

MUHAMMAD ASIF



PYTHON FOR GEEKS

Build production-ready applications using advanced
Python concepts and industry best practices

Packt

Python para frikis

Cree aplicaciones listas para la producción con avanzados
Conceptos de Python y mejores prácticas de la industria

Muhammad Asif



BIRMINGHAM—MUMBAI

"Python" y el logotipo de Python son marcas comerciales de Python Software Foundation.

Python para frikis

Copyright © 2021 Packt Publishing

Reservados todos los derechos. Ninguna parte de este libro puede reproducirse, almacenarse en un sistema de recuperación o transmitirse de ninguna forma ni por ningún medio sin el permiso previo por escrito del editor, excepto en el caso de citas breves incrustadas en artículos críticos o reseñas.

Se ha hecho todo lo posible en la preparación de este libro para garantizar la exactitud de la información presentada. Sin embargo, la información contenida en este libro se vende sin garantía, ya sea expresa o implícita. Ni el autor, ni Packt Publishing ni sus comerciantes y distribuidores serán responsables de los daños causados o presuntamente causados directa o indirectamente por este libro.

Packt Publishing se ha esforzado por proporcionar información de marcas registradas sobre todas las empresas y productos mencionados en este libro mediante el uso apropiado de capitales. Sin embargo, Packt Publishing no puede garantizar la exactitud de esta información.

Gerente de Producto del Grupo: Richa Tripathi

Gerente de productos editoriales: Sathyaranarayanan Ellapulli

Editor sénior: Rohit Singh

Editor de desarrollo de contenido: Vaishali Ramkumar

Editor técnico: Karan Solanki

Editor de textos: Safis Editing

Coordinador del proyecto: Deeksha Thakkar

Corrector: Safis Editing

Indexador: Pratik Shirodkar

Diseño de producción: Prashant Ghare

Primera publicación: agosto de 2021

Referencia de producción: 1090921

Publicado por Packt Publishing Ltd.

Lugar de librea

Calle librea 35

Birmingham

B3 2PB, Reino Unido.

ISBN 978-1-80107-011-9

www.packt.com

A mi esposa, Saima Arooj, sin cuyo amoroso apoyo no hubiera sido posible completar este libro. A mis hijas, Sana Asif y Sara Asif, y a mi hijo, Zain Asif, quienes fueron mi inspiración a lo largo de este viaje. A la memoria de mi padre y mi madre por sus sacrificios y por ejemplificar el poder de la determinación para mí. Y a mis hermanos, por todo el apoyo y aliento que me han brindado en términos de mi educación y carrera.

– Muhammad Asif

Colaboradores

Sobre el Autor

Muhammad Asif es un arquitecto principal de soluciones con una amplia gama de experiencia multidisciplinaria en desarrollo web, automatización de redes y nubes, virtualización y aprendizaje automático. Con una sólida experiencia en múltiples dominios, ha dirigido muchos proyectos a gran escala hasta su implementación exitosa. Aunque ha pasado a más roles de liderazgo en los últimos años, Muhammad siempre ha disfrutado resolviendo problemas del mundo real mediante el uso de la tecnología adecuada y escribiendo el código él mismo. Obtuvo un Ph.D. en sistemas informáticos de la Universidad de Carleton, Ottawa, Canadá en 2012 y actualmente trabaja para Nokia como líder de soluciones.

Deseo agradecer a aquellas personas que han estado cerca de mí y me han apoyado, especialmente Imran Ahmad, quien contribuyó como coautor del capítulo 1.

Sobre los revisores

Harshit Jain es un científico de datos con cinco años de experiencia en programación que ayuda a las empresas a crear y aplicar algoritmos avanzados de aprendizaje automático para mejorar la eficiencia empresarial. Tiene una amplia comprensión de la ciencia de datos, que va desde el aprendizaje automático clásico hasta el aprendizaje profundo y la visión artificial. Es muy hábil en la aplicación de técnicas de ciencia de datos a empresas de comercio electrónico. Durante su tiempo libre, es mentor de aspirantes a científicos de datos para mejorar sus habilidades. Sus charlas y artículos, que incluyen Learning the Basics of Data Science y How to Check the Impact on Marketing Activities – Market Mix Modeling, muestran que la ciencia de datos no es solo su profesión sino una pasión. Revisar este libro es su primer intento, pero ciertamente no el último, de consolidar su valioso conocimiento de Python para ayudarlo en su aprendizaje.

Sourabh Bhavsar es un desarrollador senior completo, ágil y practicante de la nube con más de 7 años de experiencia en la industria del software. Completó un curso de posgrado en inteligencia artificial y aprendizaje automático de la Universidad de Texas en Austin, y también tiene un MBA (en marketing) y una licenciatura en ingeniería (TI) de la Universidad de Pune, India. Actualmente trabaja en PayPal como miembro técnico principal, donde es responsable de diseñar y desarrollar soluciones basadas en microservicios e implementar varios tipos de motores de orquestación y flujo de trabajo. Sourabh cree en el aprendizaje continuo y disfruta explorando nuevas tecnologías web. Cuando no está programando, le gusta jugar Tabla y leer sobre astrología.

Tabla de contenido

Prefacio

Sección 1: Python, más allá de lo básico

1

Ciclo de vida óptimo de desarrollo de Python

Cultura y comunidad Python 4		Paquetes	21
Diferentes fases de un proyecto		Módulos	21
de Python	6	Convenciones de importación	22
Estrategia del proceso de desarrollo		Argumentos	22
	9	Herramientas útiles	22
Iterando a través de las fases	9	Exploración de opciones para el control	
Apuntando a MVP primero	10	de código fuente	23
Desarrollo de estrategias para		¿Qué no pertenece al repositorio de control	
dominios especializados	10	de código fuente?	23
Documentación efectiva del código		Comprender las estrategias para	
Python	14	implementar el código	
Comentarios de Python	14	24	
cadena de documentación	15	Desarrollo por lotes	24
Documentación funcional o de nivel de clase	16		
Desarrollo de un esquema de		Entornos de desarrollo	
nomenclatura eficaz	18	Python	26
Métodos	19	INACTIVO	26
Variables	19	Texto sublime	26
Constante	20	PyCharm	27
Clases	21	código de estudio visual	27
		PyDev	27

viii Tabla de contenido

espía	28	Preguntas	29
Resumen	28	Otras lecturas	29
		respuestas	29

2**Uso de la modularización para manejar proyectos complejos**

Requisitos técnicos	32	Archivo de inicialización del paquete	53
Introducción a módulos y paquetes		Construyendo un paquete	54
Importación de módulos	32		
Uso de la declaración de importación	33	Acceder a paquetes desde cualquier ubicación	58
Uso de la declaración <code>__import__</code>	35		
Uso de la declaración <code>importlib.import_module</code>	40	Compartir un paquete	63
	41	Creación de un paquete según las pautas de PyPA	64
Importación absoluta versus relativa	42	Instalación desde el código fuente local usando pip	67
Carga e inicialización de un módulo	44	Publicación de un paquete para probar PyPI	70
Módulos estándar	46	Instalando el paquete desde PyPI	71
Escribir módulos reutilizables	47		
Paquetes de construcción	53	Resumen	72
Denominación	53	Preguntas	72
		Otras lecturas	72
		respuestas	73

3**Programación Python avanzada orientada a objetos**

Requisitos técnicos	76	Métodos especiales	83
Introducción a clases y objetos	76	Comprender los principios de la programación orientada a objetos	85
Distinguir entre atributos de clase y atributos de instancia	76	Encapsulación de datos	85
Uso de constructores y destructores con clases	80	Abarcando datos y acciones.	85
Distinguir entre métodos de clase y métodos de instancia	81	ocultar información	87
		Protegiendo los datos	90
		Uso de getters y setters tradicionales	90
		Uso de decoradores de propiedades	92

Ampliación de clases con herencia	93	Usar la composición como un enfoque de diseño alternativo	103
herencia simple	94		
Herencia múltiple	96	Introducción a la escritura de patos	
		Piton	106
Polimorfismo	98		
Sobrecarga de métodos	98	Aprendiendo cuándo no usar OOP en Python	107
Anulación de método	99		
Abstracción	101	Resumen	108
		Preguntas	109
		Otras lecturas	109
		respuestas	109

Sección 2:

Conceptos de programación avanzada

4

Bibliotecas de Python para programación avanzada

Requerimientos técnicos	114	Manejo de errores y excepciones	132
Introducción a los contenedores de datos de Python	114	Trabajando con excepciones en Python	133
		Generar excepciones	136
	114	Definición de excepciones personalizadas	137
Instrumentos de cuerda			
Liza	115	Usando el módulo de registro de Python	138
tuplas	116		
Diccionarios	117	Introducción a los componentes básicos de registro	139
Conjuntos	118	Trabajar con el módulo de registro	141
		Qué registrar y qué no registrar	148
Uso de iteradores y generadores para el procesamiento de datos	119		
iteradores	119	Resumen	149
Generadores	124	Preguntas	149
Manejo de archivos en Python	127	Otras lecturas	150
Operaciones de archivo	127	respuestas	150
Usar un administrador de contexto	130		
Operando en varios archivos	131		

x Tabla de contenido

5

Pruebas y Automatización con Python

Requerimientos técnicos	152	Ejecutando TDD	179
Comprensión de varios niveles de prueba	152	Rojo	179
Examen de la unidad	153	Verde	179
Pruebas de integración	154	refactorizar	180
Pruebas del sistema	154		
Test de aceptación	155	Introducción a la IC automatizada	181
Trabajar con marcos de pruebas de Python	155	Resumen	182
Trabajar con el marco unittest	157	Preguntas	182
Trabajando con el marco pytest	169	Otras lecturas	182
		respuestas	183

6

Consejos y trucos avanzados en Python

Requerimientos técnicos	186	Incrustar un diccionario dentro de un diccionario	207
Aprender trucos avanzados para usar funciones	186	Usando la comprensión	211
Presentamos las funciones counter, itertools y zip para tareas iterativas	186		
Uso de filtros, mapeadores y reductores para transformaciones de datos	192	Presentamos trucos avanzados con pandas DataFrame	214
Aprendiendo a construir funciones lambda	196	Aprendizaje de operaciones de DataFrame	215
Incrustar una función dentro de otra función	197	Aprendiendo trucos avanzados para un Objeto de marco de datos	222
Modificar el comportamiento de la función usando decoradores	200		
Comprehension de conceptos avanzados con estructuras de datos	207	Resumen	228
		Preguntas	229
		Otras lecturas	229
		respuestas	229

Sección 3:

Escalando más allá de un solo subprocesso

7

Multiprocesamiento, multiproceso y asíncrono Programación

Requisitos técnicos	234	Estudio de caso: una aplicación multiprocesador para descargar archivos de Google Drive	
Comprender los subprocessos múltiples en Python y sus limitaciones	234		264
¿Qué es un punto ciego de Python?	236		
Aprender los componentes clave de la programación multiproceso en Python	237		
Estudio de caso: una aplicación multiproceso para descargar archivos de Google Drive	247		
Más allá de una sola CPU: implementación de multiprocesamiento	250		
Creando múltiples procesos	251		
Compartir datos entre procesos	254		
Intercambio de objetos entre procesos	259		
Sincronización entre procesos	262		
		Uso de programación asíncrona para sistemas receptivos	266
		Entendiendo el módulo asyncio	267
		Distribuir tareas usando colas	270
		Caso práctico: aplicación asyncio para descargar archivos de Google Drive	272
		Resumen	274
		Preguntas	275
		Otras lecturas	276
		respuestas	276

8

Escalado horizontal de Python mediante clústeres

Requerimientos técnicos	278	Introducción a los RDD	285
Información sobre las opciones de clúster para el procesamiento en paralelo	279	Aprendiendo operaciones RDD	286
Mapa de HadoopReduce	279	Creación de objetos RDD	287
chispa apache	282	Uso de PySpark para el procesamiento de datos en paralelo	288

xii Tabla de contenido

Creación de SparkSession y Programas SparkContext	290	Estudio de caso 1: Calculadora Pi (π) en chispa apache	302
Explorando PySpark para operaciones RDD	292	Estudio de caso 2 – Nube de palabras usando PySpark	307
Aprender sobre PySpark DataFrames	294		
Presentación de PySpark SQL	300	Resumen	310
Casos prácticos de uso de Apache Chispa y PySpark	302	Preguntas	311
		Otras lecturas respuestas	311

9**Programación en Python para la nube**

Requerimientos técnicos	314	Uso de Google Cloud Platform para el procesamiento de datos	332
Aprender sobre las opciones de nube para aplicaciones de Python	314	Aprendiendo los fundamentos de Apache Haz	333
Introducción a los entornos de desarrollo de Python para la nube	314	Introducción a las canalizaciones de Apache Beam	334
Presentamos opciones de tiempo de ejecución en la nube para Piton	317	Creación de canalizaciones para Cloud Dataflow	341
		Resumen	347
Creación de servicios web de Python para la implementación en la nube	319	Preguntas	348
Uso del SDK de Google Cloud	320	Otras lecturas respuestas	348
Usar la consola web de GCP	329		

Sección 4:**Uso de Python para la web, la nube y la red
Casos de uso****10****Uso de Python para desarrollo web y API REST**

Requerimientos técnicos	354	Requisitos de aprendizaje para el desarrollo web	355
--------------------------------	------------	---	------------

Marcos web	355	Interactuar con sistemas de bases de datos.	368
Interfaz de usuario	356	Manejo de errores y excepciones en aplicaciones	
servidor web/servidor de aplicaciones	357	web	373
Base de datos	357	Creación de una API REST	376
Seguridad	358	Uso de Flask para una API REST	377
API	358	Desarrollo de una API REST para base de datos	
Documentación	358	acceso	379
Presentamos el marco			
Flask	359	Estudio de caso: creación de una	
Construyendo una aplicación básica con routing	359	aplicación web con REST API 382	
Manejo de solicitudes con diferentes tipos de		Resumen 389	
métodos HTTP	361	Preguntas	390
Renderización de contenidos estáticos y dinámicos	365	Otras lecturas	390
Extraer parámetros de una solicitud HTTP		respuestas	390
	366		

11

Uso de Python para el desarrollo de microservicios

Requerimientos técnicos	392	Presentación de opciones de implementación	
Introduciendo microservicios	392	para microservicios	398
Aprendizaje de mejores prácticas		Desarrollo de una aplicación basada en	
para microservicios	395	microservicios de muestra	400
Creación de aplicaciones basadas		Resumen	413
en microservicios	397	Preguntas	413
Aprender opciones de desarrollo de		Otras lecturas	413
microservicios en Python	397	respuestas	414

12

Creación de funciones sin servidor con Python

Requisitos técnicos 416		Comprensión de las opciones de	
Presentación de funciones sin servidor 416		implementación para funciones sin servidor 419	
Beneficios	418	Aprender a crear funciones	
casos de uso	418	sin servidor	420

xiv Tabla de contenido

Creación de una nube basada en HTTP		Resumen	431
Función usando GCP Console	421	Preguntas	431
Estudio de caso: creación de una aplicación de notificación para eventos de almacenamiento en la nube	426	Otras lecturas	431
		respuestas	432

13**Python y aprendizaje automático**

Requisitos técnicos	434	Creación de un modelo de ML de muestra	441
Introducción al aprendizaje automático	434	Evaluación de un modelo mediante validación cruzada y ajuste fino de hiperparámetros	447
Uso de Python para el aprendizaje automático	437	Guardar un modelo de ML en un archivo	451
Introducción a las bibliotecas de aprendizaje automático en Python	437	Implementación y predicción de un modelo de ML en GCP Cloud	452
Las mejores prácticas de entrenamiento de datos con Pitón	439	Resumen	455
Creación y evaluación de un modelo de aprendizaje automático	439	Preguntas	456
Aprender sobre un proceso de creación de modelos de ML	440	Otras lecturas	456
		respuestas	457

14**Uso de Python para la automatización de redes**

Requerimientos técnicos	460	Interactuar con dispositivos de red usando Bibliotecas de Python basadas en SSH	467
Introducción a la automatización de la red	461	Interactuar con dispositivos de red usando NETCONF	477
Méritos y desafíos de la automatización de redes	462	Integración con sistemas de gestión de red	483
casos de uso	463	Uso de puntos finales de servicios de ubicación	484
Interactuar con dispositivos de red	464	Obtener un token de autenticación	485
Protocolos para interactuar con dispositivos de red	464	Obtener dispositivos de red y un inventario de interfaz	487
		Actualización del puerto del dispositivo de red	488

Integración con sistemas controlados por eventos	490	Resumen	495
Creación de suscripciones para Apache Kafka	492	Preguntas	496
Procesamiento de eventos de Apache Kafka	493	Otras lecturas	496
Renovación y eliminación de una suscripción	494	respuestas	497
Otros libros que puede disfrutar			
Índice			

Prefacio

Python es un lenguaje multipropósito que se puede utilizar para resolver cualquier problema medio a complejo en varios campos. Python for Geeks te enseñará cómo avanzar en tu carrera con la ayuda de trucos y consejos de expertos.

Comenzará explorando las diferentes formas de usar Python de manera óptima, tanto desde el punto de vista del diseño como de la implementación. A continuación, comprenderá el ciclo de vida de un proyecto de Python a gran escala. A medida que avance, se centrará en diferentes formas de crear un diseño elegante mediante la modularización de un proyecto de Python y aprenderá las prácticas recomendadas y los patrones de diseño para usar Python. También descubrirá cómo escalar Python más allá de un solo hilo y cómo implementar multiprocesamiento y multihilo en Python. Además de esto, comprenderá cómo no solo puede usar Python para implementar en una sola máquina, sino también usar clústeres en un entorno privado, así como en entornos de computación en la nube pública. Luego, explorará las técnicas de procesamiento de datos, se centrará en canalizaciones de datos escalables y reutilizables y aprenderá a usar estas técnicas avanzadas para la automatización de redes, las funciones sin servidor y el aprendizaje automático. Finalmente, se concentrará en crear estrategias de diseño de desarrollo web utilizando las técnicas y mejores prácticas cubiertas en el libro.

Al final de este libro de Python, podrá realizar una programación Python seria para proyectos complejos a gran escala.

para quien es este libro

Este libro es para desarrolladores de Python de nivel intermedio en cualquier campo que buscan desarrollar sus habilidades para desarrollar y administrar proyectos complejos a gran escala. Los desarrolladores que deseen crear módulos reutilizables y bibliotecas de Python y los desarrolladores de la nube que crean aplicaciones para la implementación en la nube también encontrarán útil este libro. La experiencia previa con Python lo ayudará a aprovechar al máximo este libro.

