combinación, preservando exactamente el índice de fila del marco izquierdo. También admite unir el índice del DataFrame pasado en una de las columnas del DataFrame que llama:

```
En [74]: left1.join(right1, on='key')
Salida[74]:
valor clave group_val 3.5
0      a      0
1      b      1      7.0
2      a      2      3.5
      -      3.5
3 4      3 4      7.0
5      C      5      Yaya
```

Por último, para fusiones simples de índice sobre índice, puede pasar una lista de DataFrames para unirse como alternativa al uso de la función concat más general que se describe a continuación. sección:

```
En [75]: otro = pd.DataFrame([ [7., 8.], [9., 10.], [11., 12.], [16., 17.] ], index=['a', 'c', 'e', 'f'], columns=['Nueva York', 'Oregón'])
.....:
.....:

Entrada [76]: otra
Salida[76]:
Nueva York Oregón 7,0
a      8,0 9,0 10,0 11,0
C      12,0
mi
F      16.0      17.0
```

```
En [77]: left2.join(right2, otro)
Fuera[77]:
Ohio Nevada Misuri Alabama Nueva York Oregón a 1,0 2,0 8,0 c 3,0 4,0 10,0
NaN 7,0
9,0 NaN 10,0 9,0
5,0 _ 6,0 13,0 14,0 11,0 12,0

En [78]: left2.join(right2, another, how='outer')
Fuera[78]:
Ohio Nevada Misuri Alabama Nueva York Oregón 8,0
a 1,0 b NaN 2,0 Yaya Yaya 7,0
Yaya 7,0 8,0 Yaya Yaya
c3,0 _ 4,0 9,0 10,0 9,0 10,0
d NaN e NaN 11,0 12,0
5,0 f NaN 6,0 13,0 14,0 NaN 11,0 NaN 12,0
Yaya Yaya Yaya 16,0 17,0
```

Concatenación a lo largo de un eje Otro

tipo de operación de combinación de datos se conoce indistintamente como concatenación, enlace o apilamiento. La función de concatenación de NumPy puede hacer esto con matrices NumPy:

```
En [79]: arr = np.arange(12).reshape((3, 4))

En [80]: arr
Salida[80]:
matriz([[ 0, 1, 2, 3], [ 4, 5, 6, 7], [ 8, 9, 10, 11]])

En [81]: np.concatenate([arr, arr], eje=1)
Salida[81]:
matriz([[ 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 le permite generalizar aún más la concatenación de matrices. En particular, tiene una serie de cosas adicionales en las que pensar:

- Si los objetos están indexados de manera diferente en los otros ejes, ¿debemos combinar los distintos elementos en estos ejes o usar solo los valores compartidos (la intersección)? • ¿Los fragmentos concatenados de datos deben ser identificables en el resultado? ¿objeto?
- ¿El “eje de concatenación” contiene datos que deben conservarse? En muchos casos, es mejor descartar las etiquetas de enteros predeterminadas en un DataFrame durante la concatenación.

La función `concat` en pandas proporciona una forma consistente de abordar cada una de estas preocupaciones. Voy a dar una serie de ejemplos para ilustrar cómo funciona. Supongamos que tenemos tres series sin superposición de índice:

```
En [82]: s1 = pd.Series([0, 1], index=['a', 'b'])
```

```
En [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
```

```
En [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

Llamar a `concat` con estos objetos en una lista une los valores y los índices:

```
En [85]: pd.concat([s1, s2, s3])
```

```
Salida[85]:
```

```
a      0
b      1
c      2
d      3
e      4
f      5
g      6
tipo g: int64
```

Por defecto, `concat` funciona a lo largo del eje = 0, produciendo otra Serie. Si pasa `axis=1`, el resultado será un DataFrame (axis=1 son las columnas):

```
En [86]: pd.concat([s1, s2, s3], eje=1)
```

```
Salida[86]:
```

```
      0      1      2
a  0,0 NaN NaN b  1,0
NaN NaN c NaN  2,0
NaN d NaN  3,0 NaN
e NaN  4,0 NaN
f NaN NaN  5,0 g NaN
NaN  6,0
```

En este caso, no hay superposición en el otro eje, que como puede ver es la unión ordenada (la unión 'externa') de los índices. En su lugar, puede cruzarlos pasando `join='inner'`:

```
En [87]: s4 = pd.concat([s1, s3])
```

```
En [88]: s4
```

```
Salida[88]:
```

```
a      0
      1
bf     5
      6
tipo g: int64
```

```
En [89]: pd.concat([s1, s4], eje=1)
```

```
Fuera[89]:
```

```

      0 1
a 0.0 0 b 1.0
1
f NaN 5 g
NaN 6

En [90]: pd.concat([s1, s4], axis=1, join='inner')
Salida[90]:
      0 1 a
0 0 b 1 1

```

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

Incluso puede especificar los ejes que se utilizarán en los otros ejes con `join_axes`:

```

En [91]: pd.concat([s1, s4], eje=1, join_axes=[['a', 'c', 'b', 'e']])
Salida[91]:
      1
0 a 0.0 0.0
c NaN NaN b
1,0 1,0
e NaN NaN

```

Un problema potencial es que las piezas concatenadas no son identificables en el resultado. Suponga que, en cambio, desea crear un índice jerárquico en el eje de concatenación. Para hacer esto, use el argumento de las claves :

```

En [92]: resultado = pd.concat([s1, s1, s3], claves=['uno', 'dos', 'tres'])

En [93]: resultado
Fuera[93]:
uno      a      0
          b      1
dos      a      0
          b      1
tres f
...
5 6 dtipo: int64

En [94]: resultado.unstack()
Fuera[94]:
          a      b      F
uno      g 0,0 1,0 NaN NaN
dos 0,0 1,0 NaN NaN tres NaN
NaN 5,0 6,0

```

En el caso de combinar Series a lo largo del eje = 1, las claves se convierten en los encabezados de columna de DataFrame:

```

En [95]: pd.concat([s1, s2, s3], eje=1, teclas=['uno', 'dos', 'tres'])
Fuera[95]:

```

```

      Uno, dos, tres
a 0,0 NaN b 1,0      Yaya
NaN                      Yaya
c NaN 2,0 d          Yaya
NaN 3,0              Yaya
e NaN 4,0 f NaN      Yaya
NaN                  5.0
g NaN NaN            6.0

```

La misma lógica se extiende a los objetos DataFrame:

```

En [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'], column=['one', 'two'])
.....:

```

```

En [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
.....:                      columnas=['tres', 'cuatro'])

```

```

En [98]: df1
Fuera[98]:
      uno dos
a      0    1
b      2    3
c      4    5

```

```

En [99]: df2
Salida[99]:
      tres cuatro
a      5      6
c      7      8

```

```

En [100]: pd.concat([df1, df2], eje=1, teclas=['nivel1', 'nivel2'])
Salida[100]:
      nivel1      nivel2
      uno dos tres cuatro 5.0 6.0
a      0    1
b      2    3      NaN NaN
c      4    5      7,0 8,0

```

Si pasa un dictado de objetos en lugar de una lista, las claves del dictado se utilizarán para la opción de claves :

```

En [101]: pd.concat({'nivel1': df1, 'nivel2': df2}, eje=1)
Salida[101]:
      nivel1 nivel2 uno dos tres
      cuatro
a      0    1      5.0 6.0
b      2    3      NaN NaN
c      4    5      7,0 8,0

```

Hay argumentos adicionales que gobiernan cómo se crea el índice jerárquico (vea [la Tabla 8-3](#)). Por ejemplo, podemos nombrar los niveles de eje creados con el argumento de nombres :

```
En [102]: pd.concat([df1, df2], eje=1, teclas=['nivel1', 'nivel2'], nombres=['superior', 'inferior'])
.....:
Salida[102]:
superior nivel1 nivel2 inferior uno
dos tres cuatro
a          0      1      5.0 6.0
b          2      3      NaN NaN
c          4      5      7.0 8.0
```

Una última consideración se refiere a los marcos de datos en los que el índice de fila no contiene
Cualquier dato relevante:

```
En [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columnas=['a', 'b', 'c', 'd'])

En [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columnas=['b', 'd', 'a'])

En [105]: df1
Fuera[105]:
           a          b          c          d
0 1.246435 1.007189 -1.296221 0.274992
1 0.228913 1.352917 0.886429 -2.001637
2 -0.371843 1.669025 -0.438570 -0.539741

En [106]: df2
Fuera[106]:
           b          d          a
0 0.476985 3.248944 -1.021228 1 -0.577087
0.124121 0.302614
```

En este caso, puede pasar ignore_index=True:

```
En [107]: pd.concat([df1, df2], ignore_index=True)
Fuera[107]:
-1.296221 0.274992 0 1.246435 1.007189
0.228913 1.352917 0.886429 -2.001637
2 -0.371843 1.669025 -0.438570 -0.539741
3 -1.021228 0.476985          NaN 3.248944
4 0.302614 -0.577087          NaN 0.124121
```

Tabla 8-3. argumentos de la función concat

objetos de	Descripción
	Lista o dictado de objetos pandas para ser concatenados; este es el único argumento requerido
eje	Eje para concatenar a lo largo; el valor predeterminado es 0 (a lo largo de las filas)
unirse	Ya sea 'interior' o 'exterior' ('exterior' por defecto); ya sea para la intersección (interior) o la unión (exterior) de índices a lo largo de los otros ejes
unir_ejes	Índices específicos para usar con los otros n-1 ejes en lugar de realizar la lógica de unión/intersección
llaves	Valores para asociar con objetos que se concatenan, formando un índice jerárquico a lo largo del eje de concatenación; puede ser una lista o una matriz de valores arbitrarios, una matriz de tuplas o una lista de matrices (si las matrices de varios niveles se pasan en niveles)

Argumento	Descripción
niveles	Índices específicos para usar como nivel o niveles de índice jerárquico si se pasan claves
nombres	Nombres para los niveles jerárquicos creados si las claves y /o los niveles
pasaron por defecto (Falso)	permite duplicados
ignorar_índice	No conserve los índices a lo largo del eje de concatenación, sino que produzca un nuevo índice de rango (longitud total)

Combinación de datos con superposición

Hay otra situación de combinación de datos que no se puede expresar como una operación de combinación o concatenación. Puede tener dos conjuntos de datos cuyos índices se superpongan total o parcialmente. Como ejemplo motivador, considere la función where de NumPy , que realiza el equivalente orientado a matrices de una expresión if-else:

```
En [108]: a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
.....:                  índice=['f', 'e', 'd', 'c', 'b', 'a'])

En [109]: b = pd.Series(np.arange(len(a), dtype=np.float64), index=['f', 'e', 'd', 'c', 'b',
.....:                  'a'])

En [110]: b[-1] = np.nan

Entrada [111]: a
Salida [111]:
f NaN 2.5
mi
d      Yaya
c      3.5
b      4.5

NaN un tipo de d: float64

Entrada [112]:
b Salida
[112]: f 0.0
mi      1.0
d      2.0
c      3.0
b      4.0

NaN un tipo de d: float64

En [113]: np.where(pd.isnull(a), b, a)
Salida[113]: matriz([ 0. , 3.5, 4.5, nan]) , 2.5, 2.
```

Series tiene un método combine_first , que realiza el equivalente de esta operación junto con la lógica de alineación de datos habitual de pandas:

```

En [114]: b[:-2].combine_first(a[2:])
Salida[114]:
a      NaN
b      4.5
c      3.0
d      2.0
mi     1.0
F      0.0
tipo: float64

```

Con DataFrames, `combine_first` hace lo mismo columna por columna, por lo que puede considerarlo como "parchar" los datos que faltan en el objeto que llama con los datos del objeto que pasa:

```

En [115]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan], 'b': [np.nan, 2., np.nan, 6.], 'c':
.....:                      rango (2, 18, 4)})
.....:

```

```

En [116]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.], 'b': [np.nan, 3., 4., 6., 8.]})
.....:

```

```

En [117]: df1
Fuera[117]:
a      antes de Cristo
0 1,0 NaN 2
1 NaN 2,0 6
2 5,0 NaN 10 3 NaN
6,0 14

```

```

En [118]: df2
Fuera[118]:

```

```

b a 0 5,0 NaN
1 4,0 3,0
2 NaN 4,0
3 3,0 6,0
4 7,0 8,0

```

```

En [119]: df1.combine_first(df2)
Fuera[119]:
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

```

8.3 Reformar y pivotar

Hay una serie de operaciones básicas para reorganizar datos tabulares. Estas se conocen alternativamente como operaciones de remodelación o pivote.

Remodelación con indexación jerárquica La

indexación jerárquica proporciona una forma coherente de reorganizar los datos en un DataFrame.

Hay dos acciones principales:

pila

Esto "rota" o gira de las columnas en los datos a las filas

desapilar

Esto gira de las filas a las columnas.