Lo que cubre este libro

El Capítulo 1, Ciclo de vida de desarrollo óptimo de Python, lo ayuda a comprender el ciclo de vida de un proyecto de Python típico y sus fases, con una discusión de las mejores prácticas para escribir código de Python.

El Capítulo 2, Uso de la modularización para manejar proyectos complejos, se enfoca en comprender los conceptos de módulos y paquetes en Python.

El Capítulo 3, Programación avanzada en Python orientada a objetos, explica cómo se pueden implementar los conceptos avanzados de la programación orientada a objetos mediante Python.

El Capítulo 4, Bibliotecas de Python para programación avanzada, explora conceptos avanzados como iteradores, generadores, manejo de errores y excepciones, manejo de archivos e inicio de sesión en Python.

El Capítulo 5, Pruebas y automatización con Python, presenta no solo diferentes tipos de automatización de pruebas, como pruebas unitarias, pruebas de integración y pruebas de sistemas, sino que también analiza cómo implementar pruebas unitarias utilizando marcos de prueba populares.

El Capítulo 6, Sugerencias y trucos avanzados en Python, analiza las funciones avanzadas de Python para la transformación de datos, la construcción de decoradores y también cómo usar estructuras de datos, incluidos pandas DataFrames para aplicaciones de análisis.

El Capítulo 7, Multiprocesamiento, multiproceso y programación asíncrona, lo ayuda a conocer las diferentes opciones para crear aplicaciones multiproceso o multiproceso mediante bibliotecas integradas en Python.

El Capítulo 8, Escalado horizontal de Python mediante clústeres, explora cómo trabajar con Apache Spark y cómo podemos escribir aplicaciones de Python para grandes aplicaciones de procesamiento de datos que se pueden ejecutar mediante un clúster de Apache Spark.

El Capítulo 9, Programación en Python para la nube, analiza cómo desarrollar e implementar aplicaciones en una plataforma en la nube y cómo usar Apache Beam en general y para Google Cloud Platform en particular.

El Capítulo 10, Uso de Python para desarrollo web y API REST, se centra en el uso del marco Flask para desarrollar aplicaciones web, interactuar con bases de datos y compilar API REST.

El Capítulo 11, Uso de Python para el desarrollo de microservicios, presenta los microservicios y cómo usar el marco Django para crear un microservicio de muestra e integrarlo con un microservicio basado en Flask.

El Capítulo 12, Creación de funciones sin servidor mediante Python, aborda la función de las funciones sin servidor en la informática en la nube y cómo crearlas mediante Python.

El Capítulo 13, Python y el aprendizaje automático, lo ayuda a comprender cómo usar Python para crear, entrenar y evaluar modelos de aprendizaje automático y cómo implementarlos en la nube.

El Capítulo 14, Uso de Python para la automatización de redes, analiza el uso de bibliotecas de Python para obtener datos de un dispositivo de **red y sistemas de gestión de red (NMS)** y para enviar datos de configuración a dispositivos o NMS.

Para aprovechar al máximo este libro

El conocimiento previo de Python es imprescindible para obtener beneficios reales de este libro.

Necesitará Python versión 3.7 o posterior instalada en su sistema. Todos los ejemplos de código se han probado con Python 3.7 y Python 3.8 y se espera que funcionen con cualquier versión futura de 3.x.

Una cuenta de Google Cloud Platform (una prueba gratuita funcionará bien) será útil para implementar algunos ejemplos de código en la nube.

Software/hardware covered in the book	Operating system requirements
Python release 3.7 or above	Windows
Apache Spark release 3.1.1	macOS
Cisco IOS XR (network device) release 7.12	Linux
Nokia Network Services Platform (NSP) release 21.6	
Google Cloud Platform Cloud SDK release 343.0.0	

Si está utilizando la versión digital de este libro, le recomendamos que escriba el código usted mismo o acceda al código desde el repositorio de GitHub del libro (hay un enlace disponible en la siguiente sección). Si lo hace, le ayudará a evitar posibles errores relacionados con el copiado y pegado de código.

Descargue los archivos de código de ejemplo

Puede descargar los archivos de código de ejemplo para este libro desde GitHub en <https://github.com/PacktPublishing/Python-for-Geeks>. Si hay una actualización del código, se actualizará en el repositorio de GitHub.

También tenemos otros paquetes de códigos de nuestro rico catálogo de libros y videos disponibles en <https://github.com/PacktPublishing/>. ¡Échales un vistazo!

Descarga las imágenes a color

También proporcionamos un archivo PDF que tiene imágenes en color de las capturas de pantalla y los diagramas utilizados en este libro. Puede descargarlo aquí: https://static.packt-cdn.com/downloads/9781801070119_ColorImages.pdf.

Convenciones utilizadas

Hay una serie de convenciones de texto utilizadas a lo largo de este libro.

Código en texto: Indica palabras de código en texto, nombres de tablas de bases de datos, nombres de carpetas, nombres de archivos, extensiones de archivos, nombres de rutas, direcciones URL ficticias, entrada de usuario y identificadores de Twitter. Aquí hay un ejemplo: "Monte el archivo de imagen de disco WebStorm-10*.dmg descargado como otro disco en su sistema".

Un bloque de código se establece de la siguiente manera:

```
recurso = {  
    "api_key": "AlzaSyDYKmm85kebxddKrGns4z0", "id":  
    "0B8TxHW2Ci6dbckVwTRtI3RUU", "campos": "archivos  
(nombre, id, webContentLink)",  
}
```

Cuando deseamos llamar su atención sobre una parte particular de un bloque de código, las líneas o elementos relevantes se muestran en negrita:

```
#casestudy1.py: Calculadora Pi desde la  
importación del operador agregar desde la  
importación aleatoria aleatoria  
  
desde pyspark.sql importar SparkSession  
  
chispa = SparkSession.constructor.  
master("spark://192.168.64.2:7077") \ .appName("Pi  
    claculator app") \ .getOrCreate()  
  
  
particiones = 2 n =  
10000000 * particiones  
  
función def(_):
```

```
x = aleatorio() * 2 - 1
y = aleatorio() * 2 - 1
devuelve 1 si x ** 2 + y          ** 2 <= 1 más 0
cuenta = chispa.chispaContexto.parallelizar(rango(1, n + 1),
particiones).mapa(función).reducir(agregar)
print("Pi es aproximadamente %f" % (4.0 * cuenta / n))
```

Cualquier entrada o salida de la línea de comandos se escribe de la siguiente manera:

Pi es aproximadamente 3.141479

Negrita: indica un nuevo término, una palabra importante o palabras que ve en pantalla. Por ejemplo, las palabras en los menús o cuadros de diálogo aparecen en **negrita**. Este es un ejemplo: "Como se mencionó anteriormente, Cloud Shell viene con una herramienta de edición que se puede iniciar con el botón **Abrir editor**".

Consejos o notas importantes

Aparecer así.

Ponerse en contacto

Los comentarios de nuestros lectores es siempre bienvenido.

Comentarios generales: si tiene preguntas sobre cualquier aspecto de este libro, envíenos un correo electrónico a customercare@packtpub.com y mencione el título del libro en el asunto de su mensaje.

Errata: Aunque hemos tomado todas las precauciones para garantizar la precisión de nuestro contenido, los errores ocurren. Si ha encontrado un error en este libro, le agradeceríamos que nos lo informara. Visite www.packtpub.com/support/errata y complete el formulario.

Piratería: si encuentra copias ilegales de nuestros trabajos en cualquier forma en Internet, le agradeceríamos que nos proporcionara la dirección de la ubicación o el nombre del sitio web.

Póngase en contacto con nosotros en copyright@packt.com con un enlace al material.

Si está interesado en convertirse en autor: si hay un tema en el que tiene experiencia y está interesado en escribir o contribuir a un libro, visite authors.packtpub.com.

Comparte tus pensamientos

Una vez que hayas leído Python para Geeks, ¡nos encantaría escuchar tus pensamientos! Haga clic aquí para ir directamente a la página de reseñas de Amazon de este libro y compartir sus comentarios.

Su revisión es importante para nosotros y la comunidad tecnológica y nos ayudará a asegurarnos de que ofrecemos contenido de excelente calidad.

Sección 1:

Python, más allá de lo básico

Comenzamos nuestro viaje explorando diferentes formas de usar Python de manera óptima, tanto desde el punto de vista del diseño como de la implementación. Brindamos una comprensión más profunda del ciclo de vida de un proyecto de Python a gran escala y sus fases. Una vez que tenemos esa comprensión, investigamos diferentes formas de crear un diseño elegante modularizando un proyecto de Python. Siempre que sea necesario, miramos debajo del capó para comprender el funcionamiento interno de Python. A esto le sigue una inmersión profunda en la programación orientada a objetos en Python.

Esta sección contiene los siguientes capítulos:

- Capítulo 1, Ciclo de vida óptimo de desarrollo de Python
- Capítulo 2, Uso de la modularización para manejar proyectos complejos
- Capítulo 3, Programación Python avanzada orientada a objetos

1

Python óptimo Desarrollo Ciclo vital

Teniendo en cuenta su experiencia previa con Python, hemos omitido los detalles introductorios del lenguaje Python en este capítulo. Primero, tendremos una breve discusión sobre la comunidad Python de código abierto más amplia y su cultura específica. Esta introducción es importante, ya que esta cultura se refleja en el código escrito y compartido por la comunidad de Python. Luego, presentaremos las diferentes fases de un proyecto típico de Python. A continuación, veremos diferentes formas de crear estrategias para el desarrollo de un proyecto típico de Python.

Más adelante, exploraremos diferentes formas de documentar el código de Python. Más adelante, analizaremos varias opciones para desarrollar un esquema de nomenclatura efectivo que pueda ayudar mucho a mejorar el mantenimiento del código. También analizaremos varias opciones para usar el control de fuente para proyectos de Python, incluidas situaciones en las que los desarrolladores usan principalmente portátiles Jupyter para el desarrollo. Finalmente, exploraremos las mejores prácticas para implementar el código para su uso, una vez que se haya desarrollado y probado.

4 Ciclo de vida óptimo de desarrollo de Python

Cubriremos los siguientes temas en este capítulo:

- Cultura y comunidad Python
- Diferentes fases de un proyecto de Python
- Elaboración de estrategias para el proceso de desarrollo
- Documentación efectiva del código Python
- Desarrollar un esquema de nombres eficaz
- Exploración de opciones para el control de fuente
- Comprender las estrategias para implementar el código
- Entornos de desarrollo Python

Este capítulo lo ayudará a comprender el ciclo de vida de un proyecto típico de Python y sus fases para que pueda utilizar completamente el poder de Python.

Cultura y comunidad Python

Python es un lenguaje de alto nivel interpretado que fue desarrollado originalmente por Guido van Rossum en 1991. La comunidad de Python es especial en el sentido de que presta mucha atención a cómo se escribe el código. Por eso, desde los primeros días de Python, la comunidad de Python ha creado y mantenido un sabor particular en su filosofía de diseño. Hoy en día, Python se usa en una amplia variedad de industrias, desde la educación hasta la medicina. Pero independientemente de la industria en la que se use, la cultura particular de la vibrante comunidad de Python generalmente se considera parte integral de los proyectos de Python.

En particular, la comunidad de Python quiere que escribamos código simple y evitemos la complejidad siempre que sea posible. De hecho, hay un adjetivo, *Pythonic*, que significa que hay varias formas de realizar una determinada tarea, pero hay una forma preferida según las convenciones de la comunidad de Python y según la filosofía fundacional del lenguaje.

Los nerds de Python hacen todo lo posible para crear artefactos que sean lo más pitónicos posible. Obviamente, el código no pitónico significa que no somos buenos codificadores a los ojos de estos nerds. En este libro, intentaremos ser lo más pitónicos posible en nuestro código y diseño.

Y también hay algo oficial en ser *Pythonic*. Tim Peters ha escrito de manera concisa la filosofía de Python en un breve documento, *The Zen of Python*. Sabemos que se dice que Python es uno de los lenguajes más fáciles de leer y *The Zen of Python* quiere que siga siendo así. Espera que Python sea explícito a través de una buena documentación y lo más limpio y claro posible. Podemos leer *The Zen of Python* nosotros mismos, como se explica a continuación.

Para leer The Zen of Python, abra una consola de Python y ejecute la importación de este comando, como se muestra en la siguiente captura de pantalla:

```
[1] import this
this

[+] The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
<module 'this' from '/usr/lib/python3.6>this.py'>
```



Figura 1.1 – El Zen de Python

El Zen de Python parece ser un texto críptico descubierto en una antigua tumba egipcia.

Aunque está escrito deliberadamente de esta manera críptica e informal, cada línea de texto tiene un significado más profundo. En realidad, mire más de cerca: se puede usar como una guía para codificar en Python.

Nos referiremos a diferentes líneas de El Zen de Python a lo largo del libro. Veamos primero algunos extractos de él, como sigue:

- **Hermoso es mejor que feo:** es importante escribir código que esté bien escrito, legible y autoexplicativo. No solo debería funcionar, sino que debería estar bellamente escrito. Al codificar, debemos evitar el uso de atajos en favor de un estilo que se explique por sí mismo.

6 Ciclo de vida óptimo de desarrollo de Python

- Lo **simple es mejor que lo complejo**: no debemos complicarnos innecesariamente cosas. Siempre que nos enfrentemos a una elección, debemos preferir la solución más sencilla. Se desaconsejan las formas nerd, innecesarias y complicadas de escribir código. Incluso cuando agrega algunas líneas más al código fuente, lo más simple sigue siendo mejor que la alternativa compleja.
- **Debería haber una, y preferiblemente sólo una, forma obvia de hacerlo**: en términos más amplios, para un problema dado debería haber una mejor solución posible. Debemos esforzarnos por descubrir esto. A medida que iteramos a través del diseño para mejorarlo, independientemente de nuestro enfoque, se espera que nuestra solución evolucione y converja hacia esa solución preferible.
- **Ahora es mejor que nunca**: en lugar de esperar a la perfección, comenzemos a resolver el problema dado utilizando la información, las suposiciones, las habilidades, las herramientas y la infraestructura que tenemos. A través del proceso de iteración, seguiremos mejorando la solución. Mantengamos las cosas en movimiento en lugar de permanecer inactivas. No se afloje mientras espera el momento perfecto. Lo más probable es que el momento perfecto nunca llegue.
- **Explícito es mejor que implícito**: El código debe explicarse por sí mismo tanto como sea posible. Esto debe reflejarse en la elección de los nombres de las variables, la clase y el diseño de la función, así como en la arquitectura general **de extremo a extremo (E2E)**. Es mejor errar por el lado de la precaución. Hazlo siempre más explícito cuando te enfrentes a una elección.
- **Plana es mejor que anidada**: una estructura anidada es concisa pero también genera confusión. Prefiere una estructura plana siempre que sea posible.

Diferentes fases de un proyecto de Python

Antes de discutir el ciclo de vida de desarrollo óptimo, comencemos identificando las diferentes fases de un proyecto de Python. Cada fase se puede considerar como un grupo de actividades de naturaleza similar, como se ilustra en el siguiente diagrama:

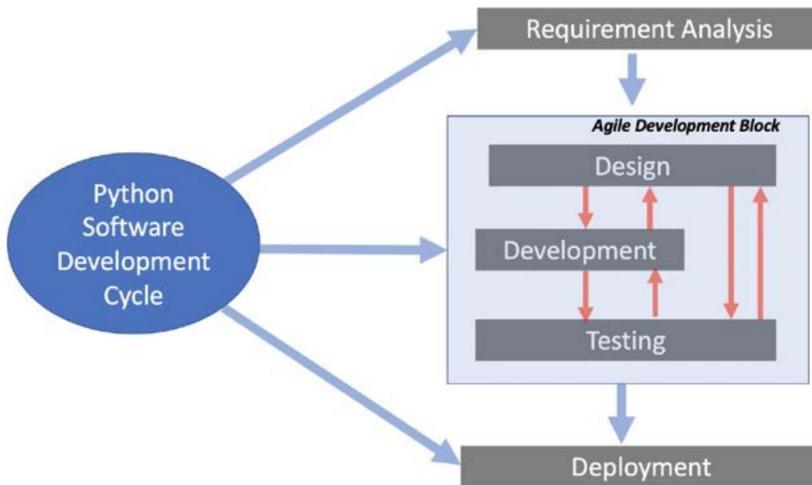


Figura 1.2 – Varias fases de un proyecto de Python

Las diversas fases de un proyecto típico de Python se describen aquí:

- **Análisis de requisitos:** esta fase consiste en recopilar los requisitos de todas las partes interesadas clave y luego analizarlos para comprender qué se debe hacer y luego pensar en el cómo. Las partes interesadas pueden ser nuestros usuarios reales del software o propietarios de negocios. Es importante recopilar los requisitos con el mayor detalle posible. Siempre que sea posible, los requisitos deben establecerse, comprenderse y analizarse por completo con el usuario final y las partes interesadas antes de comenzar el diseño y el desarrollo.

Un punto importante es asegurarse de que la fase de análisis de requisitos se mantenga fuera del ciclo iterativo de las fases de diseño, desarrollo y prueba. El análisis de requisitos debe llevarse a cabo y completarse antes de pasar a las siguientes fases. Los requisitos deben incluir tanto **los requisitos funcionales (FR) y requisitos no funcionales (NFR)**. Los FR deben agruparse en módulos. Dentro de cada módulo, los requisitos deben enumerarse en un esfuerzo por mapearlos lo más cerca posible de los módulos del código. • **Diseño:** El diseño es nuestra respuesta técnica a los requisitos establecidos en el fase de requerimiento. En la fase de diseño, averiguamos la parte del cómo de la ecuación. Es un proceso creativo en el que usamos nuestra experiencia y habilidades para crear el conjunto y la estructura correctos de módulos y las interacciones entre ellos de la manera más eficiente y óptima.

8 Ciclo de vida óptimo de desarrollo de Python

Tenga en cuenta que crear el diseño correcto es una parte importante de un proyecto de Python. Cualquier paso en falso en la fase de diseño será mucho más costoso de corregir que los pasos en falso en fases posteriores. En cierta medida, se necesita 20 veces más esfuerzo para cambiar el diseño e implementar los cambios de diseño en las fases posteriores (por ejemplo, la fase de codificación), en comparación con un grado similar de cambios si ocurren en la fase de codificación, por ejemplo, la incapacidad para identificar correctamente las clases o averiguar los datos correctos y calcular la dimensión del proyecto tendrá un gran impacto en comparación con un error al implementar una función. Además, debido a que generar el diseño correcto es un proceso conceptual, es posible que los errores no sean obvios y que las pruebas no los detecten. Por otro lado, los errores en la codificación serán detectados por un sistema de manejo de excepciones bien pensado.

En la fase de diseño, realizamos las siguientes actividades:

- a) Diseñamos la estructura del código e identificamos los módulos dentro del código.
- b) Decidimos el enfoque fundamental y decidimos si debemos usar programación funcional, programación orientada a objetos o un enfoque híbrido.
- c) También identificamos las clases y funciones y elegimos los nombres de estos componentes de nivel superior.

También producimos documentación de alto nivel.

• **Codificación:** Esta es la fase donde implementaremos el diseño usando Python. Comenzamos implementando las abstracciones, componentes y módulos de nivel superior identificados primero por el diseño, seguidos por la codificación detallada. Mantendremos una discusión sobre la fase de codificación al mínimo en esta sección, ya que la discutiremos extensamente a lo largo del libro.

• **Testing:** Testing es el proceso de verificación de nuestro código.

• **Implementación:** una vez probada exhaustivamente, debemos entregar la solución al usuario final. El usuario final no debe ver los detalles de nuestro diseño, codificación o prueba. La implementación es el proceso de proporcionar una solución al usuario final que se puede utilizar para resolver el problema como se detalla en los requisitos. Por ejemplo, si estamos trabajando para desarrollar un proyecto de **aprendizaje automático (ML)** para predecir las precipitaciones en Ottawa, la implementación se trata de descubrir cómo proporcionar una solución utilizable para el usuario final.

Habiendo entendido cuáles son las diferentes fases de un proyecto, pasaremos a ver cómo podemos crear una estrategia para el proceso general.

Estrategia del proceso de desarrollo

La elaboración de estrategias para el proceso de desarrollo consiste en planificar cada una de las fases y observar el flujo del proceso de una fase a otra. Para diseñar una estrategia del proceso de desarrollo, primero debemos responder las siguientes preguntas:

1. ¿Buscamos un enfoque de diseño mínimo e ir directamente a la fase de codificación con poco diseño?
2. ¿Queremos **un desarrollo basado en pruebas (TDD)**, mediante el cual primero creamos pruebas utilizando los requisitos y luego las codificamos?
3. ¿Queremos crear un **producto mínimo viable (MVP)** primero y evolucionar iterativamente? ¿la solución?
4. ¿Cuál es la estrategia para validar NFR como seguridad y rendimiento?
5. ¿Buscamos un desarrollo de un solo nodo o queremos desarrollar e implementar ¿en el clúster o en la nube?
6. ¿Cuál es el volumen, la velocidad y la variedad de nuestros datos de **entrada y salida (E/S)** ?
¿Es un **sistema de archivos distribuido de Hadoop (HDFS)** o una estructura basada en archivos de **Simple Storage Service (S3)** , o un **lenguaje de consulta estructurado (SQL)** o una base de datos NoSQL?
¿Los datos están en las instalaciones o en la nube?
7. ¿Estamos trabajando en casos de uso especializados como ML con requisitos específicos para crear canalizaciones de datos, probar modelos e implementarlos y mantenerlos?

Con base en las respuestas a estas preguntas, podemos diseñar estrategias para los pasos de nuestro proceso de desarrollo. En tiempos más recientes, siempre se prefiere utilizar procesos de desarrollo iterativos de una forma u otra. El concepto de MVP como objetivo inicial también es popular. Los discutiremos en las siguientes subsecciones, junto con las necesidades de desarrollo específicas de los dominios.

Iterando a través de las fases

La filosofía moderna de desarrollo de software se basa en ciclos iterativos cortos de diseño, desarrollo y prueba. El modelo de cascada tradicional que se utilizó en el desarrollo de código murió hace mucho tiempo. Seleccionar la granularidad, el énfasis y la frecuencia correctos de estas fases depende de la naturaleza del proyecto y de nuestra elección de estrategia de desarrollo de código. Si queremos elegir una estrategia de desarrollo de código con un diseño mínimo y queremos pasar directamente a la codificación, entonces la fase de diseño es delgada. Pero incluso comenzar el código de inmediato requerirá pensar un poco en términos del diseño de los módulos que eventualmente se implementarán.

10 ciclo de vida óptimo de desarrollo de Python

Independientemente de la estrategia que elijamos, existe una relación iterativa inherente entre las fases de diseño, desarrollo y prueba. Inicialmente comenzamos con la fase de diseño, lo implementamos en la fase de codificación y luego lo validamos probándolo. Una vez que hayamos señalado las deficiencias, debemos volver a la mesa de dibujo revisando la fase de diseño.

Apuntando a MVP primero

A veces, seleccionamos un pequeño tema de los requisitos más importantes para implementar primero el MVP con el objetivo de mejorarlo iterativamente. En un proceso iterativo, diseñamos, codificamos y probamos hasta que creamos un producto final que se puede implementar y usar.

Ahora, hablemos de cómo implementaremos la solución de algunos dominios especializados en Python.

Desarrollo de estrategias para dominios especializados

Python se está utilizando actualmente para una amplia variedad de escenarios. Analicemos los siguientes cinco casos de uso importantes para ver cómo podemos diseñar estrategias para el proceso de desarrollo para cada uno de ellos de acuerdo con sus necesidades específicas:

- ML
- Computación en la nube y computación en clúster
- Programación de sistemas
- Programación de redes
- Computación sin servidor

Hablaremos de cada uno de ellos en las siguientes secciones.

ML

A lo largo de los años, Python se ha convertido en el lenguaje más común utilizado para implementar algoritmos de ML. Los proyectos de ML deben tener un entorno bien estructurado. Python tiene una amplia colección de bibliotecas de alta calidad que están disponibles para su uso en ML.

Para un proyecto de ML típico, existe un **proceso estándar entre industrias para la minería de datos (CRISP-DM)** ciclo de vida que especifica varias fases de un proyecto ML. Un ciclo de vida CRISP-DM se ve así:

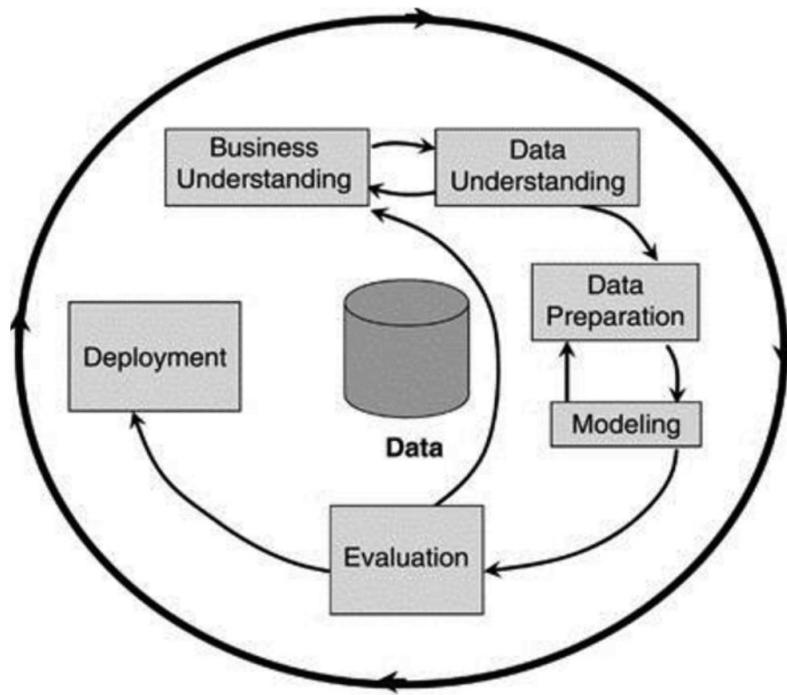


Figura 1.3 – Un ciclo de vida CRISP-DM

Para los proyectos de ML, se estima que el diseño y la implementación de canalizaciones de datos representan casi el 70 % del esfuerzo de desarrollo. Al diseñar canalizaciones de procesamiento de datos, debemos tener en cuenta que las canalizaciones idealmente tendrán estas características:

- Deben ser escalables.
- Deben ser reutilizables en la medida de lo posible.
- Deben procesar datos tanto de transmisión como por lotes conforme a los estándares de **Apache Beam**.
- En su mayoría, deberían ser una concatenación de funciones de ajuste y transformación, como veremos discutir en el Capítulo 6, Sugerencias y trucos avanzados en Python.

Además, una parte importante de la fase de prueba de los proyectos de ML es la evaluación del modelo. Necesitamos averiguar cuál de las métricas de rendimiento es la mejor para cuantificar el rendimiento del modelo según los requisitos del problema, la naturaleza de los datos y el tipo de algoritmo que se está implementando. ¿Estamos analizando la exactitud, la precisión, la recuperación, la puntuación de F1 o una combinación de estas métricas de rendimiento? La evaluación del modelo es una parte importante del proceso de prueba y debe realizarse además de las pruebas estándar realizadas en otros proyectos de software.

12 Ciclo de vida óptimo de desarrollo de Python

Computación en la nube y computación en clúster

La computación en la nube y la computación en clúster agregan complejidad adicional a la subyacente infraestructura. Los proveedores de servicios en la nube ofrecen servicios que necesitan bibliotecas especializadas. La arquitectura de Python, que comienza con paquetes básicos mínimos y la capacidad de importar cualquier paquete adicional, lo hace ideal para la computación en la nube. La independencia de la plataforma que ofrece un entorno de Python es fundamental para la computación en la nube y en clúster. **Python** es el lenguaje elegido por **Amazon Web Services (AWS)**, Windows Azure y Google Cloud Platform (GCP).

Los proyectos de computación en la nube y computación en clúster tienen entornos de desarrollo, prueba y producción separados. Es importante mantener sincronizados los entornos de desarrollo y producción.

Cuando se usa **infraestructura como servicio (IaaS)**, los contenedores Docker pueden ayudar mucho y se recomienda usarlos. Una vez que estemos usando el contenedor Docker, no importa dónde estemos ejecutando el código, ya que el código tendrá exactamente el mismo entorno y dependencias.

Programación de sistemas

Python tiene interfaces para los servicios del sistema operativo. Sus bibliotecas principales tienen enlaces **de interfaz de sistema operativo portátil (POSIX)** que permiten a los desarrolladores crear las llamadas herramientas de shell, que se pueden usar para la administración del sistema y varias utilidades. Las herramientas de shell escritas en Python son compatibles en varias plataformas. La misma herramienta se puede usar en Linux, Windows y macOS sin ningún cambio, lo que los hace bastante potentes y fáciles de mantener.

Por ejemplo, una herramienta de shell que copia un directorio completo desarrollado y probado en Linux puede ejecutarse sin cambios en Windows. El soporte de Python para la programación de sistemas incluye lo siguiente:

- Definición de variables de entorno
- Compatibilidad con archivos, sockets, conductos, procesos y varios subprocesos
- Capacidad para especificar una **expresión regular (regex)** para la coincidencia de patrones
- Capacidad para proporcionar argumentos de línea de comandos
- Compatibilidad con interfaces de flujo estándar, lanzadores de comandos de shell y expansión de nombre de archivo
- Capacidad para comprimir utilidades de archivos
- Capacidad para analizar el lenguaje de **marcado extensible (XML)** y el **objeto JavaScript**
Archivos de **notación (JSON)**

Cuando se usa Python para el desarrollo del sistema, la fase de implementación es mínima y puede ser tan simple como empaquetar el código como un archivo ejecutable. Es importante mencionar que Python no está destinado a ser utilizado para el desarrollo de controladores a nivel de sistema o bibliotecas de sistemas operativos.

Programación de red

En la era de la transformación digital, en la que los sistemas **de tecnología de la información (TI)** avanzan rápidamente hacia la automatización, las redes se consideran el principal cuello de botella en la automatización completa. La razón de esto es la propiedad de los sistemas operativos de red de diferentes proveedores y la falta de apertura, pero los requisitos previos de la transformación digital están cambiando esta tendencia y se está trabajando mucho para hacer que la red sea programable y consumible como un servicio (**red- como servicio o NaaS**). La verdadera pregunta es: ¿Podemos usar Python para la programación de redes? La respuesta es un gran Sí. De hecho, es uno de los lenguajes más populares en uso para la automatización de redes.

El soporte de Python para la programación de redes incluye lo siguiente:

- Programación de sockets, incluido **el Protocolo de control de transmisión (TCP) y Usuario Conectores de protocolo de datagramas (UDP)**
- Compatibilidad con la comunicación entre el cliente y el servidor
- Compatibilidad con puerto de escucha y procesamiento de datos
- Ejecución de comandos en un sistema remoto **Secure Shell (SSH)**
- Carga y descarga de archivos mediante el **Protocolo de copia segura (SCP)/ Transferencia de archivos Protocolo (FTP)**
- Compatibilidad con la biblioteca para **el Protocolo simple de administración de redes (SNMP)**
- Soporte para la **Transferencia de Estado Representacional (RESTCONF) y Red Protocolos de configuración (NETCONF)** para recuperar y actualizar la configuración

Computación sin servidor

La computación sin servidor es un modelo de ejecución de aplicaciones basado en la **nube en el que los proveedores de servicios** en la nube (**CSP**) proporcionan los recursos informáticos y los servidores de aplicaciones para permitir que los desarrolladores implementen y ejecuten las aplicaciones sin la molestia de administrar los recursos informáticos y los servidores. Todos los principales proveedores de nube pública (Microsoft Azure Serverless Functions, AWS Lambda y **Google Cloud Platform o GCP**) admiten la computación sin servidor para Python.

14 Ciclo de vida óptimo de desarrollo de Python

Necesitamos entender que todavía hay servidores en un entorno sin servidor, pero esos servidores son administrados por CSP. Como desarrollador de aplicaciones, no somos responsables de la instalación y el mantenimiento de los servidores, ni tampoco tenemos responsabilidad directa por la escalabilidad y el rendimiento de los servidores.

Hay bibliotecas y marcos sin servidor populares disponibles para Python. Estos se describen a continuación:

- **Sin servidor:** Serverless Framework es un marco de código abierto para aplicaciones sin servidor. Funciones o servicios AWS Lambda y está escrito usando Node.js. Serverless es el primer marco desarrollado para crear aplicaciones en AWS Lambda.
- **Chalice:** este es un microframework sin servidor de Python desarrollado por AWS. Este es una opción predeterminada para los desarrolladores que desean poner en marcha e implementar rápidamente sus aplicaciones de Python con los servicios de AWS Lambda, ya que esto le permite poner en marcha e implementar rápidamente una aplicación sin servidor en funcionamiento que se escala hacia arriba y hacia abajo por sí sola según sea necesario, utilizando AWS Lambda. Otra característica clave de Chalice es que proporciona una utilidad para simular su aplicación localmente antes de enviarla a la nube.
- **Zappa:** esta es más una herramienta de implementación integrada en Python y hace que el implementación de su **aplicación Web Server Gateway Interface (WSGI)** fácil.

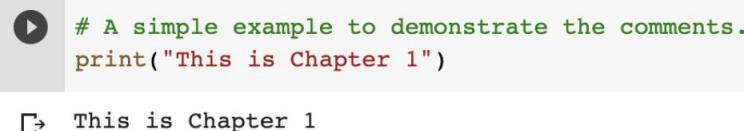
Ahora, veamos formas efectivas de desarrollar código Python.

Documentación efectiva del código Python

Siempre es importante encontrar una forma eficaz de documentar el código. El desafío es desarrollar una forma integral pero simple de desarrollar código Python. Primero veamos los comentarios de Python y luego las cadenas de documentación.

Comentarios de Python

A diferencia de una cadena de documentos, los comentarios de Python no son visibles para el compilador en tiempo de ejecución. Se utilizan como una nota para explicar el código. Los comentarios comienzan con el signo # en Python, como se muestra en la siguiente captura de pantalla:



```
# A simple example to demonstrate the comments.
print("This is Chapter 1")
```

C → This is Chapter 1

Figura 1.4 – Un ejemplo de un comentario en Python

cadena de documentación

El caballo de batalla principal para documentar el código es el bloque de comentarios de varias líneas llamado **docstring**. Una de las características del lenguaje Python es que los DocStrings están asociados con un objeto y están disponibles para su inspección. Las pautas para DocStrings se describen en la [Propuesta de mejora de Python \(PEP\) 257](#). De acuerdo con las pautas, su propósito es brindar una descripción general a los lectores. Deben tener un buen equilibrio entre ser concisos y elaborados. DocStrings utiliza un formato de cadena de comillas dobles triples: ("").

Aquí hay algunas pautas generales al crear una cadena de documentos:

- Se debe colocar una cadena de documentación justo después de la función o la definición de clase.
- Una cadena de documentación debe tener un resumen de una línea seguido de una descripción más detallada.
- Los espacios en blanco deben usarse estratégicamente para organizar los comentarios, pero no se debe abusar. Puede usar líneas en blanco para organizar el código, pero no las use en exceso.

En las siguientes secciones, echemos un vistazo a conceptos más detallados de docStrings.

Estilos de cadena de documentación

Una cadena de documentación de Python tiene los siguientes estilos ligeramente diferentes:

- Google
- NumPy/SciPy
- Epitexto
- Reestructurado

Tipos de cadenas de documentación

Mientras se desarrolla el código, es necesario producir varios tipos de documentación, incluidos los siguientes:

- Comentarios línea por línea
- Documentación funcional o de nivel de clase
- Detalles algorítmicos

Discutámoslos, uno por uno.

16 Ciclo de vida óptimo de desarrollo de Python

Comentario línea por línea

Un uso simple de una cadena de documentos es usarlo para crear comentarios de varias líneas, como se muestra aquí:



```
"""
This is a comment
written in
more than just one line
"""

print("Chapter 2 will be about Python Modules")
```

⇒ Chapter 2 will be about Python Modules

Figura 1.5: un ejemplo de una cadena de documentación de tipo comentario línea por línea

Documentación funcional o de nivel de clase

Un uso poderoso de una cadena de documentos es para la documentación funcional o de nivel de clase. Si colocamos la cadena de documentos justo después de la definición de una función o una clase, Python asocia la cadena de documentos con la función o una clase. Esto se coloca en el atributo `__doc__` de esa función o clase en particular. Podemos imprimir eso en tiempo de ejecución usando el `__doc__` atributo o usando la función de ayuda `help`, como se muestra en el siguiente ejemplo:

```
[5] def double(n):
    '''Takes in a number n, returns the double of its value'''
    return n**2
```

```
[6] print(double.__doc__)
```

⇒ Takes in a number n, returns the double of its value

```
[8] help(double)
```

⇒ Help on function double in module __main__:

```
double(n)
Takes in a number n, returns the double of its value
```

Figura 1.6 – Un ejemplo de la función de ayuda

Cuando se utiliza una cadena de documentos para documentar clases, la estructura recomendada es la siguiente:

- Un resumen: generalmente una sola línea
- Primera línea en blanco
- Cualquier explicación adicional sobre la cadena de documentación
- Segunda línea en blanco

Aquí se muestra un ejemplo del uso de una cadena de documentos en el nivel de clase:

```
▶ class ComplexNumber:  
    """  
        This is a class for mathematical operations on complex numbers.  
  
        Attributes:  
            real (int): The real part of complex number.  
            imag (int): The imaginary part of complex number.  
    """  
  
    def __init__(self, real, imag):  
        """  
            The constructor for ComplexNumber class.  
  
            Parameters:  
                real (int): The real part of complex number.  
                imag (int): The imaginary part of complex number.  
        """  
  
    def add(self, num):  
        """  
            The function to add two Complex Numbers.  
  
            Parameters:  
                num (ComplexNumber): The complex number to be added.  
  
            Returns:  
                ComplexNumber: A complex number which contains the sum.  
        """  
  
        re = self.real + num.real  
        im = self.imag + num.imag  
  
        return ComplexNumber(re, im)
```

Figura 1.7: un ejemplo de una cadena de documentación a nivel de clase

18 Ciclo de vida óptimo de desarrollo de Python

Detalles algorítmicos

Cada vez con más frecuencia, los proyectos de Python utilizan análisis descriptivos o predictivos y otra lógica compleja. Los detalles del algoritmo que se utiliza deben especificarse claramente con todas las suposiciones que se hicieron. Si un algoritmo se implementa como una función, entonces el mejor lugar para escribir el resumen de la lógica del algoritmo es antes de la firma de la función.