Ilustraré estas operaciones a través de una serie de ejemplos. Considere un marco de datos pequeño con matrices de cadenas como índices de fila y columna:

```
En [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)), index=pd.Index(['Ohio',
.....:                                'Colorado'], name='state'), columns=pd.Index(['uno', 'dos', 'tres'],
.....:                                nombre='número'))
```

```
En [121]: datos
```

```
Salida[121]:
```

número	Uno, dos, tres		
estado			
Ohio	0	1	2
Colorado	3	4	5

El uso del método de pila en estos datos pivota las columnas en las filas, produciendo un Serie:

```
En [122]: resultado = data.stack()
```

```
En [123]: resultado
```

```
Salida[123]:
```

estado	número	
Ohio	uno	0
	dos	1
	tres	2
Colorado	uno dos	
	tres	3 4
	dtype:	5

int64

A partir de una serie indexada jerárquicamente, puede reorganizar los datos de nuevo en una serie de datos. Marco con desapilar:

```
En [124]: resultado.unstack()
```

```
Salida[124]:
```

número	Uno, dos, tres		
estado			
Ohio	0	1	2
Colorado	3	4	5

Por defecto, el nivel más interno está desapilado (lo mismo con la pila). Puede desapilar un nivel diferente pasando un número o nombre de nivel:

```
En [125]: resultado.unstack(0)
Out[125]:
número del estado de Ohio
Colorado
uno
dos      0 1      3 4
tres     2      5
```

```
En [126]: result.unstack('estado')
Salida[126]:
número del estado de Ohio
Colorado
uno      0      3
dos      1      4
tres     2      5
```

Desapilar podría introducir datos faltantes si no se encuentran todos los valores en el nivel en cada uno de los subgrupos:

```
En [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
```

```
En [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
```

```
En [129]: data2 = pd.concat([s1, s2], keys=['one', 'two'])
```

```
En [130]: datos2
```

```
Salida[130]:
```

```
uno un b
```

```
      0 1
C      2
d      3
dos c      4
      d      5
      mi      6
```

```
tipo: int64
```

```
En [131]: data2.unstack()
```

```
Fuera[131]:
```

```
      a      b      C      d      mi
uno 0.0 1.0 2.0 3.0 NaN
dos NaN NaN 4.0 5.0 6.0
```

El apilamiento filtra los datos que faltan de forma predeterminada, por lo que la operación es más fácil de invertir:

```
En [132]: data2.unstack()
```

```
Fuera[132]:
```

```
      a      b      C      d      mi
uno 0.0 1.0 2.0 3.0 NaN dos NaN NaN 4.0
5.0 6.0
```

```

En [133]: datos2.unstack().stack()
Fuera[133]:
uno un      0.0
      b      1.0
      c      2.0
      d      3.0
dos c      4.0
      d      5.0
      mi     6.0

dtipo: float64

En [134]: data2.unstack().stack(dropna=False)
Fuera[134]:
uno un      0.0
      b      1.0
      c      2.0
      d      3.0
      mi     Yaya
dos abdominales Yaya
      Yaya
      c      4.0
      d      5.0
      mi     6.0

tipo: float64

```

Cuando desapila en un DataFrame, el nivel desapilado se convierte en el nivel más bajo en el resultado:

```

En [135]: df = pd.DataFrame({'izquierda': resultado, 'derecha': resultado + 5},
.....:                      columnas=pd.Index(['izquierda', 'derecha'], nombre='lado'))

En [136]: d.f.
Fuera[136]:
lado          izquierda derecha
estado  número
Ohio      uno          0         5
          dos          1         6
          tres         2         7
colorado uno          3         8
          dos          4         9
          tres         5        10

En [137]: df.unstack('estado')
Salida[137]:
lado izquierdo derecho estado Ohio
Colorado Ohio Colorado
número
uno          0          3          5          8
dos          1          4          6          9
tres         2          5          7         10

```

Al llamar a stack, podemos indicar el nombre del eje a apilar:

```
En [138]: df.unstack('estado').stack('lado')
Salida[138]:
estado          colorado ohio
número lado
uno              3          0
            izquierda  8          5
dos              4          1
            derecha   9          6
izquierda derecha 2
tres izquierda derecha 5 10      7
```

Cambio de formato "largo" a "ancho" Una forma

común de almacenar varias series temporales en bases de datos y CSV es en el llamado formato largo o apilado. Carguemos algunos datos de ejemplo y hagamos una pequeña cantidad de disputas de series temporales y otra limpieza de datos:

```
En [139]: datos = pd.read_csv('ejemplos/macrodato.csv')

En [140]: datos.head()
Out[140]:
año trimestre realgdp realcons realinv realgovt realdpi 0 1959.0 1 1959.0 ipc \
            1.0 2710.349 2.0          1707.4 286.898 470.045 1886.9 28.98 1733.7 310.859
            2778.801          481.301 1919.7 29.15
2 1959.0          3.0 2775.488          1751.8 289.226 491.260 1916.4 29.35
3 1959.0          4.0 2785.204          1753.7 299.356 484.052 1931.3 29.37
4 1960.0          1.0 2847.699          1770.5 331.722 462.199 1955.5 29.54

m1 tbilrate unemp 0 139.7          pop inf realint 5.8 177.146
2.82          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

En [141]: periodos = pd.PeriodIndex(año=datos.año, trimestre=datos.trimestre, nombre='fecha')
.....:

En [142]: columnas = pd.Index(['realgdp', 'infl', 'unemp'], name='item')

En [143]: datos = datos.reindex(columnas=columnas)

En [144]: data.index = period.to_timestamp('D', 'end')

En [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

Veremos PeriodIndex un poco más de cerca en el **Capítulo 11**. En resumen, combina las columnas de año y trimestre para crear una especie de tipo de intervalo de tiempo.

Ahora, ldata se parece a:

```
En [146]: ldata[:10]
Fuera[146]:
```

fecha		valor	0 1959-03-31		realgdp
		del artículo			
2710.349	1	1959-03-31	inflación		0.000
5.800	2	1959-03-31	desempleo		5.800
2778.801	4	1959-06-30	inflación		2.340
5.100	5	1959-06-30	desempleo		5.100
2775.488	6	30-09-1959	PIB real		2775.488
5.300	8	30-09-1959	inflación		5.300
2785.204			desempleo		2785.204

Este es el llamado formato largo para series temporales múltiples u otros datos de observación con dos o más claves (aquí, nuestras claves son la fecha y el elemento). Cada fila de la tabla representa una sola observación.

Los datos se almacenan con frecuencia de esta manera en bases de datos relacionales como MySQL, ya que un esquema fijo (nombres de columna y tipos de datos) permite que la cantidad de valores distintos en la columna del elemento cambie a medida que se agregan datos a la tabla. En el ejemplo anterior, la fecha y el elemento normalmente serían las claves principales (en el lenguaje de la base de datos relacional), lo que ofrece integridad relacional y uniones más sencillas. En algunos casos, puede ser más difícil trabajar con los datos en este formato; es posible que prefiera tener un DataFrame que contenga una columna por valor de elemento distinto indexado por marcas de tiempo en la columna de fecha . El método pivote de DataFrame realiza exactamente esta transformación:

```
En [147]: pivoted = ldata.pivot('fecha', 'elemento', 'valor')
```

```
En [148]: pivotedo
```

Salida[148]:

fecha	inflación		realgdp		desempleado
del artículo					
1959-03-31	0.00	2710.349			5.8
1959-06-30	2.34	2778.801	1959-09-30		5.1
2.74	2775.488	1959-12-31	0.27	2785.204	5.3
					5.6
1960-03-31	2.31	2847.699			5.2
1960-06-30	0.14	2834.390			5.2
1960-09-30	2.70	2839.022	1960-12-31		5.6
1.21	2802.616	1961-03-31	-0.40	2819.264	6.3
					6.8
1961-06-30	1.47	2872.005			7.0
...	...		...		...
2007-06-30	2.75	13203.977	2007-09-30		4.5
3.45	13321.109	2007-12-31	6.38		4.7
13391.249					4.8
2008-03-31	2.82	13366.865			4.9
2008-06-30	8.53	13415.266			5.4
2008-09-30	-3.16	13324.600	2008-12-31		6.0
-8.79	13141.920	2009-03-31	0.94		6.9
12925.410					8.1
2009-06-30	3.37	12901.504			9.2

```
2009-09-30 3.56 12990.341          9.6
[203 filas x 3 columnas]
```

Los primeros dos valores pasados son las columnas que se usarán respectivamente como índice de fila y columna, luego, finalmente, una columna de valor opcional para llenar el DataFrame. Suponga que tiene dos columnas de valor que desea remodelar simultáneamente:

```
En [149]: ldata['value2'] = np.random.randn(len(ldata))

En [150]: ldatos[:10]
Out[150]:
valor del elemento de fecha value2 0
1959-03-31 realgdp 2710.349 0.523772 infl
1 1959-03-31          0.000 0.000940
2 1959-03-31 5.800 1.343810 desempleo 3 1959-06-30
realgdp 2778.801 -0.713544 infl 4 1959-06-30 2.340
-0.831154
5 1959-06-30          desempleado 5.100 -2.370232
6 1959-09-30 realgdp 2775.488 -1.860761 7
1959-09-30 infl 2.740 -0.860757 8 1959-09-30 5.300
0.560145 unemp 9 1959-12-31 realgdp 2785.204
-1.265934
```

Al omitir el último argumento, obtiene un DataFrame con columnas jerárquicas:

```
En [151]: pivoted = ldata.pivot('fecha', 'elemento')

En [152]: pivotado[:5]
Fuera[152]:
value                value2
fecha      infl realgdp unemp      infl realgdp      desempleo
del artículo
1959-03-31 0.00 2710.349 5.8 0.000940 0.523772 1.343810
1959-06-30 2.34 2778.801 5.1 -0.831154 -0.713544 -2.370232
1959-09-30 2.74 2775.488 5.3 -0.860757 -1.860761 0.560145 1959-12-31 0.27
2785.204 5.6 0.119827 -1.265934 -1.063512
1960-03-31 2.31 2847.699 5.2 -2.359419 0.332883 -0.199543

En [153]: pivotado['valor'][:5]
Salida[153]:
fecha      infl realgdp unemp
del artículo
1959-03-31 0.00 2710.349          5.8
1959-06-30 2.34 2778.801          5.1
1959-09-30 2.74 2775.488          5.3
1959-12-31 0.27 2785.204          5.6
1960-03-31 2.31 2847.699          5.2
```

Tenga en cuenta que pivote es equivalente a crear un índice jerárquico usando set_index seguido de una llamada para desapilar:

```
En [154]: sin apilar = ldata.set_index(['fecha', 'elemento']).unstack('elemento')

En [155]: desapilado[:7]
Fuera[155]:
```

	value	value2
fecha	infl realgdp unemp	infl realgdp
del artículo		desempleado
1959-03-31	0.00 2710.349 5.8 0.000940 0.523772	1.343810
30-06-1959	2,34 2778,801 5,1 -0,831154 -0,713544 -2,370232	30-09-1959 2,74 2775,488 5,3 -0,860757 -1,860761 0,560145
1959-12-31	0.27 2785.204 5.6 0.119827 -1.265934	-1.063512
1960-03-31	2.31 2847.699 5.2 -2.359419 0.332883	-0.199543
1960-06-30	0.14 2834.390 5.2 -0.970736 -1.541996	-1.307030
1960-09-30	2.70 2839.022 5.6 0.377984 0.286350	-0.753887

Rotación de formato "ancho" a "largo" Una

operación inversa para pivotar para DataFrames es pandas.melt. En lugar de transformar una columna en muchas en un nuevo DataFrame, fusiona múltiples columnas en una, produciendo un DataFrame que es más largo que la entrada. Veamos un ejemplo:

```
En [157]: df = pd.DataFrame({'clave': ['foo', 'bar', 'baz'], 'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})

En [158]: d.f.
Fuera[158]:
```

	Tecla ABC
foo	1 2 5 8 bar 2 3 6
baz	

La columna 'clave' puede ser un indicador de grupo, y las otras columnas son valores de datos. Al usar pandas.melt, debemos indicar qué columnas (si las hay) son indicadores de grupo. Usemos 'clave' como el único indicador de grupo aquí:

```
En [159]: melted = pd.melt(df, ['clave'])

En [160]: fundido
Out[160]:
```

	valor de la variable clave	0
foo	1	1
bar	2	5
baz	3	8
foo	1	2
bar	2	3
baz	3	6
foo	1	7
bar	2	4
baz	3	9

Usando pivot, podemos remodelar de nuevo al diseño original:

```
En [161]: reformado = melted.pivot('clave', 'variable', 'valor')
```

```
En [162]: reformado
```

```
Out[162]:
barra de teclas ABC
```

```
variable      2 5 8
baz           3 6 9 1 4 7
foo
```

Dado que el resultado de pivot crea un índice de la columna utilizada como etiquetas de fila, es posible que deseemos usar `reset_index` para mover los datos nuevamente a una columna:

```
En [163]: reformado.reset_index()
Out[163]:
tecla variable ABC bar 2 5 8 0
7          baz 3 6 9 foo 1 4
1
2
```

También puede especificar un subconjunto de columnas para usar como columnas de valor:

```
En [164]: pd.melt(df, id_vars=['clave'], value_vars=['A', 'B'])
Out[164]:
valor de la variable clave 0
foo 1 1 bar 2 baz 3 foa
      A      2
      A      3
      B      4
4 bar 5      B      5
baz      B      6
```

`pandas.melt` también se puede usar sin ningún identificador de grupo:

```
En [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Salida[165]:
valor variable A 1 2
0
1      A
2      A      3
3      B      4
4      B      5
5      B      6
6      C      7
7      C      8
8      C      9
```