Desarrollo de un esquema de nomenclatura efectivo

Si desarrollar e implementar la lógica correcta en el código es ciencia, entonces hacerlo bonito y legible es un arte. Los desarrolladores de Python son famosos por prestar especial atención al esquema de nombres y por incluir el Zen de Python. Python es uno de los pocos lenguajes que tiene pautas completas sobre el esquema de nombres escrito por Guido van Rossum.

Están escritos en un documento PEP 8 que tiene una sección completa sobre convenciones de nomenclatura, seguida de muchas bases de código. PEP 8 tiene pautas de nomenclatura y estilo que se sugieren. Puede leer más sobre esto en <https://www.Python.org/dev/peps/pep-0008/>.

El esquema de nombres sugerido en PEP 8 se puede resumir de la siguiente manera:

- En general, todos los nombres de los módulos deben estar en minúsculas.
- Todos los nombres de clases y excepciones deben ser CamelCase.
- Todas las variables globales y locales deben estar en minúsculas.
- Todos los nombres de funciones y métodos deben estar en minúsculas.
- Todas las constantes deben ser ALL_UPPER_CASE.

Aquí se dan algunas pautas sobre la estructura del código de PEP 8:

- La sangría es importante en Python. No use tabulador para sangría. En su lugar, utilice cuatro espacios.
- Limite el anidamiento a cuatro niveles.
- Recuerde limitar el número de líneas a 79 caracteres. Use el símbolo \ para romper líneas largas.
- Para que el código sea legible, inserte dos líneas en blanco para separar las funciones.
- Insertar una sola línea negra entre varias secciones lógicas.

Recuerde que las pautas de PEP son solo sugerencias que pueden ser personalizadas por diferentes equipos. Cualquier esquema de nomenclatura personalizado aún debe usar PEP 8 como guía básica.

Ahora, veamos con más detalle el esquema de nombres en el contexto de varias estructuras del lenguaje Python.

Métodos

Los nombres de los métodos deben usar minúsculas. El nombre debe constar de una sola palabra o más de una palabra separadas por guiones bajos. Puedes ver un ejemplo de esto aquí:

`calcular_suma`

Para que el código sea legible, el método debe ser preferiblemente un verbo, relacionado con el procesamiento que se supone que debe realizar el método.

Si un método no es público, debe tener un guión bajo al principio. He aquí un ejemplo de esto:

`_mi_calcular_suma`

Los métodos **Dunder o mágicos** son métodos que tienen un guión bajo al principio y al final.

Aquí se muestran ejemplos de Dunder o métodos mágicos:

- `__init__`
- `__añadir__`

Nunca es una buena idea usar dos guiones bajos al principio y al final para nombrar un método, y se desaconseja su uso por parte de los desarrolladores. Dicho esquema de nombres está diseñado para los métodos de Python.

Variables

Use una palabra en minúscula o palabras separadas por un guión bajo para representar variables.

Las variables deben ser sustantivos que correspondan a la entidad que representan.

Aquí se dan ejemplos de variables:

- `x`
- `mi_var`

Los nombres de las variables privadas deben comenzar con un guión bajo. Un ejemplo es `_mi_variable_secreta`.

20 Ciclo de vida óptimo de desarrollo de Python

Variables booleanas

Comenzar una variable booleana con is o has la hace más legible. Puedes ver un par de ejemplos de esto aquí:

```
clase Paciente:
```

```
    es_admitido = Falso
```

```
    has_heartbeat = Falso
```

Variables de colección

Dado que las colecciones son cubos de variables, es una buena idea nombrarlas en formato plural, como se ilustra aquí:

```
clase Paciente:
```

```
    pacientes_admitidos = ['Juan','Pedro']
```

Variables de diccionario

Se recomienda que el nombre del diccionario sea lo más explícito posible. Por ejemplo, si tenemos un diccionario de personas asignadas a las ciudades en las que viven, se puede crear un diccionario de la siguiente manera:

```
personas_ciudades = {'Imran': 'Ottawa', 'Steven': 'Los  
Ángeles'}
```

Constante

Python no tiene variables inmutables. Por ejemplo, en C++, podemos especificar una const palabra clave para especificar que la variable es inmutable y es una constante. Python se basa en convenciones de nomenclatura para especificar constantes. Si el código intenta tratar una constante como una variable regular, Python no dará un error.

Para constantes, la recomendación es utilizar palabras en mayúsculas o palabras separadas por un guión bajo. Un ejemplo de una constante se da aquí:

```
FACTOR DE CONVERSIÓN
```

Clases

Las clases deben seguir el estilo CamelCase; en otras palabras, deben comenzar con una letra mayúscula. Si necesitamos usar más de una palabra, las palabras no deben estar separadas por un guión bajo, sino que cada palabra que se agregue debe tener una letra mayúscula inicial.

Las clases deben usar un sustantivo y deben nombrarse de manera que representen mejor la entidad a la que corresponde la clase. Una forma de hacer que el código sea legible es usar clases con sufijos que tengan algo que ver con su tipo o naturaleza, como las siguientes:

- Motor Hadoop
- Tipo Parquet
- Widget de cuadro de texto

Aquí hay algunos puntos a tener en cuenta:

- Hay clases de excepción que manejan errores. Sus nombres siempre deben tener Error como palabra final. He aquí un ejemplo de esto:

Error de archivo no encontrado

- Algunas de las clases integradas de Python no siguen esta directriz de nomenclatura.
- Para hacerlo más legible, para clases base o abstractas, se puede usar un prefijo Base o Abstract . Un ejemplo podría ser este:

ResumenCoche

BaseClass

Paquetes

No se recomienda el uso de guiones bajos al nombrar un paquete. El nombre debe ser corto y todo en minúsculas. Si se necesita usar más de una palabra, la palabra o palabras adicionales también deben estar en minúsculas. He aquí un ejemplo de esto:

mi paquete

Módulos

Al nombrar un módulo, se deben usar nombres cortos y directos. Deben estar en minúsculas y más de una palabra se unirá con guiones bajos. Aquí hay un ejemplo:

módulo_principal.py

22 Ciclo de vida óptimo de desarrollo de Python

Convenciones de importación

A lo largo de los años, la comunidad de Python ha desarrollado una convención para los alias que se utilizan para paquetes de uso común. Puedes ver un ejemplo de esto aquí:

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

Argumentos

Se recomienda que los argumentos tengan una convención de nomenclatura similar a las variables, porque los argumentos de una función son, de hecho, variables temporales.

Herramientas útiles

Hay un par de herramientas que se pueden usar para probar qué tan cerca se ajusta su código a las pautas de PEP 8. Veámoslos, uno por uno.

pilinto

Pylint se puede instalar ejecutando el siguiente comando:

```
$ pip instalar pylint
```

Pylint es un analizador de código fuente que verifica la convención de nomenclatura del código con respecto a PEP 89. Luego, imprime un informe. Se puede personalizar para utilizarlo en otras convenciones de nomenclatura.

PEP 8

PEP 8 se puede instalar ejecutando el siguiente comando:

```
pip: $ pip instalar pep8
```

pep8 comprueba el código con respecto a PEP 8.

Hasta ahora, hemos aprendido acerca de las diversas convenciones de nomenclatura en Python. A continuación, exploraremos diferentes opciones para usar el control de fuente para Python.

Exploración de opciones para el control de código fuente

Primero, veremos una breve historia de los sistemas de control de fuente para proporcionar un contexto. Los sistemas de control de fuente modernos son bastante poderosos. La evolución de los sistemas de control de fuentes pasó por las siguientes etapas:

- **Etapa 1:** el código fuente fue iniciado inicialmente por los sistemas de control de fuente locales que fueron almacenados en un disco duro. Esta colección de código local se denominó repositorio local.
- **Etapa 2:** Pero el uso local del control de código fuente no era adecuado para equipos más grandes. Este La solución eventualmente se convirtió en un repositorio central basado en un servidor que fue compartido por los miembros del equipo que trabajaban en un proyecto en particular. Resolvió el problema de compartir el código entre los miembros del equipo, pero también creó el desafío adicional de bloquear los archivos para el entorno multiusuario.
- **Etapa 3:** Los repositorios modernos de control de versiones, como Git, desarrollaron aún más este modelo. Todos los miembros de un equipo ahora tienen una copia completa del repositorio almacenado. Los miembros del equipo ahora trabajan fuera de línea en el código. Necesitan conectarse al repositorio solo cuando es necesario compartir el código.

¿Qué no pertenece al repositorio de control de código fuente?

Veamos lo que no debe registrarse en el repositorio de control de código fuente.

En primer lugar, nada que no sea el archivo de código fuente debe registrarse. Los archivos generados por computadora no deben registrarse en el control de código fuente. Por ejemplo, supongamos que tenemos un archivo fuente de Python llamado main.py. Si lo compilamos, el código generado no pertenece al repositorio. El código compilado es un archivo derivado y no debe registrarse en el control de código fuente. Hay tres razones para esto, que se describen a continuación:

- El archivo derivado puede ser generado por cualquier miembro del equipo una vez que tengamos la código fuente.
- En muchos casos, el código compilado es mucho más grande que el código fuente, y agregarlo al repositorio lo hará lento y lento. Además, recuerde que si hay 16 miembros en el equipo, todos ellos obtienen innecesariamente una copia de ese archivo generado, lo que ralentizará innecesariamente todo el sistema.
- Los sistemas de control de fuente están diseñados para almacenar el delta o los cambios que ha realizado en los archivos fuente desde su última confirmación. Los archivos distintos de los archivos de código fuente suelen ser archivos binarios. Lo más probable es que el sistema de control de fuente no pueda tener una diferencia herramienta para eso, y necesitará almacenar el archivo completo cada vez que se confirme. Tendrá un efecto negativo en el rendimiento del marco de control de código fuente.

24 Ciclo de vida óptimo de desarrollo de Python

En segundo lugar, cualquier cosa que sea confidencial no pertenece al control de código fuente. Esto incluye claves API y contraseñas.

Para el repositorio fuente, GitHub es la opción preferida de la comunidad de Python. Gran parte del control de código fuente de los famosos paquetes de Python también reside en GitHub. Si el código de Python se va a utilizar en todos los equipos, es necesario desarrollar y mantener el protocolo y los procedimientos correctos.

Comprender las estrategias para implementar el código

Para proyectos en los que el equipo de desarrollo no es el usuario final, es importante idear una estrategia para implementar el código para el usuario final. Para proyectos de escala relativamente grande, cuando hay un entorno DEV y PROD bien definido , la implementación del código y la elaboración de estrategias se vuelven importantes.

Python también es el lenguaje de elección para los entornos informáticos en la nube y en clúster.

Los problemas relacionados con la implementación del código se enumeran a continuación:

- Deben ocurrir exactamente las mismas transformaciones en DEV, TEST y PROD entornos.
- A medida que el código se sigue actualizando en el entorno DEV , ¿cómo se sincronizarán los cambios con el entorno PROD ?
- ¿Qué tipo de prueba tiene previsto realizar en los entornos DEV y PROD ?

Veamos dos estrategias principales para implementar el código.

Desarrollo por lotes

Este es el proceso de desarrollo tradicional. Desarrollamos el código, lo compilamos y luego lo probamos. Este proceso se repite iterativamente hasta que se cumplan todos los requisitos. Luego, se implementa el código desarrollado.

Emplear integración continua y entrega continua La integración continua/entrega

continua (CI/CD) en el contexto de Python se refiere a la integración e implementación continuas en lugar de llevarlas a cabo como un proceso por lotes.

Ayuda a crear un entorno de **operaciones de desarrollo (DevOps)** cerrando la brecha entre el desarrollo y las operaciones.

CI se refiere a la integración, creación y prueba continuas de varios módulos del código a medida que se actualizan. Para un equipo, esto significa que el código desarrollado individualmente por cada miembro del equipo se integra, crea y prueba, normalmente muchas veces al día. Una vez que se prueban, se actualiza el repositorio en el control de código fuente.

Una ventaja de CI es que los problemas o errores se solucionan desde el principio. Un error típico corregido el día en que se creó tarda mucho menos tiempo en resolverse de inmediato en lugar de resolverlo días, semanas o meses después, cuando ya se ha filtrado a otros módulos y los afectados pueden haber creado dependencias de varios niveles.

A diferencia de Java o C++, Python es un lenguaje interpretado, lo que significa que el código creado se puede ejecutar en cualquier máquina de destino con un intérprete. En comparación, el código compilado generalmente se crea para un tipo de máquina de destino y puede ser desarrollado por diferentes miembros del equipo. Una vez que sabemos qué pasos hay que seguir cada vez que se realiza un cambio, podemos automatizarlo.

Como el código de Python depende de paquetes externos, hacer un seguimiento de sus nombres y versiones es parte de la automatización del proceso de compilación. Una buena práctica es enumerar todos estos paquetes en un archivo llamado requisitos.txt. El nombre puede ser cualquier cosa, pero la comunidad de Python normalmente tiende a llamarlo requisitos.txt.

Para instalar los paquetes ejecutaremos el siguiente comando:

```
$pip install -r requisitos.txt
```

Para crear un archivo de requisitos que represente los paquetes utilizados en nuestro código, podemos usar el siguiente comando:

```
$pip congelar > requisitos.txt
```

El objetivo de la integración es detectar errores y defectos temprano, pero tiene el potencial de hacer que el proceso de desarrollo sea inestable. Habrá ocasiones en las que un miembro del equipo haya introducido un error importante, rompiendo así el código, si es posible que otros miembros del equipo tengan que esperar hasta que se resuelva ese error. La autoevaluación sólida por parte de los miembros del equipo y la elección de la frecuencia adecuada para la integración ayudarán a resolver el problema. Para pruebas sólidas, se debe implementar la ejecución de pruebas cada vez que se realiza un cambio. Este proceso de prueba debería eventualmente ser completamente automatizado. En caso de errores, la compilación debe fallar y se debe notificar al miembro del equipo responsable del módulo defectuoso. El miembro del equipo puede optar por proporcionar primero una solución rápida antes de tomarse el tiempo para resolver y probar completamente el problema para asegurarse de que otros miembros del equipo no estén bloqueados.

26 Ciclo de vida óptimo de desarrollo de Python

Una vez que el código está construido y probado, podemos optar por actualizar también el código implementado. Eso implementará la parte del **CD**. Si elegimos tener un proceso de CI/CD completo, significa que cada vez que se realiza un cambio, se construye y prueba y los cambios se reflejan en el código implementado. Si se gestiona correctamente, el usuario final se beneficiará de tener una solución en constante evolución. En algunos casos de uso, cada ciclo de CI/CD puede ser un movimiento iterativo de MVP a

una solución completa. En otros casos de uso, estamos tratando de capturar y formular un problema del mundo real que cambia rápidamente, descartando suposiciones obsoletas e incorporando nueva información. Un ejemplo es el análisis de patrones de la situación de COVID-19, que está cambiando hora a hora. Además, la nueva información está llegando a un ritmo rápido, y cualquier caso de uso relacionado con ella puede beneficiarse de CI/CD, mediante el cual los desarrolladores actualizan constantemente sus soluciones en función de nuevos hechos e información emergentes.

A continuación, analizaremos los entornos de desarrollo comúnmente utilizados para Python.

Entornos de desarrollo Python

Los editores de texto son una opción tentadora para editar código Python. Pero para cualquier proyecto de tamaño mediano a grande, tenemos que considerar seriamente **los entornos de desarrollo integrado (IDE) de Python**, que son muy útiles para escribir, depurar y solucionar problemas del código mediante el control de versiones y facilitan las implementaciones. Hay muchos IDE disponibles, en su mayoría gratuitos, en el mercado. En esta sección, revisaremos algunos de ellos. Tenga en cuenta que no intentaremos clasificarlos en ningún orden, sino que enfatizaremos el valor que aporta cada uno de ellos, y depende del lector tomar la mejor decisión en función de su experiencia pasada, los requisitos del proyecto y la complejidad de sus proyectos.

INACTIVO

El entorno de aprendizaje y desarrollo integrado (IDLE) es un editor predeterminado que viene con Python y está disponible para todas las plataformas principales (Windows, macOS y Linux). Es gratis y es un IDE decente para principiantes con fines de aprendizaje. No se recomienda para programación avanzada.

Texto sublime

Sublime Text es otro editor de código popular y se puede usar para varios idiomas. Es gratis solo con fines de evaluación. También está disponible para todas las plataformas principales (Windows, macOS y Linux). Viene con soporte básico de Python pero con su poderoso marco de extensiones, podemos personalizarlo para crear un entorno de desarrollo completo que necesita habilidades y tiempo adicionales. La integración con un sistema de control de versiones como Git o **Subversion (SVN)** es posible con complementos, pero es posible que no exponga las funciones completas de control de versiones.

Atom es otro editor popular que también está en la misma categoría que Sublime Text. Es gratis.

PyCharm

PyCharm es uno de los mejores editores IDE de Python disponibles para la programación de Python y está disponible para Windows, macOS y Linux. Es un IDE completo diseñado para la programación de Python, que ayuda a los programadores con la finalización del código, la depuración, la refactorización, la búsqueda inteligente, el acceso a servidores de bases de datos populares, la integración con sistemas de control de versiones y muchas más funciones. El IDE proporciona una plataforma de complementos para que los desarrolladores amplíen las funcionalidades básicas según sea necesario. PyCharm está disponible en los siguientes formatos:

- Versión comunitaria, que es gratuita y viene para el desarrollo puro de Python
- Versión profesional, que no es gratuita y viene con soporte para desarrollo web como **lenguaje de marcado de hipertexto (HTML)**, JavaScript y SQL

código de estudio visual

Visual Studio Code (VS Code) es un entorno de código abierto desarrollado por Microsoft.

Para Windows, VS Code es el mejor IDE de Python. No viene con un entorno de desarrollo de Python por defecto. Las extensiones de Python para VS Code pueden convertirlo en un entorno de desarrollo de Python.

Es liviano y está lleno de características poderosas. Es gratis y también está disponible para macOS y Linux. Viene con potentes funciones como finalización de código, depuración, refactorización, búsqueda, acceso a servidores de bases de datos, integración de sistemas de control de versiones y mucho más.

PyDev

Si está utilizando o ha utilizado Eclipse, puede considerar PyDev, que es un editor de terceros para Eclipse. Está en la categoría de uno de los mejores IDE de Python y también se puede usar para Jython y IronPython. Es gratis. Como PyDev es solo un complemento sobre Eclipse, está disponible para todas las plataformas principales, como Eclipse. Este IDE viene con todas las campanas y silbatos de Eclipse y, además, agiliza la integración con Django, las pruebas unitarias y **Google App Engine (GAE)**.

espía

Si planea usar Python para ciencia de datos y ML, puede considerar **Spyder** como su IDE. Spyder está escrito en Python. Este IDE ofrece herramientas para edición completa, depuración, ejecución interactiva, inspección profunda y capacidades de visualización avanzadas.

Además, admite la integración con Matplotlib, SciPy, NumPy, Pandas, Cython, IPython y SymPy para convertirlo en un IDE predeterminado para los científicos de datos.

Según la revisión de diferentes IDE en esta sección, podemos recomendar PyCharm y PyDev para desarrolladores de aplicaciones profesionales. Pero si te gusta más la ciencia de datos y el ML, seguramente vale la pena explorar Spyder.

Resumen

En este capítulo, sentamos las bases para los conceptos avanzados de Python discutidos en los capítulos posteriores de este libro. Comenzamos presentando el sabor, la orientación y el ambiente de un proyecto de Python. Comenzamos la discusión técnica identificando primero las diferentes fases del proyecto de Python y luego explorando diferentes formas de optimizarlo en función de los casos de uso en los que estamos trabajando. Para un lenguaje conciso como Python, la documentación de buena calidad contribuye en gran medida a que el código sea legible y explícito.

También analizamos varias formas de documentar el código de Python. A continuación, investigamos las formas recomendadas de crear documentación en Python. También estudiamos los esquemas de nombres que pueden ayudarnos a hacer que el código sea más legible. A continuación, analizamos las diferentes formas en que podemos usar el control de código fuente. También descubrimos cuáles son las diferentes formas de implementar el código de Python. Finalmente, revisamos algunos entornos de desarrollo para Python para ayudarlo a elegir un entorno de desarrollo en función de los antecedentes que tienen y el tipo de proyecto en el que va a trabajar.

Los temas que cubrimos en este capítulo son beneficiosos para cualquiera que esté comenzando un nuevo proyecto que involucre a Python. Estas discusiones ayudan a tomar la estrategia y la decisión de diseño de un nuevo proyecto de manera rápida y eficiente. En el próximo capítulo, investigaremos cómo podemos modularizar el código de un proyecto de Python.

Preguntas

1. ¿Qué es El Zen de Python?
2. En Python, ¿qué tipo de documentación está disponible en tiempo de ejecución?
3. ¿Qué es un ciclo de vida CRISP-DM?

Otras lecturas

- Modern Python Cookbook – Segunda edición, por Steven F. Lott
- Blueprints de programación Python, por Daniel Furtado
- Recetas secretas de Python Ninja, por Cody Jackson

respuestas

1. Una colección de 19 pautas escritas por Tim Peters que se aplican al diseño de proyectos de Python.
2. A diferencia de los comentarios normales, las cadenas de documentación están disponibles en tiempo de ejecución para el compilador
3. **CRISP-DM** son las siglas de **Cross-Industry Standard Process for Data Mining**. Se aplica al ciclo de vida de un proyecto Python en el dominio ML e identifica diferentes fases de un proyecto.

2

Utilizando Modularización a Complejo de mango Proyectos

Cuando comienzas a programar en Python, es muy tentador poner todo el código de tu programa en un solo archivo. No hay problema en definir funciones y clases en el mismo archivo donde está tu programa principal. Esta opción es atractiva para los principiantes debido a la facilidad de ejecución del programa y para evitar la gestión de código en varios archivos. Pero un enfoque de programa de archivo único no es escalable para proyectos de tamaño mediano a grande. Se vuelve un desafío hacer un seguimiento de todas las diversas funciones y clases que define.

Para superar la situación, la programación modular es el camino a seguir para proyectos medianos y grandes. La modularidad es una herramienta clave para reducir la complejidad de un proyecto. La modularización también facilita la programación eficiente, la depuración y administración sencillas, la colaboración y la reutilización. En este capítulo, discutiremos cómo construir y consumir módulos y paquetes en Python.

32 Uso de la modularización para manejar proyectos complejos

Cubriremos los siguientes temas en este capítulo:

- Introducción a módulos y paquetes
- Importación de módulos
- Cargar e inicializar un módulo
- Escribir módulos reutilizables
- Paquetes de construcción
- Acceder a paquetes desde cualquier ubicación
- Compartir un paquete

Este capítulo lo ayudará a comprender los conceptos de módulos y paquetes en Python.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- Debe registrar una cuenta con Test PyPI y crear un token de API en su cuenta.

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter02>.

Introducción a módulos y paquetes.

Los módulos en Python son archivos de Python con una extensión .py . En realidad, son una forma de organizar funciones, clases y variables utilizando uno o más archivos de Python de modo que sean fáciles de administrar, reutilizar en los diferentes módulos y extenderse a medida que los programas se vuelven complejos.

Un paquete de Python es el siguiente nivel de programación modular. Un paquete es como una carpeta para organizar múltiples módulos o subpaquetes, lo cual es fundamental para compartir los módulos para su reutilización.

Los archivos fuente de Python que usan solo las bibliotecas estándar son fáciles de compartir y distribuir mediante correo electrónico, GitHub y unidades compartidas, con la única salvedad de que debe haber compatibilidad con la versión de Python. Pero este enfoque de uso compartido no escalará para proyectos que tienen una cantidad decente de archivos y dependen de bibliotecas de terceros y pueden desarrollarse para una versión específica de Python. Para salvar la situación, construir y compartir paquetes es imprescindible para compartir y reutilizar de manera eficiente los programas de Python.

A continuación, analizaremos cómo importar módulos y los diferentes tipos de técnicas de importación compatibles con Python.

Importación de módulos

El código de Python en un módulo puede obtener acceso al código de Python en otro módulo mediante un proceso llamado importar módulos.

Para profundizar en los diferentes conceptos de módulos y paquetes, crearemos dos módulos y un script principal que usará esos dos módulos. Estos dos módulos se actualizarán o reutilizarán a lo largo de este capítulo.

Para crear un nuevo módulo, crearemos un archivo .py con el nombre del módulo. Crearemos un archivo mycalculator.py con dos funciones: sumar y restar. El complemento calcula la suma de los dos números proporcionados a la función como argumentos y devuelve el valor calculado. La función de resta calcula la diferencia entre los dos números proporcionados a la función como argumentos y devuelve el valor calculado.

A continuación se muestra un fragmento de código de mycalculator.py :

```
# mycalculator.py con funciones de suma y resta
def suma(x, y):
    """Esta función suma dos números"""
    volver x + y

def restar(x, y):
    """Esta función resta dos números"""
    volver x - y
```

Tenga en cuenta que el nombre del módulo es el nombre del archivo.

34 Uso de la modularización para manejar proyectos complejos

Crearemos un segundo módulo agregando un nuevo archivo con el nombre myrandom.py. Este módulo tiene dos funciones: random_1d y random_2d. La función random_1d es para generar un número aleatorio entre 1 y 9 y la función random_2d es para generar un número aleatorio entre 10 y 99. Tenga en cuenta que este módulo también usa la biblioteca aleatoria , que es un módulo integrado de Python.

El fragmento de código de myrandom.py se muestra a continuación:

```
# myrandom.py con funciones aleatorias predeterminadas y personalizadas
import random

def random_1d():
    """Esta función genera un número aleatorio entre 0 \
    y 9"""
    volver aleatorio.randint (0,9)

def random_2d():
    """Esta función genera un número aleatorio entre 10 \
    y 99"""
    volver aleatorio.randint (10,99)
```

Para consumir estos dos módulos, también creamos la secuencia de comandos principal de Python (calcmain1. py), que importa los dos módulos y los usa para lograr estas dos funciones de calculadora. La declaración de importación es la forma más común de importar módulos integrados o personalizados.

A continuación se muestra un fragmento de código de calcmain1.py :

```
# calcmain1.py con una función principal
importar mycalculator
importar myrandom

def mi_principal( ):
    """ Esta es una función principal que genera dos números aleatorios y luego les aplica funciones
    de calculadora """
    x = myrandom.random_2d( ) y =
    myrandom.random_1d( ) sum =
    micalculadora.sumar(x, y) diff = micalculadora.restar(x,
    y)
```

```
imprimir("x = {}, y = {}".format(x, y))
imprimir("la suma es {}".format(suma))
imprimir("la diferencia es {}".format(dif))

"""
    Esto se ejecuta solo si la variable especial '__name__' se establece como
    principal"""

si __nombre__ == "__principal__":
    mi_principal()
```

En este script principal (otro módulo), importamos los dos módulos usando la importación declaración. Definimos la función principal (my_main), que se ejecutará solo si este script o el módulo calcmain1 se ejecuta como programa principal. Los detalles de la ejecución de la función principal desde el programa principal se tratarán más adelante en la sección Configuración de variables especiales. En la función my_main , generamos dos números aleatorios usando el módulo myrandom y luego calculamos la suma y la diferencia de los dos números aleatorios usando el módulo mycalculator . Al final, enviamos los resultados a la consola usando la declaración de impresión .

Nota IMPORTANTE

Un módulo se carga solo una vez. Si un módulo es importado por otro módulo o por el script principal de Python, el módulo se inicializará ejecutando el código en el módulo. Si otro módulo en su programa importa el mismo módulo nuevamente, no se cargará dos veces sino solo una vez. Esto significa que si hay variables locales dentro del módulo, actuarán como Singleton (inicializado solo una vez).

Hay otras opciones disponibles para importar un módulo, como `importlib.import_module()` y la función `__import__()` incorporada. Analicemos cómo funciona la importación y otras opciones alternativas.

Usando la declaración de importación

Como ya se mencionó, la declaración de importación es una forma común de importar un módulo. El siguiente fragmento de código es un ejemplo del uso de una declaración de importación :

```
import matematicas
```

36 Uso de la modularización para manejar proyectos complejos

La declaración de importación es responsable de dos operaciones: primero, busca el módulo dado después de la palabra clave de importación y luego vincula los resultados de esa búsqueda a un nombre de variable (que es el mismo que el nombre del módulo) en el ámbito local de La ejecución. En las próximas dos secciones, discutiremos cómo funciona la importación y también cómo importar elementos específicos de un módulo o paquete.

Aprender cómo funciona la importación

A continuación, debemos comprender cómo funciona la declaración de importación . Primero, debemos recordar que todas las variables y funciones globales se agregan al espacio de nombres global mediante el intérprete de Python al comienzo de una ejecución. Para ilustrar el concepto, podemos escribir un pequeño programa de Python para escupir el contenido del espacio de nombres global , como se muestra a continuación:

```
# globalmain.py con función globals()
def imprimir_globales():
    imprimir (globales())

definitivamente hola():
    imprimir ("Hola")

si __nombre__ == "__principal__":
    imprimir_globales()
```

Este programa tiene dos funciones: print_globals y hola. Los print_globals La función escupirá el contenido del espacio de nombres global. La función hello no se ejecutará y se agrega aquí para mostrar su referencia en la salida de la consola del espacio de nombres global. La salida de la consola después de ejecutar este código de Python será similar a la siguiente:

```
{
    "__nombre__":="__principal__",
    "__doc__":"Ninguno",
    "__paquete__":"Ninguno",
    "__loader__":"><_frozen_importlib_external.\n    Objeto SourceFileLoader en 0x101670208>",
    "__spec__":"Ninguno",
    "__anotaciones__":{},
},
    "__incorporados__":<módulo 'incorporados' (integrado)>>,
```

```
"__file__": "/PythonForGeeks/source_code/chapter2/\módulos/globalmain.py",
 "__cached__": "Ninguno", "print_globals": "<función print_globals en \
",
 "",
 "",
 0x1016c4378>",
 "hola": "<función hola en 0x1016c4400>"
}
```

Los puntos clave a tener en cuenta en esta salida de la consola son los siguientes:

- La variable `__name__` se establece en el valor `__main__`. Esto se discutirá con más detalle en la sección Cargar e inicializar un módulo.
- La variable `__file__` se establece en la ruta del archivo del módulo principal aquí.
- Se añade una referencia a cada función al final.

Si agregamos `print(globals())` a nuestro script `calcmain1.py`, la salida de la consola después de agregar esta declaración será similar a la siguiente:

```
{ "__name__": "__main__",
 "__doc__": "Ninguno",
 "__package__": "Ninguno",
 "__loader__": "<_frozen_importlib_external.\Objeto SourceFileLoader en
0x100de1208>",
 "__spec__": "Ninguno",
 "__anotaciones__": {},
 "__incorporados__": "<módulo 'incorporados' (integrado)>",
 "__archivo__": "/PythonForGeeks/código_fuente/capítulo2/módulo1/
principal.py",
 "__cached__": "Ninguno",
 "mycalculator": "<módulo 'mycalculator' from \ '/PythonForGeeks/source_code/
chapter2/modules/\ mycalculator.py'>", "myrandom": "<módulo 'myrandom' from ' /
PythonForGeeks/source_code/chapter2/modules/myrandom.py'>",
 "",
 "",
 "mi_principal": "<función mi_principal en 0x100e351e0>" }
```

38 Uso de la modularización para manejar proyectos complejos

Un punto importante a tener en cuenta es que hay dos variables adicionales (mycalculator y myrandom) agregado al espacio de nombres global correspondiente a cada importación instrucción utilizada para importar estos módulos. Cada vez que importamos una biblioteca, se crea una variable con el mismo nombre, que contiene una referencia al módulo como una variable para las funciones globales (my_main en este caso).

Veremos, en otros enfoques de importación de módulos, que podemos definir explícitamente algunas de estas variables para cada módulo. La declaración de importación hace esto automáticamente por nosotros.

Importación específica

También podemos importar algo específico (variable, función o clase) de un módulo en lugar de importar todo el módulo. Esto se logra utilizando la instrucción `from`, como la siguiente:

```
de importación matemática pi
```

Otra práctica recomendada es usar un nombre diferente para un módulo importado por conveniencia o, a veces, cuando se usan los mismos nombres para diferentes recursos en dos bibliotecas diferentes. Para ilustrar esta idea, actualizaremos nuestro archivo calcmain1.py (el programa actualizado es calcmain2.py) del ejemplo anterior usando calc y rand alias para los módulos mycalculator y myrandom , respectivamente. Este cambio hará que el uso de los módulos en el script principal sea mucho más simple, como se muestra a continuación:

```
# calcmain2.py con alias para módulos
importar mi calculadora como calc
importar myrandom como rand

def mi_principal():
    """ Esta es una función principal que genera dos aleatorios\
números y luego aplicarles funciones de calculadora """
    x = rand.random_2d()
    y = aleatorio.aleatorio_1d()

    suma = calc.add(x,y)
    diff = calc.restar(x,y)

    imprimir("x = {}, y = {}".format(x,y))
    imprimir("la suma es {}".format(suma))
    imprimir("la diferencia es {}".format(dif))
```

```
""" Esto se ejecuta solo si la variable especial '__name__' se establece como principal"""

si __nombre__ == "__principal__":
    mi_principal()
```

Como próximo paso, combinaremos los dos conceptos discutidos anteriormente en la siguiente iteración del programa calcmain1.py (el programa actualizado es calcmain3.py). En esta actualización, usaremos la instrucción from con los nombres de los módulos y luego importaremos las funciones individuales de cada módulo. En el caso de las funciones de suma y resta , usamos la declaración as para definir una definición local diferente del recurso del módulo con fines ilustrativos.

Un fragmento de código de calcmain3.py es el siguiente:

```
# calcmain3.py con from y alias combinados

desde mycalculator importar agregar como my_add
de mycalculator importar restar como my_subtract
de myrandom importar random_2d, random_1d

def mi_principal():

    """ Esta es una función principal que genera dos aleatorios
    números y luego aplicarles funciones de calculadora """

    x = aleatorio_2d()
    y = aleatorio_1d()

    suma = mi_suma(x,y)
    diff = mi_resta(x,y)

    imprimir("x = {}, y = {}".format(x,y))
    imprimir("la suma es {}".format(suma))
    imprimir("la diferencia es {}".format(dif))

    imprimir (globales())

""" Esto se ejecuta solo si la variable especial '__name__' se establece como principal"""

si __nombre__ == "__principal__":
    mi_principal()
```

40 Uso de la modularización para manejar proyectos complejos

Como usamos la instrucción print (globals ()) con este programa, la salida de la consola de este programa mostrará que las variables correspondientes a cada función se crean según nuestro alias. La salida de la consola de muestra es la siguiente:

```
{  
    "__nombre__": "__principal__",  
    "__doc__": "Ninguno",  
    "__paquete__": "Ninguno",  
    "__loader__": "<_frozen_importlib_external.\n    Objeto SourceFileLoader en 0x1095f1208>",  
    "__spec__": "Ninguno",  
    "__anotaciones__": {},  
    "__incorporados__": "<módulo 'incorporados' (integrado)>", "__  
        archivo__": "./PythonForGeeks/código_fuente/capítulo2/módulo1/  
            principal_2.py",  
    "__cached__": "Ninguno",  
    "my_add": "<función agregar en 0x109645400>",  
    "my_subtract": "<función restar en 0x109645598>",  
    "random_2d": "<función random_2d en 0x10967a840>",  
    "random_1d": "<función random_1d en 0x1096456a8>",  
    "my_main": "<función my_main en 0x109645378>"  
}
```

Tenga en cuenta que las variables en negrita corresponden a los cambios que hicimos en la importación declaraciones en el archivo calcmain3.py .