```
En [166]: pd.melt(df, value_vars=['clave', 'A', 'B'])
Out[166]:
tecla de valor variable
0      barra de
1      teclas foo
```

2	baz clave	
3	un 1	
4	A	2
5	A	
6	B	3 4
7	B	5
8	B	6

8.4 Conclusión

Ahora que tiene algunos conceptos básicos de pandas para la importación, limpieza y reorganización de datos en su haber, estamos listos para pasar a la visualización de datos con matplotlib. Volveremos a los pandas más adelante en el libro cuando hablemos de análisis más avanzados.

Trazado y Visualización

Hacer visualizaciones informativas (a veces llamadas gráficos) es una de las tareas más importantes en el análisis de datos. Puede ser parte del proceso exploratorio, por ejemplo, para ayudar a identificar valores atípicos o transformaciones de datos necesarias, o como una forma de generar ideas para modelos. Para otros, crear una visualización interactiva para la web puede ser el objetivo final. Python tiene muchas bibliotecas complementarias para hacer visualizaciones estáticas o dinámicas, pero me centraré principalmente en `matplotlib` y las bibliotecas que se basan en él. `matplotlib` es un paquete de gráficos de escritorio diseñado para crear gráficos con calidad de publicación (principalmente bidimensionales). El proyecto fue iniciado por John Hunter en 2002 para habilitar una interfaz de trazado similar a MATLAB en Python. Las comunidades `matplotlib` e IPython han colaborado para simplificar el trazado interactivo desde el shell IPython (y ahora, el cuaderno Jupyter). `matplotlib` es compatible con varios backends de GUI en todos los sistemas operativos y, además, puede exportar visualizaciones a todos los formatos de gráficos vectoriales y de trama comunes (PDF, SVG, JPG, PNG, BMP, GIF, etc.). Con la excepción de algunos diagramas, casi todos los gráficos de este libro se produjeron utilizando `matplotlib`.

Con el tiempo, `matplotlib` ha generado una serie de kits de herramientas complementarios para la visualización de datos que utilizan `matplotlib` para su trazado subyacente. Uno de ellos es `el nacido en el mar`, que exploraremos más adelante en este capítulo.

La forma más sencilla de seguir los ejemplos de código del capítulo es utilizar el trazado interactivo en el cuaderno de Jupyter. Para configurar esto, ejecute la siguiente declaración en un cuaderno Jupyter:

```
Cuaderno %matplotlib
```

9.1 Una breve introducción a la API de `matplotlib`

Con `matplotlib`, usamos la siguiente convención de importación:

En [11]: `importar matplotlib.pyplot como plt`

Después de ejecutar `%matplotlib notebook` en Jupyter (o simplemente `%matplotlib` en IPython), podemos intentar crear un diagrama simple. Si todo está configurado correctamente, debería aparecer un diagrama de líneas como el de la Figura 9-1 :

En [12]: `importar numpy como np`

En [13]: `datos = np.arange(10)`

En [14]: `datos`

Salida[14]: `matriz([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`

En [15]: `plt.plot(datos)`

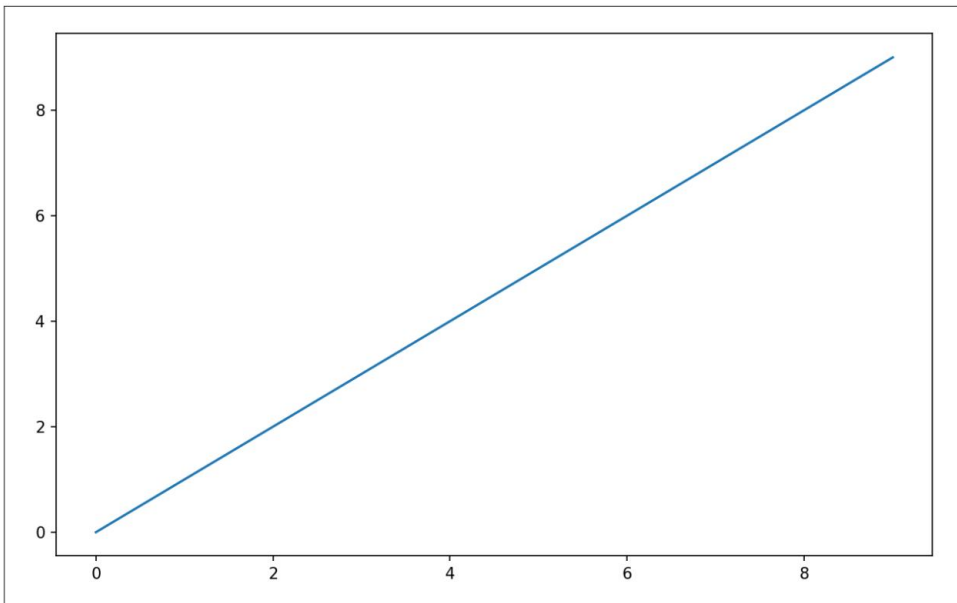


Figura 9-1. Gráfico de línea simple

Si bien las bibliotecas como las funciones de trazado integradas de `seaborn` y `pandas` se ocuparán de muchos de los detalles mundanos de la creación de gráficos, si desea personalizarlos más allá de las opciones de función proporcionadas, deberá aprender un poco sobre la API `matplotlib`.



No hay suficiente espacio en el libro para dar un tratamiento integral a la amplitud y profundidad de la funcionalidad en `matplotlib`. Debería ser suficiente para enseñarle las cuerdas para ponerse en marcha. La galería y la documentación de `matplotlib` son el mejor recurso para aprender funciones avanzadas.

Figuras y subgráficos Los

gráficos en matplotlib residen dentro de un objeto Figura . Puedes crear una nueva figura con `plt.figure()`:

En [16]: `fig = plt.figure()`

En IPython, aparecerá una ventana de trazado vacía, pero en Jupyter no se mostrará nada hasta que usemos algunos comandos más. `plt.figure` tiene varias opciones; en particular, `figsize` garantizará que la figura tenga cierto tamaño y relación de aspecto si se guarda en el disco.

No puedes hacer un gráfico con una figura en blanco. Tienes que crear una o más subparcelas usando `add_subplot`:

En [17]: `ax1 = fig.add_subplot(2, 2, 1)`

Esto significa que la figura debe ser 2×2 (hasta cuatro parcelas en total), y estamos seleccionando la primera de cuatro subparcelas (numeradas desde 1). Si crea las siguientes dos subtramas, terminará con una visualización que se parece a [la Figura 9-2](#):

En [18]: `ax2 = fig.add_subplot(2, 2, 2)`

En [19]: `ax3 = fig.add_subplot(2, 2, 3)`

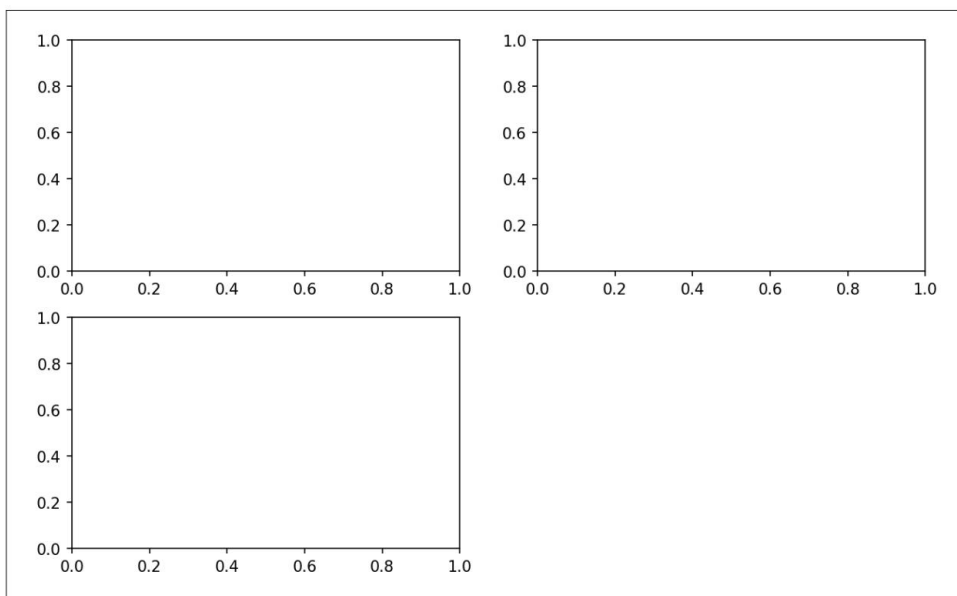


Figura 9-2. Una figura matplotlib vacía con tres subparcelas



Un matiz del uso de cuadernos de Jupyter es que los gráficos se restablecen después de evaluar cada celda, por lo que para gráficos más complejos debe poner todos los comandos de trazado en una sola celda del cuaderno.

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)
```

Cuando ejecuta un comando de trazado como `plt.plot([1.5, 3.5, -2, 1.6])`, `matplotlib` se basa en la última figura y la subtrama utilizada (creando una si es necesario), ocultando así la creación de la figura y la subtrama. Entonces, si agregamos el siguiente comando, obtendrá algo como [la Figura 9-3](#):

En [20]: `plt.plot(np.random.randn(50).cumsum(), 'k--')`

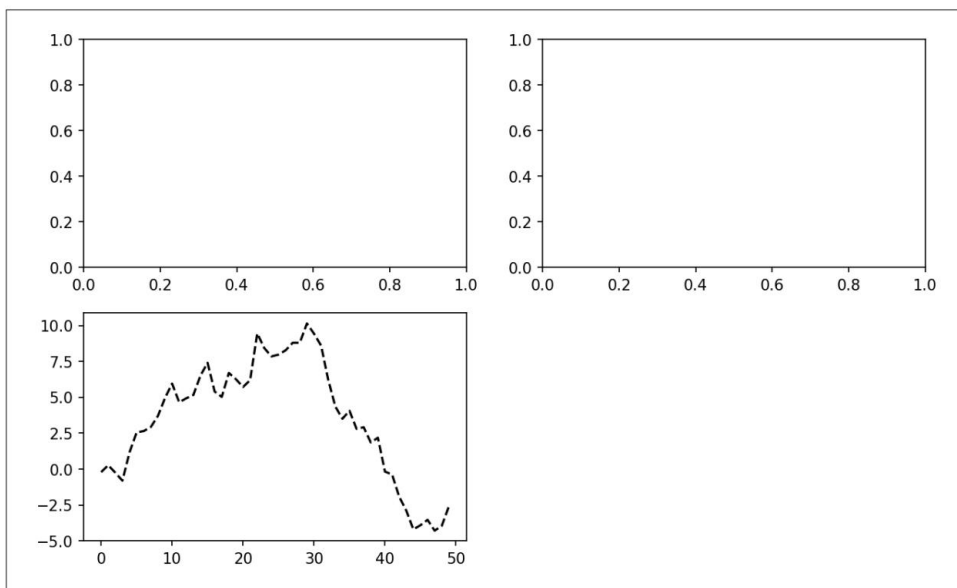


Figura 9-3. Visualización de datos después de una sola parcela

La 'k--' es una opción de estilo que indica a `matplotlib` que trace una línea discontinua negra. Los objetos devueltos por `fig.add_subplot` aquí son objetos `AxesSubplot`, en los que puede trazar directamente en las otras subtramas vacías llamando al método de instancia de cada una (vea la [Figura 9-4](#)):

En [21]: `_ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)`

En [22]: `ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))`

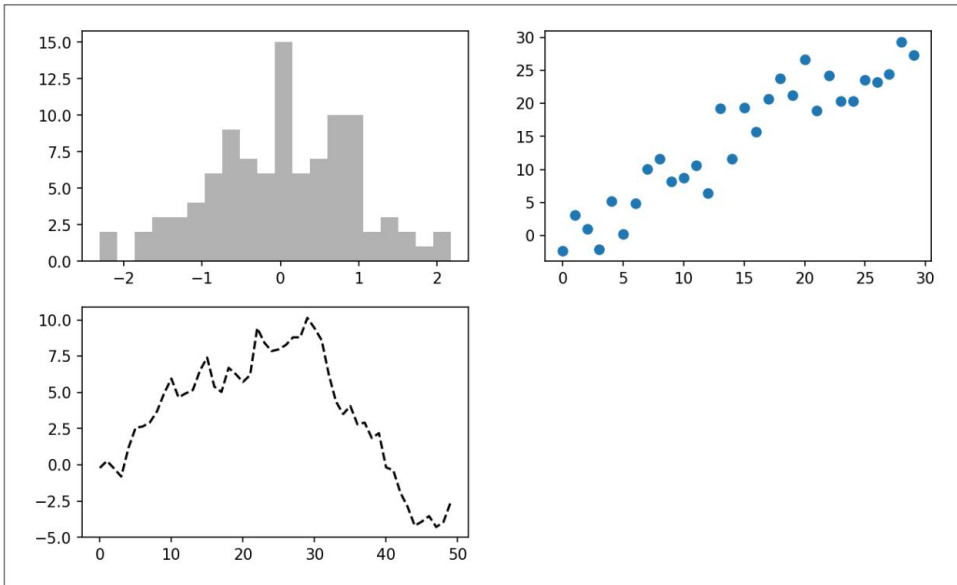


Figura 9-4. Visualización de datos después de parcelas adicionales

Puede encontrar un catálogo completo de tipos de gráficos en la [documentación de matplotlib](#).

Crear una figura con una cuadrícula de subtramas es una tarea muy común, por lo que matplotlib incluye un método conveniente, `plt.subplots`, que crea una nueva figura y devuelve una matriz NumPy que contiene los objetos de subtrama creados:

En [24]: `fig, ejes = plt.subplots(2, 3)`

In [25]: `ejes`

Out[25]:

```
array([[< objeto matplotlib.axes._subplots.AxesSubplot en 0x7fb626374048>, < objeto
matplotlib.axes._subplots.AxesSubplot en 0x7fb62625db00>, < objeto
matplotlib.axes._subplots.AxesSubplot en 0x7fb6262f6c88>], [< objeto
matplotlib.axes._subplots.AxesSubplot en 0x7fb6261a36a0>, < objeto
matplotlib.axes._subplots.AxesSubplot en 0x7fb626181860>, < objeto
matplotlib.axes._subplots.AxesSubplot en 0 x7fb6260fd4e0>]], dtype =objeto )
```

Esto es muy útil, ya que la matriz de ejes se puede indexar fácilmente como una matriz bidimensional; por ejemplo, `ejes[0, 1]`. También puede indicar que las subparcelas deben tener el mismo eje x o y usando `sharex` y `sharey`, respectivamente. Esto es especialmente útil cuando compara datos en la misma escala; de lo contrario, matplotlib escala automáticamente los límites de la trama de forma independiente. Consulte la [Tabla 9-1](#) para obtener más información sobre este método.