Usando la instrucción __import__

La declaración __import__ es una función de bajo nivel en Python que toma una cadena como entrada y activa la operación de importación real. Las funciones de bajo nivel son parte del lenguaje central de Python y, por lo general, están destinadas a usarse para el desarrollo de bibliotecas o para acceder a los recursos del sistema operativo, y no se usan comúnmente para el desarrollo de aplicaciones.

Podemos usar esta palabra clave para importar la biblioteca aleatoria en nuestro módulo myrandom.py de la siguiente manera:

```
#importar al azar  
aleatorio = __import__('aleatorio')
```

El resto del código en myrandom.py se puede usar tal como está sin ningún cambio.

Ilustramos un caso simple del uso del método `__import__` por razones académicas y omitiremos los detalles avanzados para aquellos de ustedes que estén interesados en explorar como lectura adicional. La razón de esto es que no se recomienda el uso del método `__import__` para aplicaciones de usuario; está diseñado más para intérpretes.

La declaración `importlib.import_module` es la que se debe usar además de la importación normal para la funcionalidad avanzada.

Usando la instrucción `importlib.import_module`

Podemos importar cualquier módulo usando la biblioteca `importlib`. La biblioteca `importlib` ofrece una variedad de funciones, incluida `__import__`, relacionada con la importación de módulos de una manera más flexible. Aquí hay un ejemplo simple de cómo importar un módulo aleatorio en nuestro módulo `myrandom.py` usando `importlib`:

```
importar importlib  
aleatorio = importlib.import_module('aleatorio')
```

El resto del código en `myrandom.py` se puede usar tal como está sin ningún cambio.

El módulo `importlib` es mejor conocido por importar módulos dinámicamente y es muy útil en los casos en los que el nombre del módulo no se conoce de antemano y necesitamos importar los módulos en tiempo de ejecución. Este es un requisito común para el desarrollo de complementos y extensiones.

Las funciones de uso común disponibles en el módulo `importlib` son las siguientes:

- `__import__`: Esta es la implementación de la función `__import__`, como ya discutido
- `import_module`: Esto se usa para importar un módulo y se usa más comúnmente para cargar un módulo dinámicamente. En este método, puede especificar si desea importar un módulo utilizando una ruta absoluta o relativa. La función `import_module` es un envoltorio alrededor de `importlib.__import__`. Tenga en cuenta que la primera función devuelve el paquete o módulo (por ejemplo, `paqueteA.módulo1`), que se especifica con la función, mientras que la última función siempre devuelve el paquete o módulo de nivel superior (por ejemplo, `paqueteA`).
- `importlib.util.find_spec`: Este es un método reemplazado para `find_loader`, que está en desuso desde la versión 3.4 de Python. Este método se puede utilizar para validar si el módulo existe y es válido.

42 Uso de la modularización para manejar proyectos complejos

- `invalidate_caches`: este método se puede utilizar para invalidar los cachés internos de los buscadores almacenados en `sys.meta_path`. El caché interno es útil para cargar el módulo más rápido sin activar los métodos de búsqueda nuevamente. Pero si estamos importando dinámicamente un módulo, especialmente si se crea después de que el intérprete comenzó a ejecutarse, es una buena práctica llamar al método `invalidate_caches`. Esta función borrará todos los módulos o bibliotecas del caché para asegurarse de que el sistema de importación cargue el módulo solicitado desde la ruta del sistema.
- `recargar`: como sugiere el nombre, esta función se utiliza para recargar una módulo importado. Necesitamos proporcionar el objeto del módulo como un parámetro de entrada para esta función. Esto significa que la función de importación debe realizarse correctamente. Esta función es muy útil cuando se espera editar o cambiar el código fuente del módulo y desea cargar la nueva versión sin reiniciar el programa.

Importación absoluta versus relativa

Tenemos una idea bastante buena de cómo usar declaraciones de importación . Ahora es el momento de comprender las importaciones **absolutas** y **relativas** , especialmente cuando importamos módulos personalizados o específicos del proyecto. Para ilustrar los dos conceptos, tomemos un ejemplo de un proyecto con diferentes paquetes, subpaquetes y módulos, como se muestra a continuación:

```
proyecto
    ÿÿÿ paquete1
        ÿ yyÿ módulo1.py
        ÿ yyÿ module2.py (contiene una función llamada func1 ())
    ÿÿÿ paquete2
        ÿ yyÿ __init__.py
        ÿ yyÿ módulo3.py
    ÿÿÿ sub_pkg1
        ÿ yyÿ module6.py (contiene una función llamada func2 ())
    ÿÿÿ paquete3
        ÿ yyÿ módulo4.py
        ÿ yyÿ módulo5.py
    ÿÿÿ sub_pkg2
        ÿ yyÿ módulo7.py
```

Usando esta estructura de proyecto, discutiremos cómo usar importaciones absolutas y relativas.

Importación absoluta

Podemos usar rutas absolutas comenzando desde el paquete de nivel superior y profundizando hasta el nivel de módulo y subpaquete. Aquí se muestran algunos ejemplos de importación de diferentes módulos:

```
desde pkg1 import module1  
de pkg1.module2 import func1  
  
desde pkg2 import module3  
de pkg2.sub_pkg1.module6 import func2  
  
de pkg3 import module4, module5  
desde pkg3.sub_pkg2 import module7
```

Para declaraciones de importación absoluta, debemos dar una ruta detallada para cada paquete o archivo, desde la carpeta del paquete de nivel superior, que es similar a la ruta de un archivo.

Se recomiendan las importaciones absolutas porque son fáciles de leer y seguir la ubicación exacta de los recursos importados. Las importaciones absolutas se ven menos afectadas por el uso compartido de proyectos y los cambios en la ubicación actual de las declaraciones de importación . De hecho, PEP 8 recomienda explícitamente el uso de importaciones absolutas.

A veces, sin embargo, las importaciones absolutas son sentencias bastante largas dependiendo del tamaño de la estructura de carpetas del proyecto, que no es conveniente mantener.

Importación relativa

Una importación relativa especifica el recurso que se importará en relación con la ubicación actual, que es principalmente la ubicación actual del archivo de código de Python donde se usa la declaración de importación .

Para los ejemplos de proyectos discutidos anteriormente, aquí hay algunos escenarios de importación relativa. Las declaraciones de importación relativa equivalentes son las siguientes:

- **Escenario 1:** Importando funct1 dentro de module1.py:

```
de .module2 import func1
```

Usamos un punto (.) solo porque module2.py está en la misma carpeta que module1.py.

- **Escenario 2:** Importando module4 dentro de module1.py:

```
de ..pkg3 import module4
```

44 Uso de la modularización para manejar proyectos complejos

En este caso, usamos dos puntos (..) porque module4.py está en la carpeta hermana de module1.py.

- **Escenario 3:** Importando Func2 dentro de module1.py:

```
de ..pkg2.sub_pkg_1.module2 importar Func2
```

Para este escenario, usamos dos puntos (..) porque el módulo de destino (module2.py) está dentro de una carpeta que está en la carpeta hermana de module1.py. Usamos un punto para acceder al paquete sub_pkg_1 y otro punto para acceder al módulo2.

Una ventaja de las importaciones relativas es que son simples y pueden reducir significativamente las largas declaraciones de importación . Pero las declaraciones de importación relativa pueden ser complicadas y difíciles de mantener cuando los proyectos se comparten entre equipos y organizaciones. Las importaciones relativas no son fáciles de leer y administrar.

Cargando e inicializando un módulo

Cada vez que el intérprete de Python interactúa con una declaración de importación o equivalente, realiza tres operaciones, que se describen en las siguientes secciones.

Cargando un módulo

El intérprete de Python busca el módulo especificado en un sys.path (que se analizará en la sección Acceso a paquetes desde cualquier ubicación) y carga el código fuente. Esto se ha explicado en la sección Aprender cómo funciona la importación.

Configuración de variables especiales

En este paso, el intérprete de Python define algunas variables especiales, como __name__, que básicamente define el espacio de nombres en el que se ejecuta un módulo de Python. La variable name__ es una de las variables más importantes.

En el caso de nuestro ejemplo de los módulos calcmain1.py , mycalculator.py y myrandom.py , la variable __name__ se establecerá para cada módulo de la siguiente manera:

Module Name	__name__ =
main.py	__main__
myrandom.py	myrandom
mycalculator.py	mycalculator

Tabla 2.1 – El valor del atributo __name__ para diferentes módulos

Hay dos casos de configuración de la variable `__name__`, que se describen a continuación.

Caso A – módulo como programa principal

Si está ejecutando su módulo como el programa principal, la variable `__name__` se establecerá en el valor `__main__` independientemente del nombre del archivo o módulo de Python. Por ejemplo, cuando se ejecuta `calcmain1.py`, el intérprete asignará la cadena `__main__` codificada de forma rígida a la variable `__name__`. Si ejecutamos `myrandom.py` o `mycalculator.py` como programa principal, la variable `__name__` obtendrá automáticamente el valor de `__main__`.

Por lo tanto, agregamos una línea `if __name__ == '__main__'` a todos los scripts principales para verificar si este es el programa de ejecución principal.

Caso B: el módulo es importado por otro módulo

En este caso, su módulo no es el programa principal, sino que lo importa otro módulo. En nuestro ejemplo, `myrandom` y `mycalculator` se importan en `calcmain1.py`. Tan pronto como el intérprete de Python encuentre los archivos `myrandom.py` y `mycalculator.py`, asignará los nombres `myrandom` y `mycalculator` de la declaración de importación a la variable `__name__` para cada módulo. Esta asignación se realiza antes de ejecutar el código dentro de estos módulos. Esto se refleja en la Tabla 2.1.

Algunas de las otras variables especiales notables son las siguientes:

- `__file__`: esta variable contiene la ruta al módulo que se está ejecutando actualmente .importado.
- `__doc__`: esta variable generará la cadena de documentación que se agrega en una clase o un método. Como se discutió en el Capítulo 1, Ciclo de vida óptimo de desarrollo de Python, una cadena de documentación es una línea de comentario que se agrega justo después de la definición de clase o método.
- `__paquete__`: Se utiliza para indicar si el módulo es un paquete o no. Su valor puede ser un nombre de paquete, una cadena vacía o ninguno.
- `__dict__`: Esto devolverá todos los atributos de una instancia de clase como un diccionario.
- `dir`: este es en realidad un método que devuelve cada método o atributo asociado como una lista.
- Locales y globales: también se utilizan como métodos que muestran las variables locales y globales como entradas de diccionario.

Ejecutando el código

Después de establecer las variables especiales, el intérprete de Python ejecuta el código en el archivo línea por línea. Es importante saber que las funciones (y el código debajo de las clases) no se ejecutan a menos que otras líneas de código las llamen. Aquí hay un análisis rápido de los tres módulos desde el punto de vista de la ejecución cuando se ejecuta calcmain1.py :

- mycalculator.py: después de configurar las variables especiales, no hay código para ejecutar en este módulo en el momento de la inicialización.
- myrandom.py: después de configurar las variables especiales y la declaración de importación , hay No hay más código para ejecutar en este módulo en el momento de la inicialización.
- calcmain1.py: después de configurar las variables especiales y ejecutar la importación sentencias, ejecuta la siguiente sentencia if : if __name__ == "__main__":. Esto devolverá verdadero porque lanzamos el archivo calcmain1.py . Dentro de la declaración if , se llamará a la función my_main () , que de hecho llama a los métodos de los módulos myrandom.py y mycalculator.py .

Podemos agregar una declaración if __name__ == "__main__" a cualquier módulo, independientemente de si es el programa principal o no. La ventaja de usar este enfoque es que el módulo se puede usar como módulo o como programa principal. También hay otra aplicación de usar este enfoque, que es agregar pruebas unitarias dentro del módulo.

Módulos estándar

Python viene con una biblioteca de más de 200 módulos estándar. El número exacto varía de una distribución a otra. Estos módulos se pueden importar a su programa. La lista de estos módulos es muy extensa, pero aquí solo se mencionan algunos módulos de uso común como ejemplo de módulos estándar:

- matemáticas: este módulo proporciona funciones matemáticas para operaciones aritméticas.
- aleatorio: este módulo es útil para generar números pseudoaleatorios usando diferentes tipos de distribuciones.
- estadísticas: este módulo ofrece funciones estadísticas como la media, la mediana y la varianza.
- base64: este módulo proporciona funciones para codificar y decodificar datos.
- calendario: este módulo ofrece funciones relacionadas con el calendario, lo cual es útil para los cálculos basados en el calendario.

- colecciones: este módulo contiene tipos de datos de contenedores especializados distintos de los contenedores integrados de uso general (como dict, list o set). Estos tipos de datos especializados incluyen deque, Counter y ChainMap.
- csv: este módulo ayuda a leer y escribir en archivos delimitados basados en comas.
- datetime: este módulo ofrece funciones de fecha y hora de propósito general.
- decimal: Este módulo es específico para operaciones aritméticas basadas en decimales.
- registro: este módulo se utiliza para facilitar el inicio de sesión en su aplicación.
- os y os.path: estos módulos se utilizan para acceder a funciones relacionadas con el sistema operativo.
- socket: este módulo proporciona funciones de bajo nivel para sockets red de comunicación.
- sys: este módulo proporciona acceso a un intérprete de Python para variables de bajo nivel y funciones
- tiempo: este módulo ofrece funciones relacionadas con el tiempo, como convertir a diferentes unidades de tiempo

Escribir módulos reutilizables

Para que un módulo sea declarado reutilizable, tiene que tener las siguientes características:

- Funcionalidad independiente
- Funcionalidad de uso general
- Estilo de codificación convencional
- Documentación bien definida

Si un módulo o paquete no tiene estas características, sería muy difícil, si no imposible, reutilizarlo en otros programas. Discutiremos cada característica una por una.

Funcionalidad independiente

Las funciones de un módulo deben ofrecer una funcionalidad independiente de otros módulos e independiente de cualquier variable local o global. Cuanto más independientes sean las funciones, más reutilizable será el módulo. Si tiene que usar otros módulos, tiene que ser mínimo.

48 Uso de la modularización para manejar proyectos complejos

En nuestro ejemplo de `mymath.py`, las dos funciones son completamente independientes y pueden ser reutilizadas por otros programas:

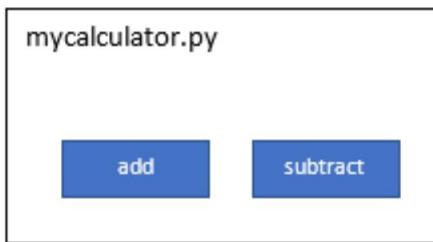


Figura 2.1 – El módulo mycalculator con funciones de suma y resta

En el caso de `myrandom.py`, estamos utilizando la biblioteca del sistema aleatorio para brindar la funcionalidad de generar números aleatorios. Este sigue siendo un módulo muy reutilizable porque la biblioteca aleatoria es uno de los módulos integrados en Python:

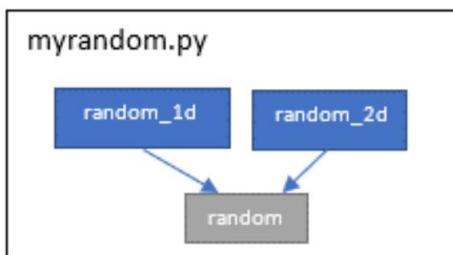


Figura 2.2 – El módulo myrandom con función dependiente de la biblioteca aleatoria

En los casos en los que tenemos que usar bibliotecas de terceros en nuestros módulos, podemos tener problemas al compartir nuestros módulos con otros si el entorno de destino no tiene las bibliotecas de terceros ya instaladas.

Para elaborar más este problema, presentaremos un nuevo módulo, `mypandas.py`, que aprovechará la funcionalidad básica de la famosa biblioteca pandas . Para simplificar, le agregamos solo una función, que es imprimir el DataFrame según el diccionario que se proporciona como una variable de entrada para la función.

El fragmento de código de `mypandas.py` es el siguiente:

```

#mispandas.py
import pandas

def print_dataframe(dict):
    """Esta función genera un diccionario como un marco de datos """
  
```

```
brics = pandas.DataFrame(dict)
imprimir (brics)
```

Nuestro módulo mypandas.py utilizará la biblioteca pandas para crear un marco de datos objeto del diccionario. Esta dependencia también se muestra en el siguiente diagrama de bloques:

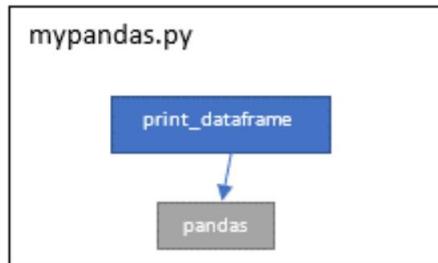


Figura 2.3: el módulo mypandas con dependencia de una biblioteca pandas de terceros

Tenga en cuenta que la biblioteca pandas no es una biblioteca integrada o del sistema. Cuando intentamos compartir este módulo con otros sin definir una dependencia clara de una biblioteca de terceros (pandas en este caso), el programa que intentará usar este módulo dará el siguiente mensaje de error:

```
ImportError: ningún módulo llamado pandas'
```

Por eso es importante que el módulo sea lo más independiente posible. Si tenemos que usar bibliotecas de terceros, debemos definir dependencias claras y usar un enfoque de empaquetado adecuado. Esto se discutirá en la sección Compartir un paquete.

Funcionalidad de generalización

Un módulo reutilizable ideal debería centrarse en resolver un problema general en lugar de un problema muy específico. Por ejemplo, tenemos el requisito de convertir pulgadas a centímetros. Podemos escribir fácilmente una función que convierta pulgadas en centímetros aplicando una fórmula de conversión. ¿Qué hay de escribir una función que convierta cualquier valor en el sistema imperial a un valor en el sistema métrico? Podemos tener una función para diferentes conversiones que pueden manejar pulgadas a centímetros, pies a metros o millas a kilómetros, o funciones separadas para cada tipo de estas conversiones. ¿Qué pasa con las funciones inversas (centímetros a pulgadas)? Es posible que esto no se requiera ahora, pero puede ser requerido más adelante o por alguien que esté reutilizando este módulo. Esta generalización hará que la funcionalidad del módulo no solo sea completa sino también más reutilizable sin extenderla.

50 Uso de la modularización para manejar proyectos complejos

Para ilustrar el concepto de generalización, revisaremos el diseño del `myrandom` módulo para hacerlo más general y, por lo tanto, más reutilizable. En el diseño actual, definimos funciones separadas para números de uno y dos dígitos. ¿Qué sucede si necesitamos generar un número aleatorio de tres dígitos o generar un número aleatorio entre 20 y 30? Para generalizar el requisito, presentamos una nueva función, `get_random`, en el mismo módulo, que toma la entrada del usuario para los límites inferior y superior de los números aleatorios. Esta función recién agregada es una generalización de las dos funciones aleatorias que ya definimos.

Con esta nueva función en el módulo, las dos funciones existentes pueden eliminarse o pueden permanecer en el módulo para mayor comodidad de uso. Tenga en cuenta que la biblioteca aleatoria también ofrece la función recién agregada lista para usar; la razón para proporcionar la función en nuestro módulo es simplemente para ilustrar la función generalizada (`get_random` en este caso) frente a las funciones específicas (`random_1d` y `random_2d` en este caso).

La versión actualizada del módulo `myrandom.py` (`myrandomv2.py`) es la siguiente:

```
# myrandomv2.py con funciones aleatorias predeterminadas y personalizadas
import random

def random_1d():
    """Esta función obtiene un número aleatorio entre 0 y 9"""
    return random.randint(0,9)

def random_2d():
    """Esta función obtiene un número aleatorio entre 10 y 99"""
    return random.randint(10,99)

def get_random(inferior, superior):
    """Esta función obtiene un número aleatorio entre menor y\ superior"""
    return random.randint(inferior, superior)
```

Estilo de codificación convencional

Esto se centra principalmente en cómo escribimos nombres de funciones, nombres de variables y nombres de módulos. Python tiene un sistema de codificación y convenciones de nomenclatura, que se discutieron en el capítulo anterior de este libro. Es importante seguir las convenciones de codificación y nomenclatura, especialmente al crear módulos y paquetes reutilizables. De lo contrario, discutiremos dichos módulos como malos ejemplos de módulos reutilizables.

Para ilustrar este punto, mostraremos el siguiente fragmento de código con los nombres de funciones y parámetros usando camel case:

```
def sumaNúmeros(numParam1, numParam2)
#se omite el código de función
Def featureCount(nombre del módulo)
#se omite el código de función
```

Si viene de un entorno de Java, este estilo de código parecerá correcto. Pero se considera una mala práctica en Python. El uso del estilo de codificación no Pythonic hace que la reutilización de dichos módulos sea muy difícil.

Aquí hay otro fragmento de un módulo con un estilo de codificación apropiado para nombres de funciones:

```
def agregar_números (num_param1, num_param2)
#se omite el código de función
Def feature_count(module_name)
#se omite el código de función
```

Otro ejemplo de un buen estilo de codificación reutilizable se ilustra en la siguiente captura de pantalla, que se toma del PyCharm IDE para la biblioteca pandas :

The screenshot shows a PyCharm IDE interface with a code editor and a completion dropdown menu. The code in the editor is:

```
brics = pandas.DataFrame(dict)
print(brics) testing (pandas)
```

The completion dropdown lists several methods from the pandas library:

- f test(extra_args) pandas.util._tester
- f tseries (pandas)
- f timedelta_range(start, end, period...) pandas.core.indexes.timedeltas
- f to_datetime(arg, errors, dayfirst, ye... pandas.core.tools.datetimes
- f to_numeric(arg, errors, downcast) pandas.core.tools.numeric
- f to_pickle(obj, filepath_or_buffer, compression, ...) pandas.io.pickle
- f to_timedelta(arg, unit, errors) pandas.core.tools.timedeltas
- v _tslib pandas
- f read_sql_table(table_name, con, schema, index_col, ...) pandas.io.sql
- f read_table pandas.io.parsers

At the bottom of the dropdown, there is a message: "Press ^ to choose the selected (or first) suggestion and insert a dot afterwards".

Figura 2.4: la vista de la biblioteca pandas en PyCharm IDE

Las funciones y los nombres de las variables son fáciles de seguir incluso sin leer ninguna documentación. Seguir un estilo de codificación estándar hace que la reutilización sea más conveniente.

Documentación bien definida

La documentación bien definida y clara es tan importante como escribir un módulo generalizado e independiente con las pautas de codificación de Python. Sin una documentación clara, el módulo no aumentará el interés de los desarrolladores por reutilizarlo con comodidad. Pero como programadores, nos enfocamos más en el código que en la documentación. Escribiendo unas líneas de documentación puede hacer que 100 líneas de nuestro código sean más usables y fáciles de mantener.

Proporcionaremos un par de buenos ejemplos de documentación desde el punto de vista de un módulo utilizando nuestro ejemplo de módulo mycalculator.py :

```
"""micalculadora.py
Este módulo proporciona funciones para sumar y restar de dos números"""

def suma(x, y):
    """ Esta función suma dos números.
    uso: añadir (3, 4)
    volver x + y

def restar(x, y):
    """ Esta función resta dos números
    uso: restar (17, 8)
    volver x - y
```

En Python, es importante recordar lo siguiente:

- Podemos usar tres caracteres de comillas para marcar una cadena que atraviesa más de una línea del archivo fuente de Python.
- Las cadenas entre comillas triples se usan al comienzo de un módulo, y luego se usa esta cadena como la documentación para el módulo como un todo.
- Si alguna función comienza con una cadena entre comillas triples, esta cadena se usa como documentación para esa función.

Como conclusión general, podemos hacer tantos módulos como queramos escribiendo cientos de líneas de código, pero se necesita más que escribir código para hacer un módulo reutilizable, incluida la generalización, el estilo de codificación y, lo que es más importante, la documentación.

Paquetes de construcción

Hay una serie de técnicas y herramientas disponibles para crear y distribuir paquetes. La verdad es que Python no tiene un gran historial de estandarización del proceso de empaquetado. Ha habido múltiples proyectos iniciados en la primera década del siglo XXI . siglo para agilizar este proceso pero no con mucho éxito. En la última década, hemos tenido cierto éxito, gracias a las iniciativas de **Python Packaging Authority (PyPA)**.

En esta sección, cubriremos las técnicas de creación de paquetes, acceso a los paquetes en nuestro programa y publicación y uso compartido de los paquetes según las pautas proporcionadas por PyPA.

Comenzaremos con los nombres de los paquetes, seguidos por el uso de un archivo de inicialización y luego pasaremos a crear un paquete de muestra.

Denominación

Los nombres de los paquetes deben seguir la misma regla para la denominación que para los módulos, que está en minúsculas sin guiones bajos. Los paquetes actúan como módulos estructurados.

Archivo de inicialización del paquete

Un paquete puede tener un archivo fuente opcional llamado `__init__.py` (o simplemente un archivo `init` expediente). Se recomienda la presencia del archivo `init` (incluso uno en blanco) para marcar las carpetas como paquetes. Desde la versión 3.3 o posterior de Python, el uso de un archivo de inicio es opcional (PEP 420: Paquetes de espacio de nombres implícitos). Puede haber múltiples propósitos para usar este archivo de inicio y siempre hay un debate sobre lo que puede ir dentro de un archivo de inicio versus lo que no puede entrar.
Aquí se analizan algunos usos del archivo `init` :

- **`__init__.py` vacío:** esto obligará a los desarrolladores a usar importaciones explícitas y administrar los espacios de nombres como deseen. Como era de esperar, los desarrolladores tienen que importar módulos separados, lo que puede resultar tedioso para un paquete grande.
- **Importación completa en `__init__.py`:** en este caso, los desarrolladores pueden importar el paquete y luego haga referencia a los módulos directamente en su código usando el nombre del paquete o su nombre de alias. Esto proporciona más comodidad pero a expensas de mantener la lista de todas las importaciones en el archivo `__init__` .

54 Uso de la modularización para manejar proyectos complejos

- **Importación limitada:** este es otro enfoque en el que los desarrolladores de módulos pueden importar solo funciones clave en el archivo init desde diferentes módulos y administrarlas bajo el espacio de nombres del paquete. Esto proporciona el beneficio adicional de proporcionar un contenedor alrededor de la funcionalidad del módulo subyacente. Si por casualidad tuviéramos que refactorizar los módulos subyacentes, tenemos la opción de mantener el mismo espacio de nombres, especialmente para los consumidores de API. El único inconveniente de este enfoque es que requiere un esfuerzo adicional para administrar y mantener dichos archivos de inicio .

A veces, los desarrolladores agregan código al archivo de inicio que se ejecuta cuando se importa un módulo de un paquete. Un ejemplo de dicho código es crear una sesión para sistemas remotos como una base de datos o un servidor SSH remoto.

Construyendo un paquete

Ahora discutiremos cómo construir un paquete con un ejemplo de paquete de muestra. Construiremos un paquete masifutil utilizando los siguientes módulos y un subpaquete:

- El módulo mycalculator.py : Ya construimos este módulo para la Importación sección de módulos.
- El módulo myrandom.py : este módulo también se creó para la sección Importación de módulos.
- El subpaquete advcalc : Este será un subpaquete y contendrá un módulo (advcalculator.py). Definiremos un archivo de inicio para este subpaquete, pero estará vacío.

El módulo advcalculator.py tiene funciones adicionales para calcular la raíz cuadrada y el registro usando base 10 y base 2. El código fuente de este módulo se muestra a continuación:

```
# advcalculator.py con funciones sqrt, log e ln
importar matematicas

def sqrt(x):
    """Esta función saca la raíz cuadrada de un número"""
    devuelve matemática.sqrt(x)

def log(x):
    """Esta función devuelve log de base 10"""
    devuelve matemáticas. log (x, 10)

definición ln(x):
```

```
"""Esta función devuelve log de base 2"""
devuelve matemáticas.log(x,2)
```

La estructura de archivos del paquete masifutil con archivos init se verá así:

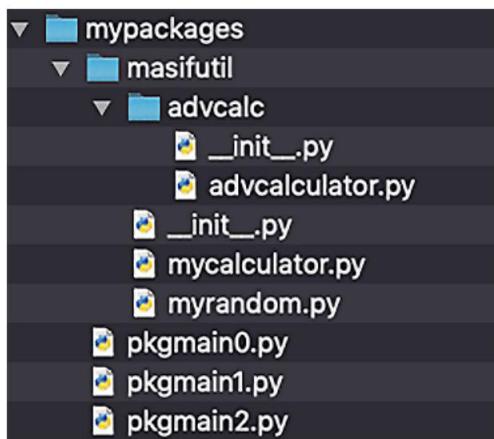


Figura 2.5 – Estructura de carpetas del paquete masifutil con módulos y subpaquetes

En el siguiente paso, crearemos un nuevo script principal (pkgmain1.py) para consumir los módulos del paquete o subcarpeta masifutil . En este script, importaremos los módulos del paquete principal y del subpaquete usando la estructura de carpetas, y luego usaremos las funciones del módulo para calcular dos números aleatorios, la suma y la diferencia de los dos números, y la raíz cuadrada y los valores logarítmicos. de los primeros números aleatorios. El código fuente de pkgmain1.py es el siguiente:

```
# pkgmain0.py con importación directa importar
masifutil.mycalculator como calc
importar masifutil.myrandom como rand
importar masifutil.advcalc.advcalculator como acalc

def mi_principal():
    """ Esta es una función principal que genera dos aleatorios\
números y luego aplicarles funciones de calculadora """
    x = rand.random_2d()
    y = aleatorio.aleatorio_1d()

    suma = calc.add(x,y)
    diff = calc.restar(x,y)
```

56 Uso de la modularización para manejar proyectos complejos

```

raíz = acalc.sqrt(x)
log10x = acalc.log(x)
log2x = acalc.ln(x)

imprimir("x = {}, y = {}".format(x, y))
imprimir("la suma es {}".format(suma))
imprimir("la diferencia es {}".format(dif))
print("la raíz cuadrada es {}".format(sroot))
print("log base de 10 es {}".format(log10x))
print("log base de 2 es {}".format(log2x))

"""
Esto se ejecuta solo si la variable especial '__name__' se establece como principal"""

si __nombre__ == "__principal__":
    mi_principal()

```

Aquí, usaremos el nombre del paquete y el nombre del módulo para importar los módulos, lo cual es engoroso, especialmente cuando necesitamos importar los subpaquetes. También podemos usar las siguientes declaraciones con los mismos resultados:

```

# mypkgmain1.py con sentencias from
desde masifutil importar mycalculator como calc
de masifutil importar myrandom como rand
desde masifutil.advcalc importar advcalculator como acalc
#el resto del código es el mismo que en mypkgmain1.py

```

Como se mencionó anteriormente, el uso del archivo `__init__.py` vacío es opcional. Pero lo hemos agregado en este caso con fines ilustrativos.

A continuación, exploraremos cómo agregar algunas declaraciones de importación al archivo de inicio . Empecemos por importar los módulos dentro del archivo `init` . En este archivo de inicio de nivel superior , importaremos todas las funciones como se muestra a continuación:

```

#__init__ archivo para el paquete 'masifutil'
desde .mycalculator importar sumar, restar
desde .myrandom importar random_1d, random_2d
desde .advcalc.advccalculator importar sqrt, log, ln

```

Tenga en cuenta el uso de `.` antes del nombre del módulo. Esto es necesario para Python para el uso estricto de importaciones relativas.

Como resultado de estas tres líneas dentro del archivo de inicio , el nuevo script principal se simplificará y el código de muestra se muestra a continuación:

```
# pkgmain2.py con función principal importar
masifutil

def mi_principal():
    """ Esta es una función principal que genera dos aleatorios\
números y luego aplicarles funciones de calculadora """
    x = masifutil.random_2d() y =
    masifutil.random_1d()

    suma = masifutil.add(x,y)
    diff = masifutil.restar(x,y)

    sroot = masifutil.sqrt(x) log10x =
    masifutil.log(x) log2x = masifutil.ln(x)

    imprimir("x = {}, y = {}".format(x, y)) imprimir("la suma es
    {}".format(suma)) imprimir("la diferencia es {}".format(dif))
    imprimir ("la raíz cuadrada es {}".format(sroot)) print("la
    base logarítmica de 10 es {}".format(log10x)) print("la base
    logarítmica de 2 es {}".format(log2x))

"""

Esto se ejecuta solo si la variable especial '__name__' se establece como principal"""

if __nombre__ == "__principal__":
    mi_principal()
```

Las funciones de los dos módulos principales y el módulo del subpaquete están disponibles en el nivel del paquete principal y los desarrolladores no necesitan conocer la jerarquía y la estructura subyacentes de los módulos dentro del paquete. Esta es la conveniencia que discutimos anteriormente de usar sentencias de importación dentro del archivo init .

58 Uso de la modularización para manejar proyectos complejos

Creamos el paquete manteniendo el código fuente del paquete en la misma carpeta donde reside el programa o script principal. Esto funciona solo para compartir los módulos dentro de un proyecto.

A continuación, discutiremos cómo acceder al paquete desde otros proyectos y desde cualquier programa desde cualquier lugar.

Acceder a paquetes desde cualquier ubicación

Solo se puede acceder al paquete que construimos en la subsección anterior si el programa que llama a los módulos está en el mismo nivel que la ubicación del paquete. Este requisito no es práctico para la reutilización de códigos y el uso compartido de códigos.

En esta sección, discutiremos algunas técnicas para hacer que los paquetes estén disponibles y se puedan usar desde cualquier programa en cualquier ubicación de nuestro sistema.

Anexando sys.path

Esta es una opción útil para configurar sys.path dinámicamente. Tenga en cuenta que sys.path es una lista de directorios en los que un intérprete de Python busca cada vez que ejecuta una importación instrucción en un programa fuente. Al usar este enfoque, agregamos (agregamos) rutas de directorios o carpetas que contienen nuestros paquetes a sys.path.

Para el paquete masifutil , crearemos un nuevo programa, pkgmain3.py, que es una copia de pkgmain2.py (que se actualizará más adelante), pero se mantiene fuera de la carpeta donde reside nuestro paquete masifutil . pkgmain3.py puede estar en cualquier carpeta que no sea la carpeta mypackages . Aquí está la estructura de carpetas con un nuevo script principal (pkgmain3.py) y el paquete masifutil como referencia:

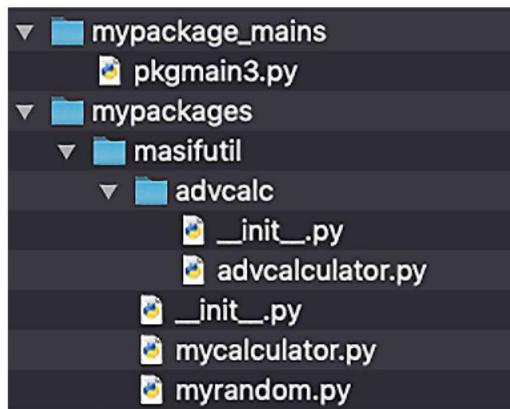


Figura 2.6 – Estructura de carpetas del paquete masifutil y un nuevo script principal, pkgmain3.py

Cuando ejecutamos el programa pkgmain3.py , devuelve un error:

ModuleNotFoundError: No module named 'masifutil'. Esto es de esperar ya que la ruta del paquete masifutil no se agrega a sys.path. Para agregar la carpeta del paquete a sys.path, actualizaremos el programa principal; llamémoslo pkgmain4.py, con declaraciones adicionales para agregar sys.path, que se muestra a continuación:

```
# pkgmain4.py con el código adjunto sys.path

import sys
sys.path.append('/Users/muasif/Google Drive/PythonForGeeks/ source_code/
chapter2/mypackages')

importar masifutil

def my_main():
    """ Esta es una función principal que genera dos números aleatorios y luego les aplica
    funciones de calculadora """
    x = masifutil.random_2d() y = masifutil.random_1d()

    suma = masifutil.add(x,y)
    diff = masifutil.restar(x,y)

    sroot = masifutil.sqrt(x) log10x =
    masifutil.log(x) log2x = masifutil.ln(x)

    imprimir("x = {}, y = {}".format(x, y)) imprimir("la suma es
    {}".format(suma)) imprimir("la diferencia es {}".format(dif))
    imprimir ("la raíz cuadrada es {}".format(sroot)) print("la
    base logarítmica de 10 es {}".format(log10x)) print("la base
    logarítmica de 2 es {}".format(log2x))
```

60 Uso de la modularización para manejar proyectos complejos

```
""" Esto se ejecuta solo si la variable especial '__name__' se establece como principal"""

si __nombre__ == "__principal__":
    mi_principal()
```

Después de agregar las líneas adicionales para agregar sys.path, ejecutamos el script principal sin ningún error y con la salida de consola esperada. Esto se debe a que nuestro masifutil El paquete ahora está disponible en una ruta donde el intérprete de Python puede cargarlo cuando lo estamos importando en nuestro script principal.