Tabla 9-1. opciones de pyplot.subplots

Argumento	Descripción
filas	Número de filas de subparcelas
ncoles	Número de columnas de subparcelas
compartir	Todas las subparcelas deben usar las mismas marcas del eje x (el ajuste de xlim afectará a todas las subparcelas)
compartir	Todas las subparcelas deben usar las mismas marcas del eje y (ajustar el ylim afectará a todas las subparcelas)
subplot_kw	Dict de palabras clave pasadas a la llamada add_subplot utilizada para crear
cada subplot **fig_kw	Se usan palabras clave adicionales para subplots al crear la figura, como plt.subplots(2, 2, tamaño de figura = (8, 6))

Ajustar el espaciado alrededor de las

subparcelas Por defecto, matplotlib deja una cierta cantidad de relleno alrededor del exterior de las subparcelas y el espacio entre las subparcelas. Este espaciado se especifica en relación con la altura y el ancho del gráfico, de modo que si cambia el tamaño del gráfico mediante programación o manualmente mediante la ventana GUI, el gráfico se ajustará dinámicamente. Puede cambiar el espaciado usando el método subplots_adjust en los objetos Figura , también disponible como una función de nivel superior:

```
subplots_adjust(izquierda=Ninguno, abajo=Ninguno, derecha=Ninguno, arriba=Ninguno,
                wspace=Ninguno, hspace=Ninguno)
```

wspace y hspace controlan el porcentaje del ancho de la figura y la altura de la figura, respectivamente, para usar como espaciado entre las subparcelas. Aquí hay un pequeño ejemplo donde reduzco el espacio hasta cero (vea la **Figura 9-5**):

```
fig, ejes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        ejes[i, j].hist(np.random.randn(500),
                        contenedores=50, color='k', alfa=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

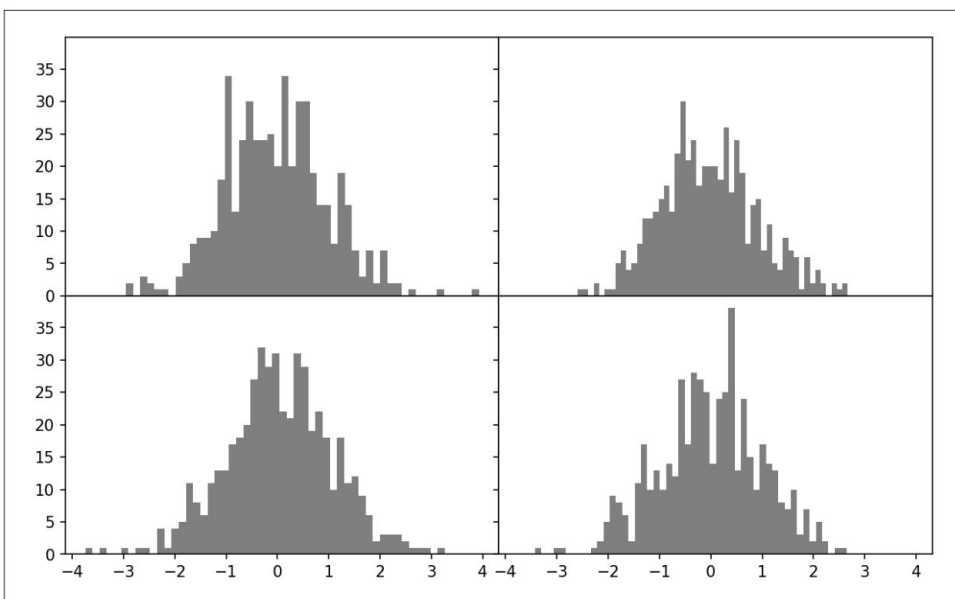


Figura 9-5. Visualización de datos sin espacio entre subparcelas

Puede notar que las etiquetas de los ejes se superponen. matplotlib no verifica si las etiquetas se superponen, por lo que, en un caso como este, debe corregir las etiquetas usted mismo especificando ubicaciones de marca explícitas y etiquetas de marca (veremos cómo hacerlo en las siguientes secciones).

Colores, marcadores y estilos de línea La

función de trazado principal de Matplotlib acepta matrices de coordenadas x e y, opcionalmente, una abreviatura de cadena que indica el color y el estilo de línea. Por ejemplo, para graficar x versus y con guiones verdes, ejecutaría:

```
ax.plot(x, y, 'g--')
```

Esta forma de especificar tanto el color como el estilo de línea en una cadena se proporciona por conveniencia; en la práctica, si estuviera creando tramas programáticamente, es posible que prefiera no tener que juntar cadenas para crear tramas con el estilo deseado. La misma trama también podría haberse expresado más explícitamente como:

```
ax.plot(x, y, estilo de línea='--', color='g')
```

Se proporcionan varias abreviaturas de colores para los colores de uso común, pero puede usar cualquier color del espectro especificando su código hexadecimal (p. ej., '#CECECE').

Puede ver el conjunto completo de estilos de línea mirando la cadena de documentos para plot (¿usar plot? en IPython o Jupyter).

Los gráficos de líneas también pueden tener marcadores para resaltar los puntos de datos reales. Dado que matplotlib crea un gráfico de líneas continuas, interpolando entre puntos, en ocasiones puede no estar claro dónde se encuentran los puntos. El marcador puede ser parte de la cadena de estilo, que debe tener un color seguido del tipo de marcador y el estilo de línea (consulte la [Figura 9-6](#)):

```
En [30]: from numpy.random import randn
```

```
En [31]: plt.plot(randn(30).cumsum(), 'ko--')
```

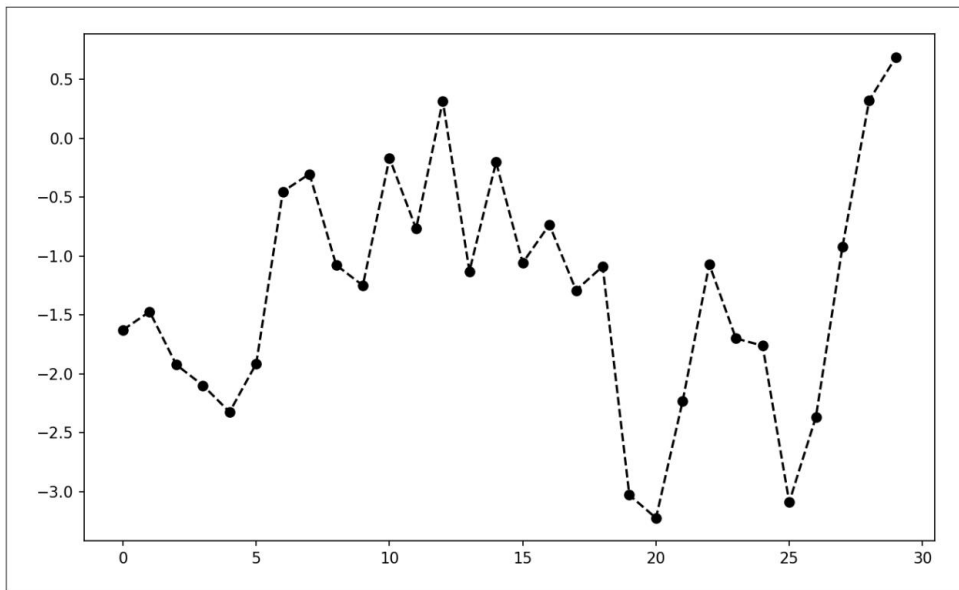


Figura 9-6. Gráfico de líneas con marcadores

Esto también podría haberse escrito más explícitamente como:

```
plot(randn(30).cumsum(), color='k', estilo de línea='discontinua', marcador='o')
```

Para los gráficos de líneas, notará que los puntos subsiguientes se interpolan linealmente de forma predeterminada.

Esto se puede modificar con la opción de estilo de dibujo ([Figura 9-7](#)):

```
En [33]: datos = np.random.randn(30).cumsum()
```

```
En [34]: plt.plot(data, 'k--', label='Default')
```

```
Salida[34]: [<matplotlib.lines.Line2D en 0x7fb624d86160>]
```

```
En [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
```

```
Salida[35]: [<matplotlib.lines.Line2D en 0x7fb624d869e8>]
```

```
En [36]: plt.legend(loc='mejor')
```

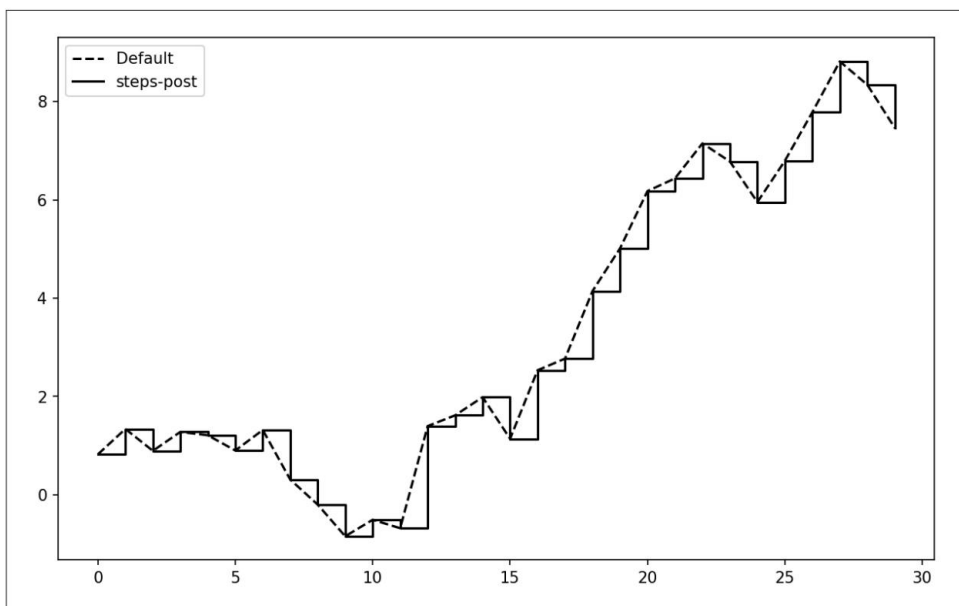


Figura 9-7. Trazado de líneas con diferentes opciones de estilo de dibujo

Puede notar una salida como `<matplotlib.lines.Line2D at ...>` cuando ejecuta esto. `matplotlib` devuelve objetos que hacen referencia al subcomponente de trama que se acaba de agregar.

La mayor parte del tiempo puede ignorar con seguridad esta salida. Aquí, dado que pasamos los argumentos de la etiqueta a `plot`, podemos crear una leyenda de `plot` para identificar cada línea usando `plt.legend`.



Debe llamar a `plt.legend` (o `ax.legend`, si tiene una referencia a los ejes) para crear la leyenda, haya pasado o no las opciones de etiqueta al trazar los datos.

Marcas, etiquetas y leyendas Para la mayoría

de los tipos de decoraciones de gráficos, hay dos formas principales de hacer las cosas: usar la interfaz de `pyplot` procedimental (es decir, `matplotlib.pyplot`) y la API `matplotlib` nativa más orientada a objetos.

La interfaz de `pyplot`, diseñada para uso interactivo, consta de métodos como `xlim`, `xticks` y `xticklabels`. Éstos controlan el rango de trazado, las ubicaciones de las marcas y las etiquetas de las marcas, respectivamente. Se pueden utilizar de dos formas:

- Llamado sin argumentos devuelve el valor del parámetro actual (p. ej., `plt.xlim()` devuelve el rango de trazado del eje x actual)
- Llamado con parámetros establece el valor del parámetro (p. ej., `plt.xlim([0, 10])`, establece el rango del eje x de 0 a 10)

Todos estos métodos actúan en el `AxesSubplot` activo o creado más recientemente. Cada uno de ellos corresponde a dos métodos en el propio objeto de la subtrama; en el caso de `xlim`, estos son `ax.get_xlim` y `ax.set_xlim`. Prefiero usar los métodos de instancia de la trama secundaria para ser explícito (y especialmente cuando trabajo con varias tramas secundarias), pero ciertamente puede usar el que le resulte más conveniente.

Establecer el título, las etiquetas de los ejes, las marcas y las etiquetas de marcas

Para ilustrar la personalización de los ejes, crearé una figura simple y un diagrama de un recorrido aleatorio (consulte [la Figura 9-8](#)):

En [37]: `fig = plt.figure()`

En [38]: `hacha = fig.add_subplot(1, 1, 1)`

En [39]: `ax.plot(np.random.randn(1000).cumsum())`

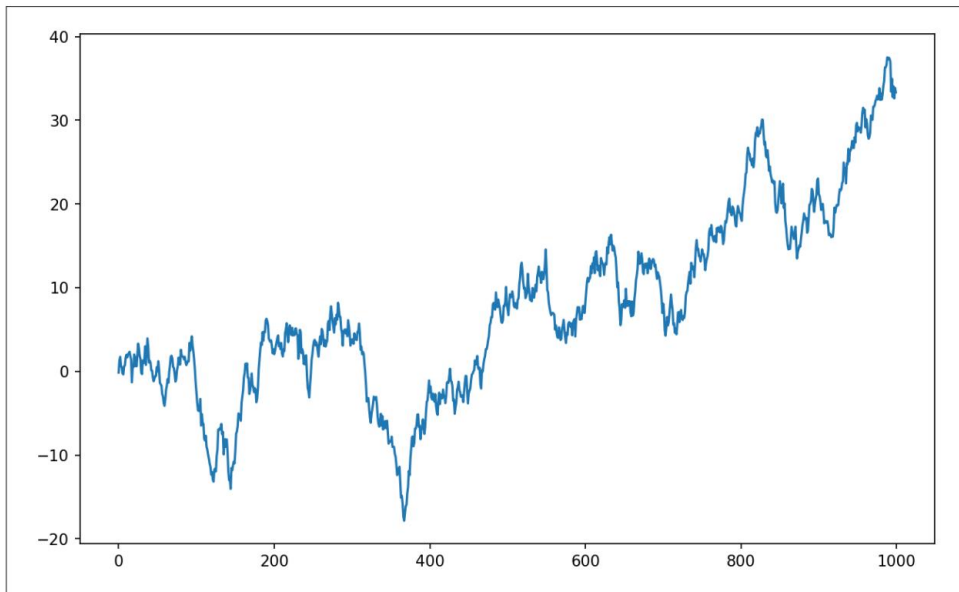


Figura 9-8. Trama simple para ilustrar `xticks` (con etiqueta)

Para cambiar las marcas del eje x, es más fácil usar `set_xticks` y `set_xticklabels`. El primero le indica a matplotlib dónde colocar las marcas a lo largo del rango de datos; por defecto

estas ubicaciones también serán las etiquetas. Pero podemos establecer cualquier otro valor como etiquetas usando `set_xticklabels`:

```
En [40]: marcas = ax.set_xticks([0, 250, 500, 750, 1000])
```

```
En [41]: etiquetas = ax.set_xticklabels(['uno', 'dos', 'tres', 'cuatro', 'cinco'], rotación=30, tamaño
.....:                                  de fuente='pequeño')
```

La opción de rotación establece las etiquetas de marca x en una rotación de 30 grados. Por último, `set_xlabel` le da un nombre al eje x y `set_title` el título de la subparcela (vea la [Figura 9-9](#) para ver la figura resultante):

```
En [42]: ax.set_title('Mi primera trama matplotlib')
```

```
Salida[42]: <matplotlib.text.Text en 0x7fb624d055f8>
```

```
En [43]: ax.set_xlabel('Etapas')
```

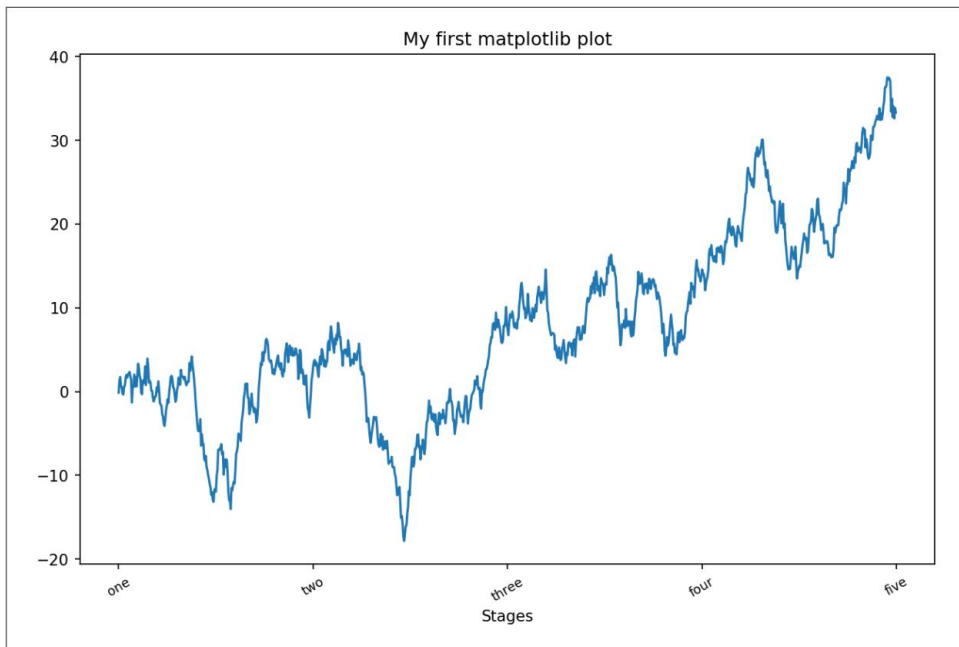


Figura 9-9. Trama simple para ilustrar xticks

Modificar el eje y consiste en el mismo proceso, sustituyendo y por x en lo anterior.

La clase de ejes tiene un método establecido que permite la configuración por lotes de las propiedades del gráfico.

Del ejemplo anterior, también podríamos haber escrito:

```
props =
{ 'title': 'Mi primera trama matplotlib', 'xlabel':
  'Etapas'
} ax.set(**accesorios)
```

Adición de

leyendas Las leyendas son otro elemento crítico para identificar los elementos de la trama. Hay un par de maneras de agregar uno. Lo más fácil es pasar el argumento de la etiqueta al agregar cada parte de la trama:

En [44]: `from numpy.random import randn`

En [45]: `fig = plt.figure(); hacha = fig.add_subplot(1, 1, 1)`

En [46]: `ax.plot(randn(1000).cumsum(), 'k', label='one')`

Salida[46]: [`<matplotlib.lines.Line2D en 0x7fb624bdf860>`]

En [47]: `ax.plot(randn(1000).cumsum(), 'k--', label='two')`

Salida[47]: [`<matplotlib.lines.Line2D en 0x7fb624be90f0>`]

En [48]: `ax.plot(randn(1000).cumsum(), 'k.', label='tres')`

Salida[48]: [`<matplotlib.lines.Line2D en 0x7fb624be9160>`]

Una vez que haya hecho esto, puede llamar a `ax.legend()` o `plt.legend()` para crear automáticamente una leyenda. El gráfico resultante se muestra en la Figura 9-10:

En [49]: `ax.legend(loc='mejor')`

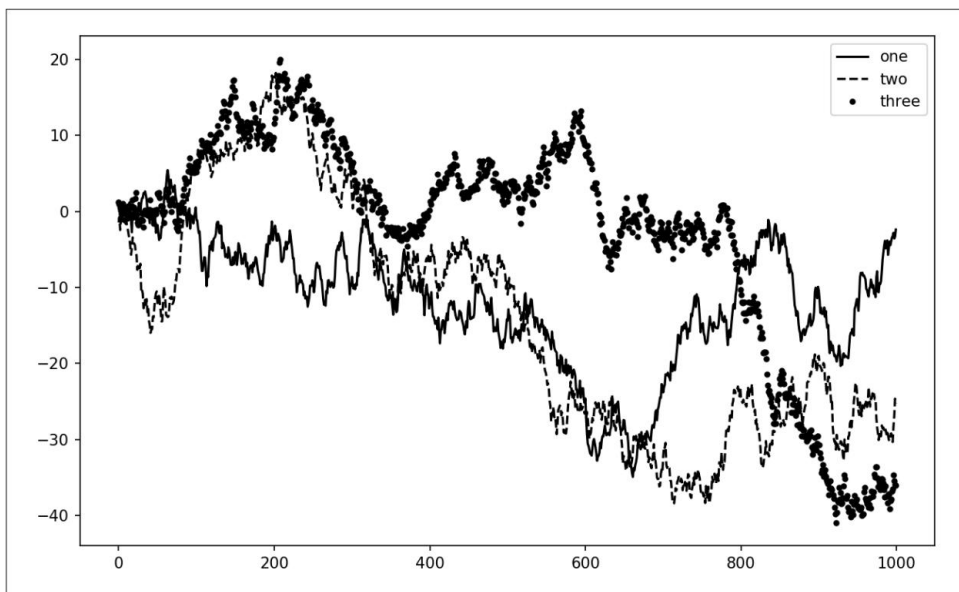


Figura 9-10. Trama simple con tres líneas y leyenda

El método de la leyenda tiene otras opciones para el argumento `loc` ubicación . Consulte la cadena de documentación (¿con `ax.legend?`) para obtener más información.

El loc le dice a matplotlib dónde colocar el gráfico. Si no eres quisquilloso, 'mejor' es una buena opción, ya que elegirá una ubicación que esté más apartada. Para excluir uno o más elementos de la leyenda, pase no label o label='_nolegend_'.

Anotaciones y dibujo en una subparcela Además de

los tipos de parcela estándar, es posible que desee dibujar sus propias anotaciones de parcela, que pueden consistir en texto, flechas u otras formas. Puede agregar anotaciones y texto usando las funciones de texto, flecha y anotar. text dibuja texto en las coordenadas dadas (x, y) en el gráfico con un estilo personalizado opcional:

```
ax.text(x, y, '¡Hola mundo!',
        family='monospace', fontsize=10)
```

Las anotaciones pueden dibujar tanto el texto como las flechas dispuestas adecuadamente. Como ejemplo, tracemos el precio de cierre del índice S&P 500 desde 2007 (obtenido de Yahoo! Finance) y anotemos algunas de las fechas importantes de la crisis financiera de 2008-2009.

Puede reproducir más fácilmente este ejemplo de código en una sola celda en un cuaderno Jupyter. Ve a [la Figura 9-11](#) para ver el resultado:

```
desde fechahora fechahora de importación

fig = plt.figure() hacha
= fig.add_subplot(1, 1, 1)

datos = pd.read_csv('ejemplos/spx.csv', index_col=0, parse_dates=True) spx =
datos['SPX']

spx.plot(ax=ax, estilo='k-')

crisis_data =
[ (datetime(2007, 10, 11), 'Pico del mercado alcista'),
  (datetime(2008, 3, 12), 'Bear Stearns falla'),
  (datetime(2008, 9, 15), 'Lehman Bancarrota')
]

para fecha, etiqueta en crisis_data:
    ax.annotate(label, xy=(fecha, spx.asof(fecha) + 75),
                xytext=(fecha, spx.asof(fecha) + 225),
                arrowprops=dict(facecolor='negro', ancho de cabeza=4, ancho=2,
                                headlength=4),
                horizontalalignment='izquierda', verticalalignment='superior')

# Zoom sobre 2007-2010
hacha.set_xlim(['1/1/2007', '1/1/2011'])
hacha.set_ylim([600, 1800])

ax.set_title(' Fechas importantes en la crisis financiera de 2008-2009')
```

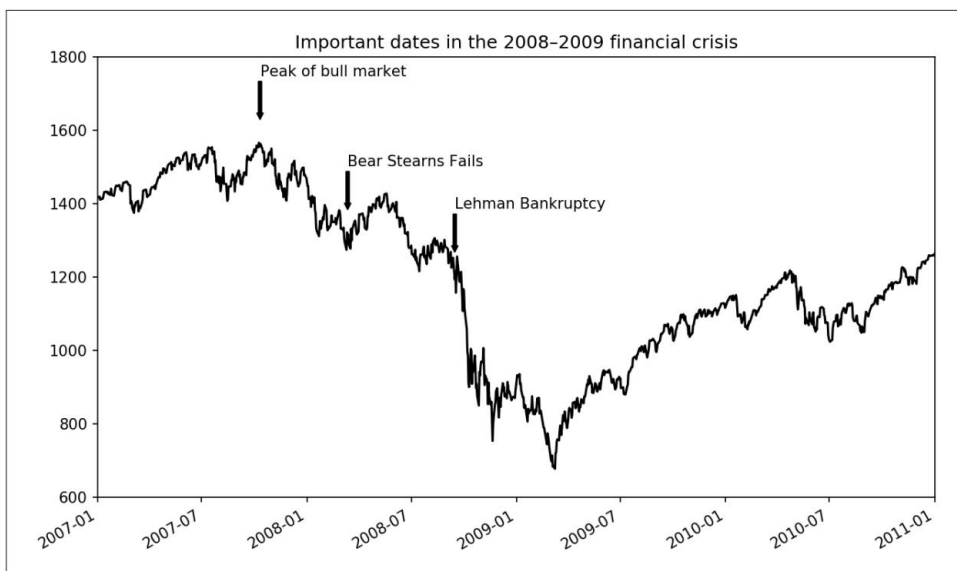


Figura 9-11. Fechas importantes en la crisis financiera de 2008-2009

Hay un par de puntos importantes para resaltar en este gráfico: el método `ax.annotate` puede dibujar etiquetas en las coordenadas `x` e `y` indicadas. Usamos los métodos `set_xlim` y `set_ylim` para establecer manualmente los límites de inicio y final de la trama en lugar de usar el valor predeterminado de `matplotlib`. Por último, `ax.set_title` agrega un título principal a la trama.

Consulte la galería en línea de `matplotlib` para obtener muchos más ejemplos de anotaciones de los que aprender.

Dibujar formas requiere algo más de cuidado. `matplotlib` tiene objetos que representan muchas formas comunes, denominadas parches. Algunos de estos, como `Rectangle` y `Circle`, se encuentran en `matplotlib.pyplot`, pero el conjunto completo se encuentra en `matplotlib.patches`.

Para agregar una forma a un gráfico, cree el objeto de parche `shp` y agréguelo a un subgráfico llamando `ax.add_patch(shp)` (vea la Figura 9-12):

```
fig = plt.figure()
hacha = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]], color='g', alfa=0.5)

hacha.add_patch(rect)
hacha.add_patch(circ)
hacha.add_patch(pgon)
```

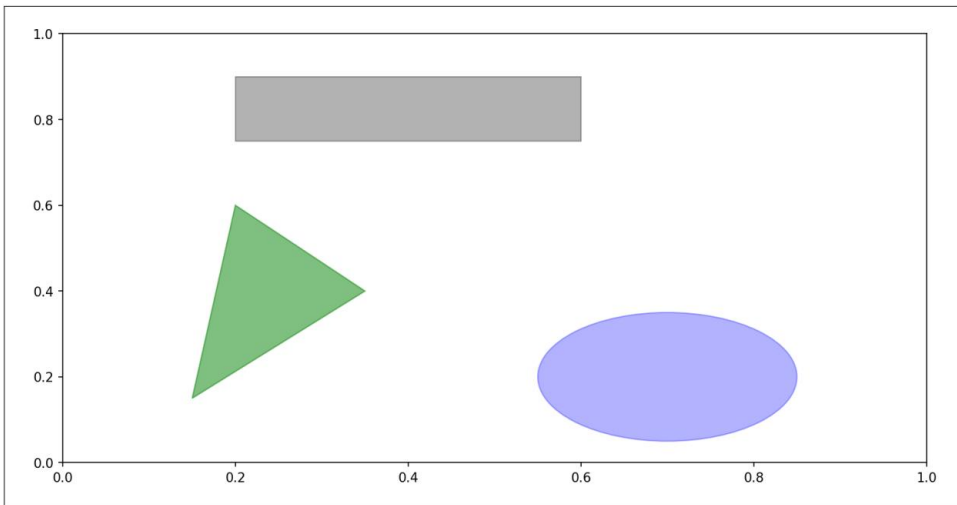


Figura 9-12. Visualización de datos compuesta por tres parches diferentes

Si observa la implementación de muchos tipos de gráficos familiares, verá que se ensamblan a partir de parches.

Guardar gráficos en un archivo

Puede guardar la figura activa en un archivo usando `plt.savefig`. Este método es equivalente al método de instancia `savefig` del objeto `figure`. Por ejemplo, para guardar una versión SVG de una figura, solo necesita escribir:

```
plt.savefig('figpath.svg')
```

El tipo de archivo se deduce de la extensión del archivo. Entonces, si usara `.pdf` en su lugar, obtendría un PDF. Hay un par de opciones importantes que utilizo con frecuencia para publicar gráficos: `dpi`, que controla la resolución de puntos por pulgada, y `bbox_inches`, que puede recortar el espacio en blanco alrededor de la figura real. Para obtener el mismo gráfico que un PNG con un espacio en blanco mínimo alrededor del gráfico y a 400 DPI, haría lo siguiente:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='apretado')
```

`savefig` no tiene que escribir en el disco; también puede escribir en cualquier objeto similar a un archivo, como un `BytesIO`:

```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

Consulte la [Tabla 9-2](#) para obtener una lista de algunas otras opciones para `savefig`.

Tabla 9-2. Opciones de figure.savefig

Argumento	Descripción
fnombre	Cadena que contiene una ruta de archivo o un objeto similar a un archivo de Python. El formato de la figura se deduce de la extensión del archivo (p. ej., .pdf para PDF o .png para PNG)
ppp	La resolución de la figura en puntos por pulgada; el valor predeterminado es 100, pero se puede configurar El
color de cara, color de borde	color del fondo de la figura fuera de las subtramas; 'w' (blanco), por defecto
formato	El formato de archivo explícito a usar ('png', 'pdf', 'svg', 'ps', 'eps', ...)
bbox_pulgadas	La porción de la figura a guardar; si se pasa 'apretado' , intentará recortar el espacio vacío alrededor de la figura

Configuración de matplotlib matplotlib

viene configurado con esquemas de color y valores predeterminados que están orientados principalmente a la preparación de figuras para su publicación. Afortunadamente, casi todo el comportamiento predeterminado se puede personalizar a través de un amplio conjunto de parámetros globales que rigen el tamaño de la figura, el espaciado de las subtramas, los colores, los tamaños de fuente, los estilos de cuadrícula, etc. Una forma de modificar la configuración mediante programación desde Python es usar el método rc ; por ejemplo, para establecer el tamaño de figura predeterminado global en 10 × 10, puede ingresar:

```
plt.rc('figura', figsize=(10, 10))
```

El primer argumento para rc es el componente que desea personalizar, como 'figura', 'ejes', 'xtick', 'ytick', 'grid', 'leyenda' o muchos otros. Después de eso puede seguir una secuencia de argumentos de palabras clave que indican los nuevos parámetros. Una manera fácil de escribir las opciones en su programa es como un dictado:

```
font_options = {'familia': 'monoespacio', 'peso': 'negrita',
                 'tamaño': 'pequeño'}
plt.rc('fuente', **font_options)
```

Para una personalización más amplia y para ver una lista de todas las opciones, matplotlib viene con un archivo de configuración matplotlibrc en el directorio matplotlib/mpl-data. Si personaliza este archivo y lo coloca en su directorio de inicio titulado .matplotlibrc, se cargará cada vez que use matplotlib.

Como veremos en la siguiente sección, el paquete seaborn tiene varios estilos o temas de trama incorporados que usan el sistema de configuración de matplotlib internamente.

9.2 Trazado con pandas y seaborn

matplotlib puede ser una herramienta de nivel bastante bajo. Un gráfico se ensambla a partir de sus componentes básicos: la visualización de datos (es decir, el tipo de gráfico: línea, barra, cuadro, dispersión, contorno, etc.), leyenda, título, etiquetas de marca y otras anotaciones.

En pandas podemos tener múltiples columnas de datos, junto con etiquetas de fila y columna. Pandas tiene métodos integrados que simplifican la creación de visualizaciones a partir de objetos de marco de datos y serie. Otra biblioteca es **seaborn**, una biblioteca de gráficos estadísticos creada por Michael Waskom. Seaborn simplifica la creación de muchos tipos de visualización comunes.



La importación de seaborn modifica los esquemas de color predeterminados de matplotlib y los estilos de trazado para mejorar la legibilidad y la estética. Incluso si no utiliza la API de seaborn, es posible que prefiera importar seaborn como una forma sencilla de mejorar la estética visual de los diagramas generales de matplotlib.

Gráficos de línea

Series y DataFrame tienen cada uno un atributo de gráfico para crear algunos tipos de gráficos básicos. Por defecto, `plot()` hace diagramas de líneas (vea la [Figura 9-13](#)):

En [60]: `s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))`

En [61]: `s.plot()`

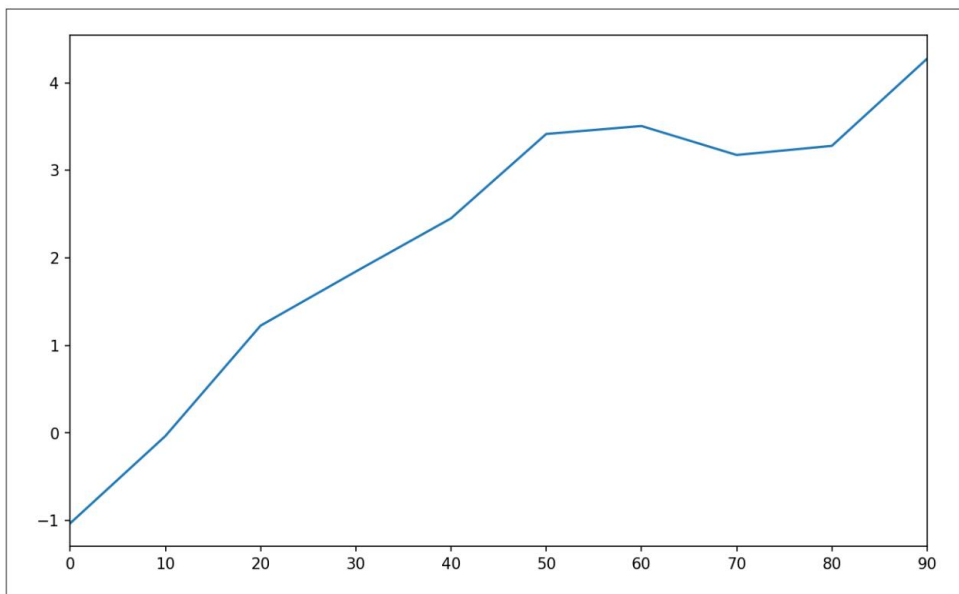


Figura 9-13. Gráfico de serie simple

El índice del objeto Serie se pasa a matplotlib para trazar en el eje x, aunque puede deshabilitarlo pasando `use_index=False`. Los ticks y límites del eje x se pueden ajustar con las opciones `xticks` y `xlim`, y el eje y respectivamente con `yticks` y `ylim`.

ylim. Consulte la [Tabla 9-3](#) para obtener una lista completa de las opciones de trazado. Comentaré algunos más a lo largo de esta sección y dejaré que usted explore el resto.

La mayoría de los métodos de trazado de pandas aceptan un parámetro `ax` opcional, que puede ser un objeto de subtrama `matplotlib`. Esto le brinda una ubicación más flexible de las subparcelas en un diseño de cuadrícula.

El método de trazado de `DataFrame` traza cada una de sus columnas como una línea diferente en el mismo subgráfico, creando una leyenda automáticamente (vea la [Figura 9-14](#)):

```
En [62]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
.....:                   columns=['A', 'B', 'C', 'D'],
.....:                   índice=np.arange(0, 100, 10))
```

```
En [63]: df.plot()
```

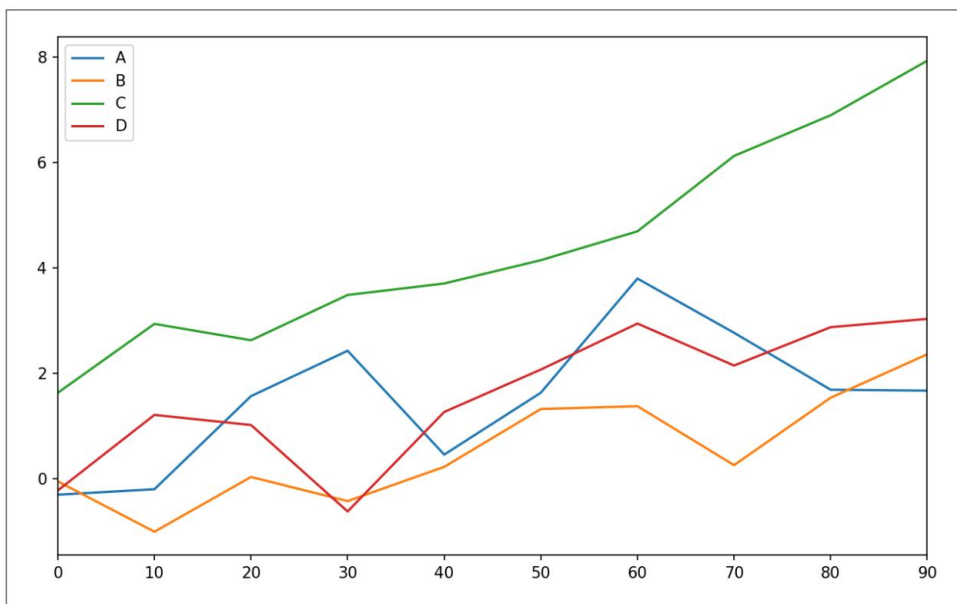


Figura 9-14. Gráfico de marco de datos simple

El atributo `plot` contiene una "familia" de métodos para diferentes tipos de parcelas. Por ejemplo, `df.plot()` es equivalente a `df.plot.line()`. Exploraremos algunos de estos métodos. próximo.



Los argumentos de palabras clave adicionales para trazar se pasan a la respectiva función de trazado de `matplotlib`, por lo que puede personalizar aún más estos gráficos aprendiendo más sobre la API de `matplotlib`.

Tabla 9-3. Argumentos del método Series.plot

Argumento	Descripción
etiqueta	Etiqueta para la leyenda
hacha	de la trama matplotlib subplot object para trazar; si no pasa nada, usa la subtrama matplotlib activa
estilo	Cadena de estilo, como 'ko--', para pasar a matplotlib
alfa	La trama llena la opacidad (de 0 a 1)
amable	Puede ser 'área', 'barra', 'barh', 'densidad', 'hist', 'kde', 'línea', 'pastel' Utilice la escala logarítmica en el eje y logy
use_index	Utilice el índice de objetos para las etiquetas de marca
rotación	Rotación de etiquetas de ticks (0 a 360)
xticks	Valores a usar para las marcas del eje x
garrapatas	Valores a usar para las marcas del eje y
xlim	Límites del eje x (p. ej., [0, 10]) Límites del eje y Cuadrícula del
cuadrícula ylim	eje de visualización (activada de forma predeterminada)

DataFrame tiene una serie de opciones que permiten cierta flexibilidad en la forma en que se manejan las columnas; por ejemplo, si trazarlos todos en la misma subparcela o crear subparcelas separadas. Consulte la [Tabla 9-4](#) para obtener más información sobre estos.

Tabla 9-4. Argumentos de trazado específicos de DataFrame

Argumentos	Descripción
secundarios	Trazar cada columna de DataFrame en una subtrama separada
compartir	Si subplots=True, comparte el mismo eje x, vinculando marcas y límites
compartir	Si subplots=True, comparte el mismo eje y
tamaño de higo	Tamaño de la figura a crear como tupla
título	Trazar título como cadena
legend	Agregar una leyenda de subtrama (Verdadero por defecto)
sort_columns	Trazar columnas en orden alfabético; por defecto usa el orden de las columnas existente



Para el trazado de series de tiempo, consulte el [Capítulo 11](#).

Parcelas de barras

`plot.bar()` y `plot.barh()` hacen gráficos de barras verticales y horizontales, respectivamente. En este caso, el índice `Series` o `DataFrame` se usará como las marcas `x` (`bar`) o `y` (`barh`) (vea la Figura 9-15):

En [64]: `fig, ejes = plt.subplots(2, 1)`

En [65]: `datos = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))`

En [66]: `data.plot.bar(ax=axes[0], color='k', alpha=0.7)`

Salida[66]: `<matplotlib.axes._subplots.AxesSubplot en 0x7fb62493d470>`

En [67]: `data.plot.barh(ax=axes[1], color='k', alpha=0.7)`

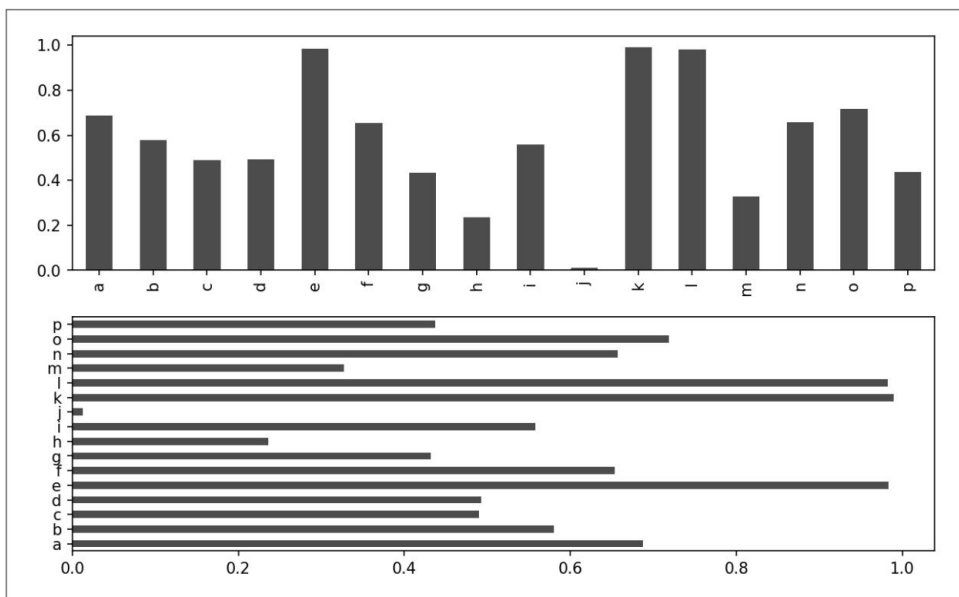


Figura 9-15. Gráfica de barras horizontales y verticales

Las opciones `color='k'` y `alpha=0.7` establecen el color de los gráficos en negro y usan transparencia parcial en el relleno.

Con un DataFrame, los diagramas de barras agrupan los valores de cada fila en un grupo de barras, una al lado de la otra, para cada valor.

Consulte la Figura 9-16:

```
En [69]: df = pd.DataFrame(np.random.rand(6, 4),
.....:                    index=['uno', 'dos', 'tres', 'cuatro', 'cinco', 'seis'], column=pd.Index(['A', 'B', 'C',
.....:                    'D'], nombre='Género'))

En [70]: d.f.
Fuera[70]:
Género      A      B      C      D
uno      0.370670 0.602792 0.229159 0.486744 0.420082
          0.571653 0.049024 0.880592 dos tres 0.814568 0.277160
0.880316 0.431326 cuatro 0.374020 0.899420 0.460304 0.100843
cinco 0.433270 0.125107 0.494675 0.961825

seis      0.601648 0.478576 0.205690 0.560547

En [71]: df.plot.bar()
```

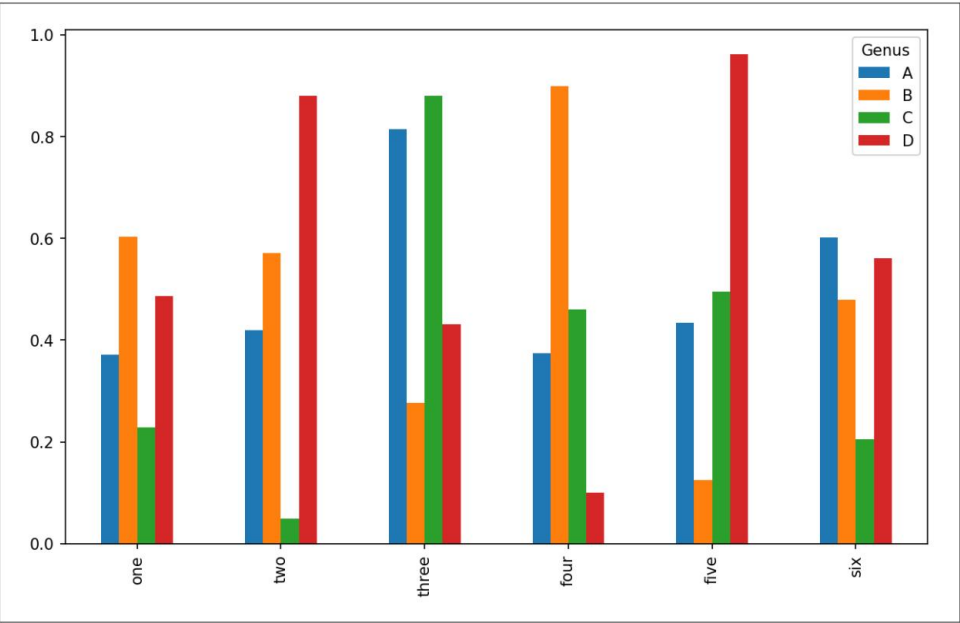


Figura 9-16. Diagrama de barras de marco de datos

Tenga en cuenta que el nombre "Género" en las columnas de DataFrame se usa para titular la leyenda.

Creemos diagramas de barras apiladas a partir de un DataFrame pasando `stacked=True`, lo que da como resultado que el valor de cada fila se apila (consulte la Figura 9-17):

```
En [73]: df.plot.barh(apilado=Verdadero, alfa=0.5)
```

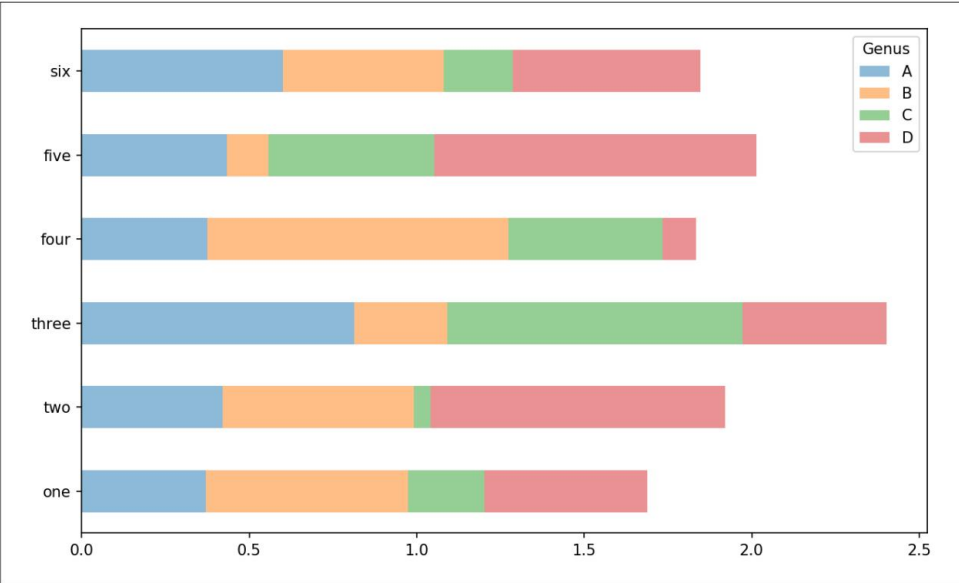


Figura 9-17. Diagrama de barras apiladas de DataFrame



Una receta útil para los diagramas de barras es visualizar la frecuencia del valor de una Serie usando `value_counts()`: `s.value_counts().plot.bar()`.

Volviendo al conjunto de datos de propinas utilizado anteriormente en el libro, supongamos que queremos hacer un gráfico de barras apiladas que muestre el porcentaje de puntos de datos para cada tamaño de grupo en cada día. Cargo los datos usando `read_csv` y hago una tabulación cruzada por día y tamaño del grupo:

```
En [75]: consejos = pd.read_csv('ejemplos/consejos.csv')

En [76]: party_counts = pd.crosstab(consejos['día'], consejos['tamaño'])

En [77]: party_counts
Fuera[77]:
tamaño 1 2 3 4 5 6 día
Vie 1 16 1          1 0 0
sáb 2 53 18 13 1 0
dom 0 39 15 18 3 1
Jue 1 48 4 5 1 3
```

```
# No hay muchas fiestas de 1 y 6 personas En
```

```
[78]: party_counts = party_counts.loc[:, 2:5]
```

Luego, normalice para que cada fila sume 1 y haga la gráfica (vea la Figura 9-18):

```
# Normalizar para sumar 1 En
```

```
[79]: party_pcts = party_counts.div(party_counts.sum(1), axis=0)
```

```
In [80]: party_pcts Out[80]:
```

```
tamaño día  
Vie          2          3          4          5
```

```
0.888889 0.055556 0.055556 0.000000 Sáb 0.623529 0.211765
```

```
0.152941 0.011765
```

```
dom 0.520000 0.200000 0.240000 0.040000 jue 0.827586
```

```
0.068966 0.086207 0.017241
```

```
En [81]: party_pcts.plot.bar()
```

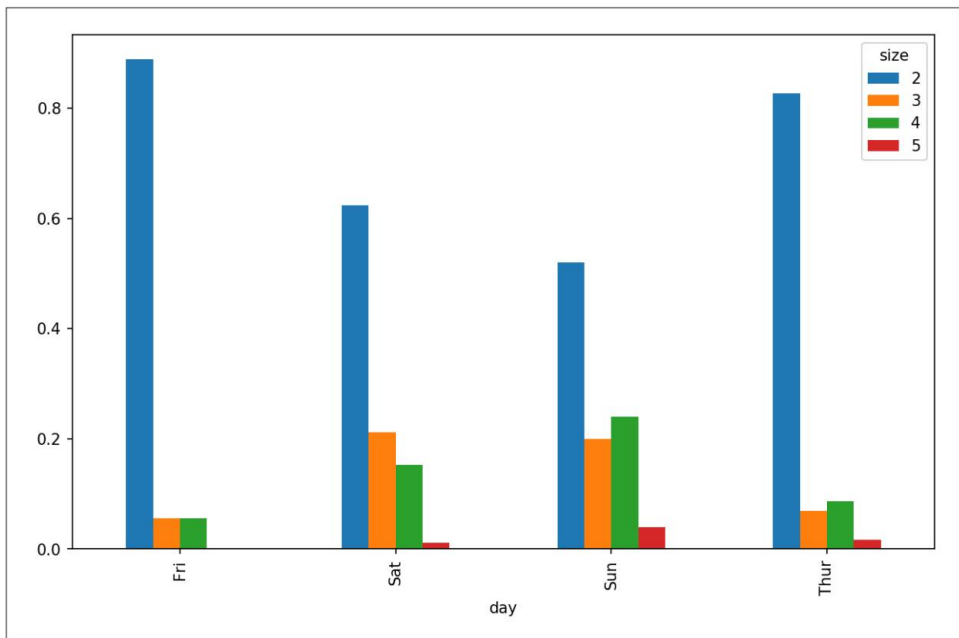


Figura 9-18. Fracción de fiestas por tamaño en cada día

Entonces puede ver que el tamaño de las fiestas parece aumentar el fin de semana en este conjunto de datos.

Con datos que requieren agregación o resumen antes de hacer un gráfico, usar el paquete seaborn puede simplificar mucho las cosas. Veamos ahora el porcentaje de propinas por día con seaborn (vea la Figura 9-19 para el gráfico resultante):

```
En [83]: importar seaborn como sns

En [84]: propinas["tip_pct"] = propinas["tip"] / (propinas["total_bill"] - propinas["tip"])

En [85]: tips.head()
Out[85]:
```

	total_bill	propina	fumador	día	tiempo	tamaño	tip_pct	No Sun
0	16,99	1,01		Dinner	2	0.063204		
1	10,34	1,66	21,01	Cena sin sol			3 0.191244	3
2	3,50			Cena sin sol			0.199886	
3	23,68	3,31		Cena sin sol			2 0.162494	
4	24,59	3,61		Cena sin sol			4 0.172069	

```
En [86]: sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

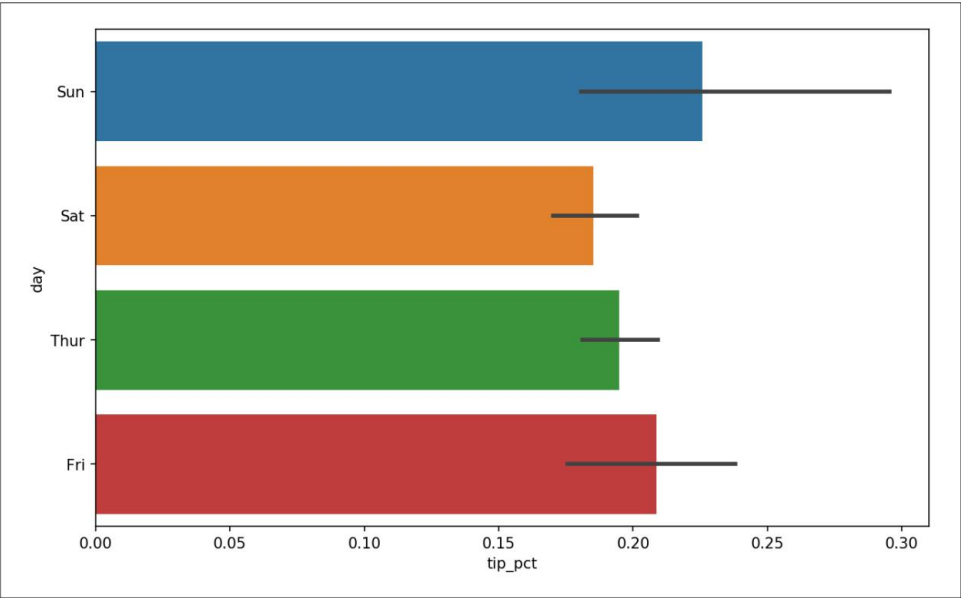


Figura 9-19. Porcentaje de propinas por día con barras de error

Las funciones gráficas en seaborn toman un argumento de datos , que puede ser un marco de datos de pandas. Los otros argumentos se refieren a nombres de columnas. Debido a que hay múltiples observaciones para cada valor en el día, las barras son el valor promedio de tip_pct. Las líneas negras dibujadas en las barras representan el intervalo de confianza del 95% (esto se puede configurar a través de argumentos opcionales).

seaborn.barplot tiene una opción de tono que nos permite dividir por un valor categórico adicional (Figura 9-20):

En [88]: `sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')`

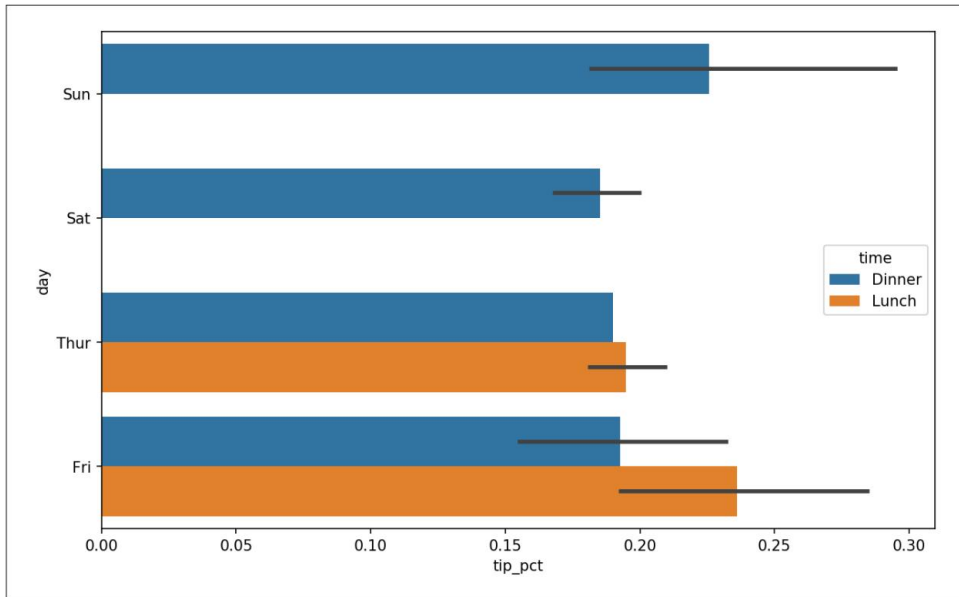


Figura 9-20. Porcentaje de propinas por día y hora

Tenga en cuenta que seaborn ha cambiado automáticamente la estética de los gráficos: la paleta de colores predeterminada, el fondo del gráfico y los colores de las líneas de cuadrícula. Puedes cambiar entre diferentes aspectos de la trama usando `seaborn.set`:

En [90]: `sns.set(style="whitegrid")`

Histogramas y gráficas de densidad

Un histograma es una especie de gráfica de barras que ofrece una visualización discreta de la frecuencia del valor. Los puntos de datos se dividen en contenedores discretos espaciados uniformemente, y se representa gráficamente el número de puntos de datos en cada contenedor. Usando los datos de propinas de antes, podemos hacer un histograma de los porcentajes de propinas de la cuenta total usando el método `plot.hist` en la Serie (vea la Figura 9-21):

En [92]: `consejos['tip_pct'].plot.hist(bins=50)`

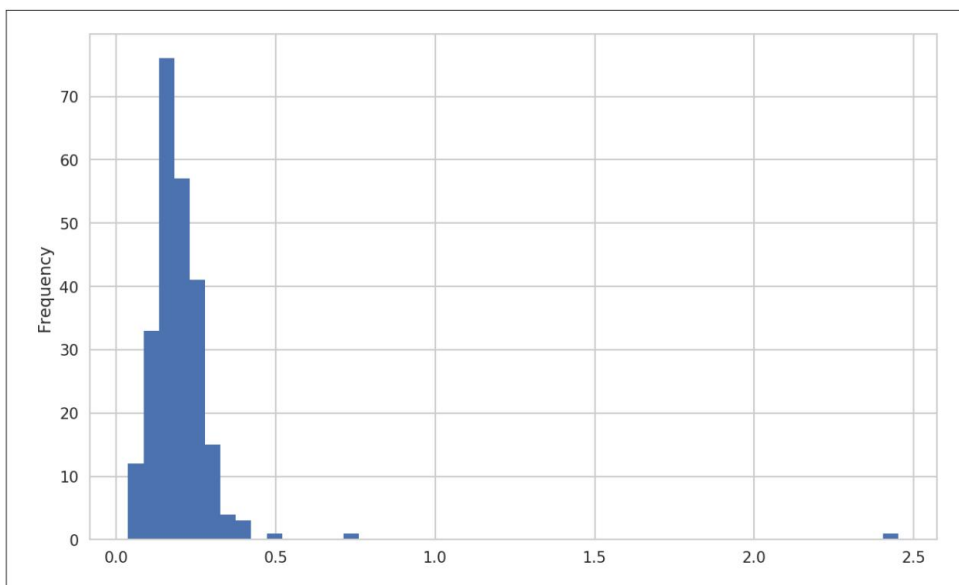


Figura 9-21. Histograma de porcentajes de propinas

Un tipo de gráfico relacionado es un gráfico de densidad, que se forma calculando una estimación de una distribución de probabilidad continua que podría haber generado los datos observados. El procedimiento habitual es aproximar esta distribución como una mezcla de "núcleos", es decir, distribuciones más simples como la distribución normal. Por lo tanto, las gráficas de densidad también se conocen como gráficas de estimación de densidad kernel (KDE). El uso de `plot.kde` crea un gráfico de densidad utilizando la estimación convencional de mezcla de normales (consulte la Figura 9-22):

```
En [94]: consejos["tip_pct"].plot.density()
```

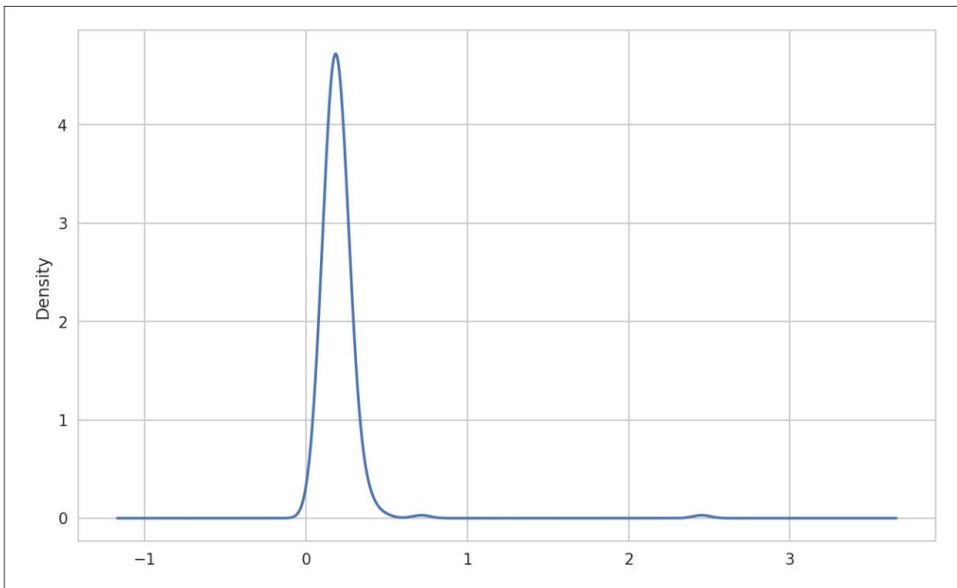


Figura 9-22. Gráfica de densidad de porcentajes de punta

Seaborn hace que los histogramas y los gráficos de densidad sean aún más fáciles a través de su método `distplot`, que puede trazar un histograma y una estimación de densidad continua simultáneamente. Como ejemplo, considere una distribución bimodal que consiste en extracciones de dos distribuciones normales estándar diferentes (vea la figura 9-23):

En [96]: `comp1 = np.random.normal(0, 1, tamaño=200)`

En [97]: `comp2 = np.random.normal(10, 2, tamaño=200)`

En [98]: `valores = pd.Series(np.concatenate([comp1, comp2]))`

En [99]: `sns.distplot(valores, bins=100, color='k')`

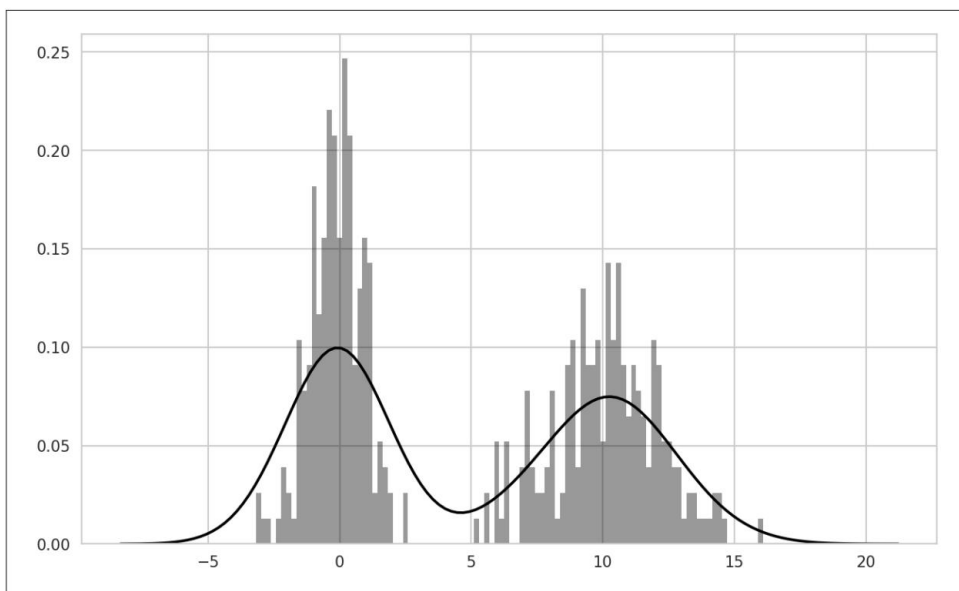


Figura 9-23. Histograma normalizado de mezcla normal con estimación de densidad

Gráficos de dispersión o de puntos

Los diagramas de puntos o de dispersión pueden ser una forma útil de examinar la relación entre dos series de datos unidimensionales. Por ejemplo, aquí cargamos el conjunto de datos de macrodatos del proyecto statsmodels, seleccionamos algunas variables y luego calculamos las diferencias de registro:

```
En [100]: macro = pd.read_csv('ejemplos/macrodata.csv')

En [101]: datos = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

En [102]: trans_data = np.log(data).diff().dropna()

En [103]: trans_datos[-5:]
Fuera[103]:
```

	ipc	desempleado	tasa de billos	m1
-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

Entonces podemos usar el método `regplot` de `seaborn`, que hace un diagrama de dispersión y ajusta una línea de regresión lineal (vea la Figura 9-24):

```
En [105]: sns.regplot('m1', 'unemp', data=trans_data)
```

```
Salida[105]: <matplotlib.axes._subplots.AxesSubplot en 0x7fb613720be0>
```

```
En [106]: plt.title('Cambios en el registro %s frente al registro %s' % ('m1', 'unemp'))
```

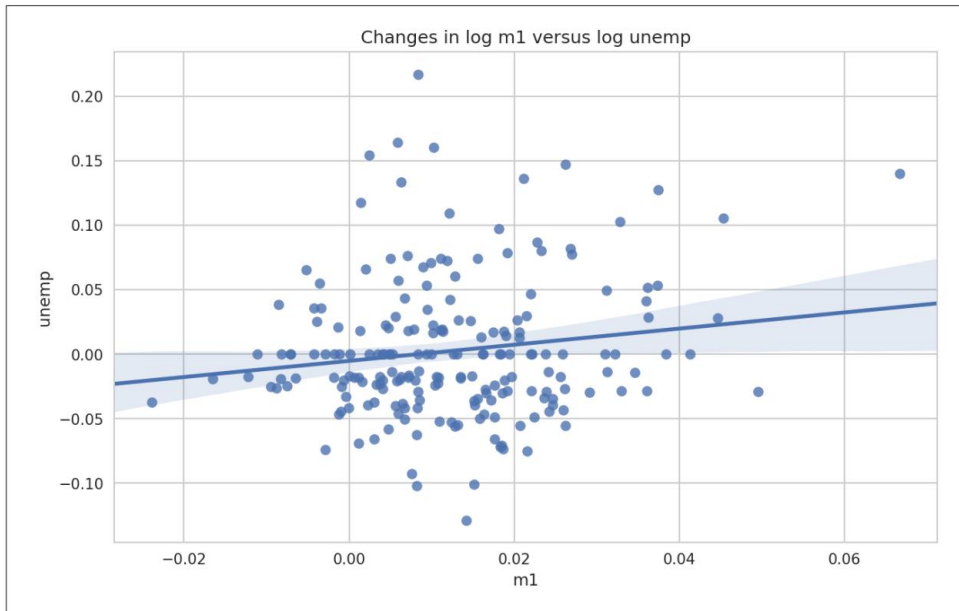


Figura 9-24. Un gráfico de dispersión/regresión marino

En el análisis exploratorio de datos, es útil poder ver todos los diagramas de dispersión entre un grupo de variables; esto se conoce como diagrama de pares o matriz de diagrama de dispersión. Hacer una gráfica de este tipo desde cero es un poco laborioso, por lo que `seaborn` tiene una función de gráfica de pares conveniente, que permite colocar histogramas o estimaciones de densidad de cada variable a lo largo de la diagonal (consulte la Figura 9-25 para ver la gráfica resultante):

```
En [107]: sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})
```