Como alternativa a agregar sys.path, también podemos usar la función site.addsitedir del módulo del sitio. La única ventaja de usar este enfoque es que esta función también busca archivos .pth dentro de las carpetas incluidas, lo que es útil para agregar carpetas adicionales, como subpaquetes. A continuación se muestra un fragmento de un script principal de muestra (pktpamin5.py) con la función addsitedir :

```
# pkgmain5.py

sitio de importación
sitio.addsitedir('/Usuarios/muasif/Google Drive/PythonForGeeks/
código_fuente/capítulo2/mispaquetes')

importar masifutil
#resto del código es el mismo que en pkymain4.py
```

Tenga en cuenta que los directorios que agregamos o agregamos usando este enfoque están disponibles solo durante la ejecución del programa. Para establecer sys.path de forma permanente (a nivel de sesión o de sistema), los enfoques que analizaremos a continuación son más útiles.

Usando la variable de entorno PYTHONPATH

Esta es una manera conveniente de agregar nuestra carpeta de paquetes a sys.path, que el intérprete de Python usará para buscar el paquete y los módulos si no están presentes en la biblioteca integrada. Dependiendo del sistema operativo que estemos usando, podemos definir esta variable de la siguiente manera.

En Windows, la variable de entorno se puede definir usando cualquiera de las siguientes opciones:

- **La línea de comando:** Establecer PYTHONPATH = "C:\pythonpath1;C:\pythonpath2". Esto es bueno para una sesión activa.
- **La interfaz gráfica de usuario:** Vaya a **Mi PC | Propiedades | Configuración avanzada del sistema | Variables de entorno.** Este es un escenario permanente.

En Linux y macOS, se puede configurar usando export PYTHONPATH= `/some/path/` .

Si se establece mediante Bash o un terminal equivalente, la variable de entorno será efectiva solo para la sesión del terminal. Para configurarlo de forma permanente, se recomienda agregar la variable de entorno al final de un archivo de perfil, como ~/bash_profile.

Si ejecutamos el programa pkgmain3.py sin configurar PYTHONPATH, devuelve un error: ModuleNotFoundError: No module named 'masifutil'. Esto se espera nuevamente ya que la ruta del paquete masifutil no se agrega a PYTHONPATH.

En el siguiente paso, agregaremos la ruta de la carpeta que contiene masifutil a PYTHONPATH variable y vuelva a ejecutar el programa pkgmain3 . Esta vez funciona sin ningún error y con la salida de consola esperada.

Usando el archivo .pth bajo el paquete del sitio de Python

Esta es otra forma conveniente de agregar paquetes a sys.path. Esto se logra definiendo un archivo .pth en los paquetes del sitio de Python. El archivo puede contener todas las carpetas que queremos agregar a sys.path.

62 Uso de la modularización para manejar proyectos complejos

Con fines ilustrativos, creamos un archivo `my.pth` en `venv /lib/Python3.7/site-packages`. Como podemos ver en la Figura 2.7, agregamos una carpeta que contiene nuestro paquete `masifutil`. Con este simple archivo `.pth`, nuestro script principal `pkymain3.py` funciona bien sin ningún error y con la salida de consola esperada:

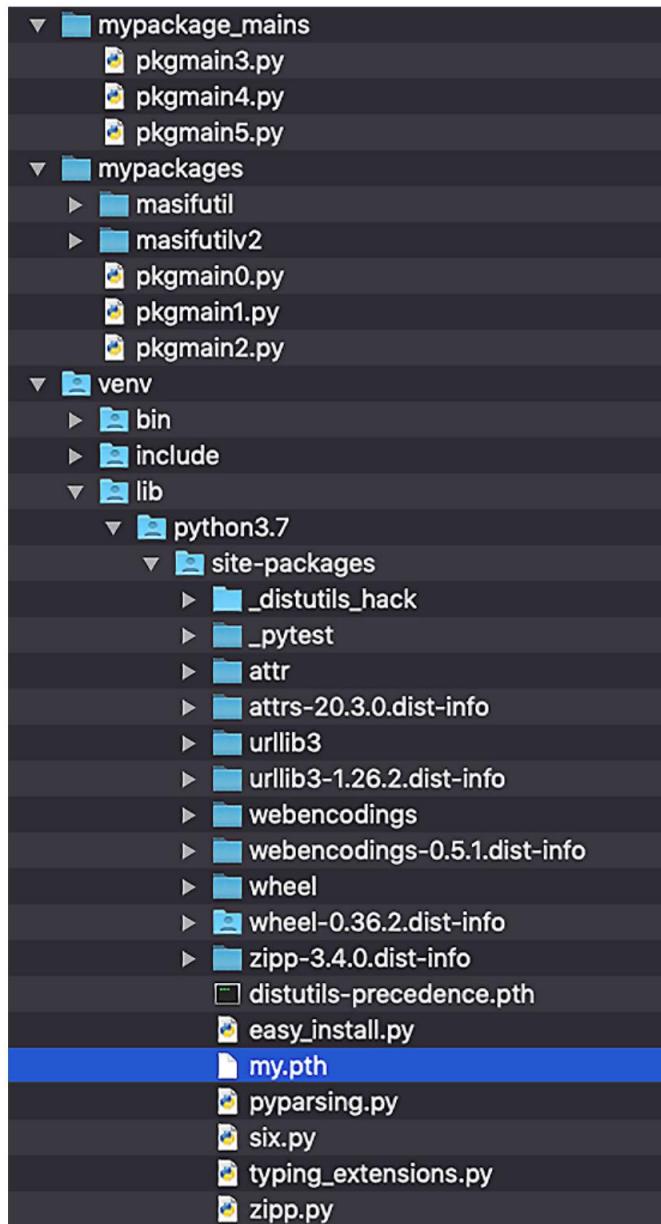


Figura 2.7 – Vista de un entorno virtual con el archivo `my.pth`

Los enfoques que discutimos para acceder a paquetes personalizados son efectivos para reutilizar los paquetes y módulos en el mismo sistema con cualquier programa. En la siguiente sección, exploraremos cómo compartir paquetes con otros desarrolladores y comunidades.

Compartir un paquete

Hay muchas herramientas disponibles para distribuir paquetes y proyectos de Python entre comunidades. Nos centraremos solo en las herramientas que se recomiendan según las pautas proporcionadas por PyPA.

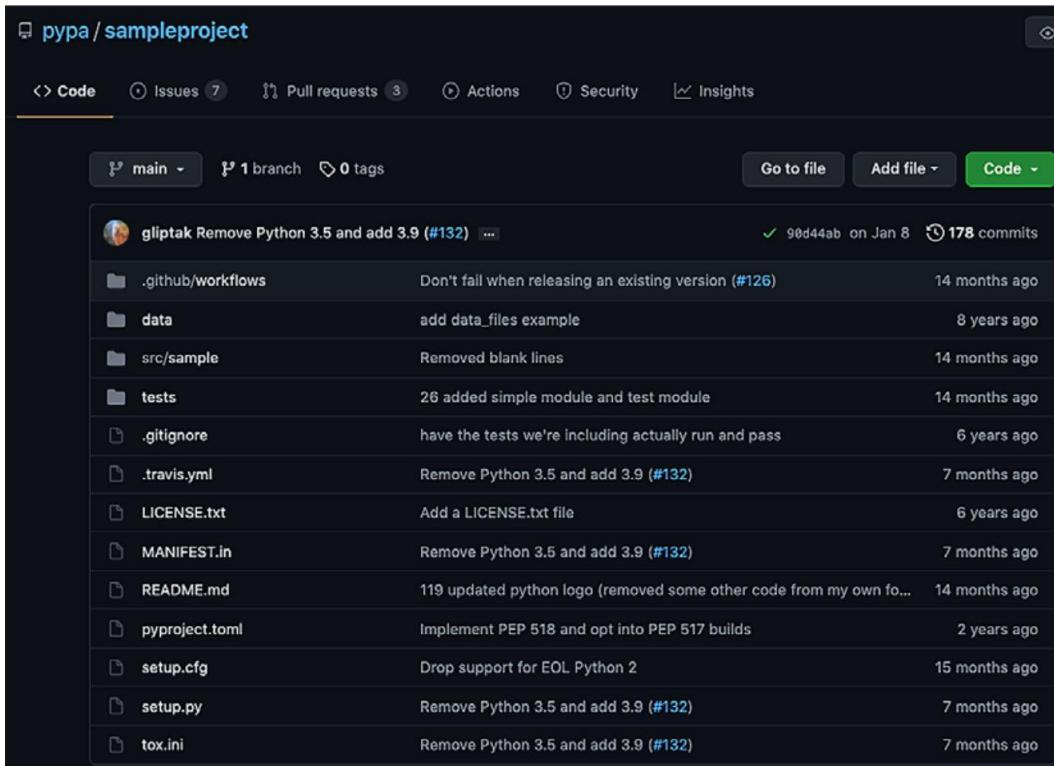
En esta sección, cubriremos las técnicas de instalación y distribución de embalaje. Algunas herramientas que usaremos o que al menos vale la pena mencionar en esta sección como referencia son las siguientes:

- **distutils:** Esto viene con Python con funcionalidad básica. No es fácil de extender para la distribución de paquetes complejos y personalizados.
- **setuptools:** Esta es una herramienta de terceros y una extensión de distutils y es Recomendado para la construcción de paquetes.
- **rueda:** Esto es para el formato de empaquetado de Python y hace que las instalaciones sean más rápidas y sencillas en comparación con sus predecesores.
- **pip:** pip es un administrador de paquetes y módulos de Python, y viene como parte de Python si está instalando la versión 3.4 o posterior de Python. Es fácil usar pip para instalar un nuevo módulo usando un comando como `pip install <nombre del módulo>`.
- **The Python Package Index (PyPI):** este es un repositorio de software para el lenguaje de programación Python. PyPI se utiliza para buscar e instalar software desarrollado y compartido por la comunidad de Python.
- **Twine:** esta es una utilidad para publicar paquetes de Python en PyPI.

En las siguientes subsecciones, actualizaremos el paquete masifutil para incluir componentes adicionales según las pautas proporcionadas por PyPA. A continuación, se instalará el paquete masifutil actualizado en todo el sistema mediante pip. Al final, publicaremos el paquete masifutil actualizado en **Test PyPI** y lo instalaremos desde Test PyPI.

Creación de un paquete según las pautas de PyPA

PyPA recomienda usar un proyecto de muestra para crear paquetes reutilizables y está disponible en <https://github.com/pypa/sampleproject>. Se muestra un fragmento del proyecto de muestra de la ubicación de GitHub:



The screenshot shows the GitHub repository page for 'pypa / sampleproject'. The main branch has 178 commits. Key commits include:

- Remove Python 3.5 and add 3.9 (#132) - Don't fail when releasing an existing version (#126)
- add data_files example
- Removed blank lines
- 26 added simple module and test module
- have the tests we're including actually run and pass
- Remove Python 3.5 and add 3.9 (#132)
- Add a LICENSE.txt file
- Remove Python 3.5 and add 3.9 (#132)
- 119 updated python logo (removed some other code from my own fo...)
- Implement PEP 518 and opt into PEP 517 builds
- Drop support for EOL Python 2
- Remove Python 3.5 and add 3.9 (#132)
- Remove Python 3.5 and add 3.9 (#132)

Figura 2.8 – Una vista del proyecto de muestra en GitHub por PyPA

Presentaremos archivos y carpetas clave, que es importante comprender antes de usarlos para actualizar nuestro paquete masifutil :

- **setup.py:** este es el archivo más importante, que debe existir en la raíz del proyecto o paquete.
Es un script para construir e instalar el paquete. Este archivo contiene una función de instalación global(). El archivo de instalación también proporciona una interfaz de línea de comandos para ejecutar varios comandos.
- **setup.cfg:** Este es un archivo ini que setup.py puede usar para definir los valores predeterminados.

- `setup()` args: los argumentos clave que se pueden pasar a la función de configuración son los siguientes: sigue:

un nombre

b) Versión

c) Descripción

d) URL

e) Autor

f) Licencia

- `README.rst/README.md`: este archivo (ya sea en formato reStructured o Markdown) puede contener información sobre el paquete o proyecto.

- `license.txt`: el archivo `license.txt` debe incluirse con cada paquete con detalles de los términos y condiciones de distribución. El archivo de licencia es importante, especialmente en países donde es ilegal distribuir paquetes sin la licencia correspondiente.

- `MANIFEST.in`: este archivo se puede utilizar para especificar una lista de archivos adicionales para incluir en el paquete. Esta lista de archivos no incluye los archivos de código fuente (que se incluyen automáticamente). • `<paquete>`: este es el paquete de nivel superior que contiene todos los módulos y paquetes en su interior. No es obligatorio su uso, pero es un enfoque recomendado.

- `datos`: este es un lugar para agregar archivos de datos si es necesario.

- `pruebas`: este es un marcador de posición para agregar pruebas unitarias para los módulos.

66 Uso de la modularización para manejar proyectos complejos

Como siguiente paso, actualizaremos nuestro paquete masifutil anterior según las pautas de PyPA. Aquí está la nueva estructura de carpetas y archivos del paquete masifutilv2 actualizado:

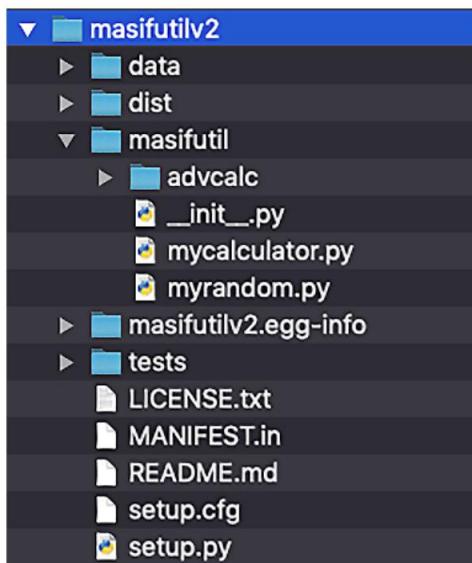


Figura 2.9: una vista de la estructura de archivos masifutilv2 actualizada

Hemos agregado directorios de datos y pruebas , pero en realidad están vacíos por ahora. Evaluaremos las pruebas unitarias en un capítulo posterior para completar este tema.

El contenido de la mayoría de los archivos adicionales se cubre en el proyecto de muestra y, por lo tanto, no se tratará aquí, excepto el archivo setup.py .

Actualizamos setup.py con argumentos básicos según nuestro proyecto de paquete. Los detalles del resto de los argumentos están disponibles en el archivo setup.py de muestra proporcionado con el proyecto de muestra por PyPA. Aquí hay un fragmento de nuestro archivo setup.py :

```
desde la configuración de importación de herramientas de configuración
configuración(
    nombre='masifutilv2',
    versión='0.1.0',
```

```
autor='Muhammad Asif',
autor_email='ma@ejemplo.com',
paquetes=['masifutil', 'masifutil/advcalc'],
python_requires='>=3.5, <4',
url='http://pypi.python.org/pypi/Nombre del paquete/',
licencia='LICENCIA.txt',
description='Un paquete de muestra con fines ilustrativos',
long_description=open('README.md').leer(),
install_requires=[
],
)
```

Con este archivo setup.py , estamos listos para compartir nuestro paquete masifutilv2 de forma local y remota, lo cual analizaremos en las siguientes subsecciones.

Instalación desde el código fuente local usando pip

Una vez que hayamos actualizado el paquete con nuevos archivos, estamos listos para instalarlo usando la utilidad pip. La forma más sencilla de instalarlo es ejecutando el siguiente comando con la ruta a la carpeta masifutilv2 :

```
> pip install <ruta a masifutilv2>
```

La siguiente es la salida de la consola del comando cuando se ejecuta sin instalar el paquete de ruedas:

```
Procesando ./masifutilv2
```

```
Usando 'setup.py install' heredado para masifutilv2, ya que el paquete 'wheel' no está instalado.
```

```
Instalación de paquetes recopilados: masifutilv2
```

```
Ejecutando setup.py install para masifutilv2... hecho
```

```
Masifutilv2-0.1.0 instalado con éxito
```

68 Uso de la modularización para manejar proyectos complejos

La utilidad pip instaló el paquete con éxito pero usando el formato de huevo ya que el paquete de rueda no estaba instalado. Aquí hay una vista de nuestro entorno virtual después de la instalación:

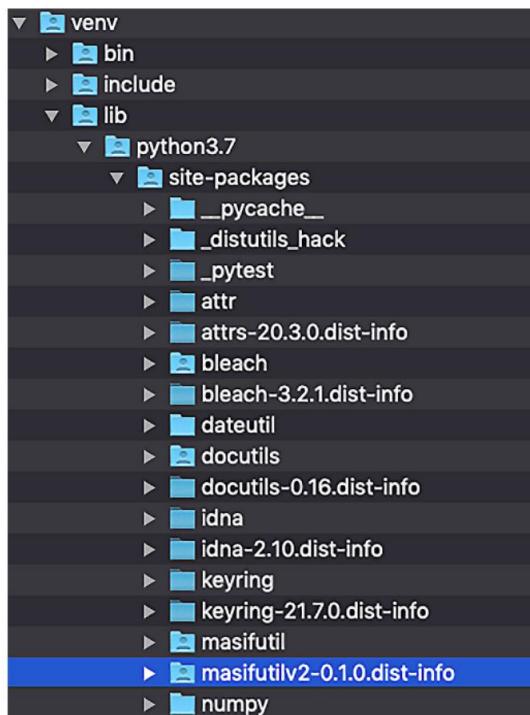


Figura 2.10: una vista del entorno virtual después de instalar masifutilv2 usando pip

Después de instalar el paquete en el entorno virtual, lo probamos con nuestro programa `pkgmain3.py`, que funcionó como se esperaba.

Sugerencia Para desinstalar el paquete, podemos usar `pip uninstall masifutilv2`.

Como siguiente paso, instalaremos el paquete de ruedas y luego volveremos a instalar el mismo paquete. Aquí está el comando de instalación:

```
> pip install <ruta a masifutilv2>
```

La salida de la consola será similar a la siguiente:

```
Procesando ./masifutilv2
```

```
Construcción de ruedas para paquetes recogidos: masifutilv2
```

```
Rueda de construcción para masifutilv2 (setup.py) ... hecho
```

```
Rueda creada para masifutilv2: filename=masi
futilv2-0.1.0-py3-none-any.whl tamaño=3497
sha256=038712975b7d7eb1f3fefa799da9e294b34
e79caea24abb444dd81f4cc44b36e

Almacenado en la carpeta: /private/var/folders/xp/g88fvmgs0k90w0rc_
qq4xkzxpsx11v/T/pip-ephem-wheel-cache-l2eyp_wq/wheels/
de/14/12/71b4d696301fd1052adf287191fdd054cc17ef6c9b59066277

Masifutilv2 construido con éxito
Instalación de paquetes recopilados: masifutilv2
Masifutilv2-0.1.0 instalado con éxito
```

El paquete se instaló con éxito usando la rueda esta vez y podemos ver que aparece en nuestro entorno virtual de la siguiente manera:

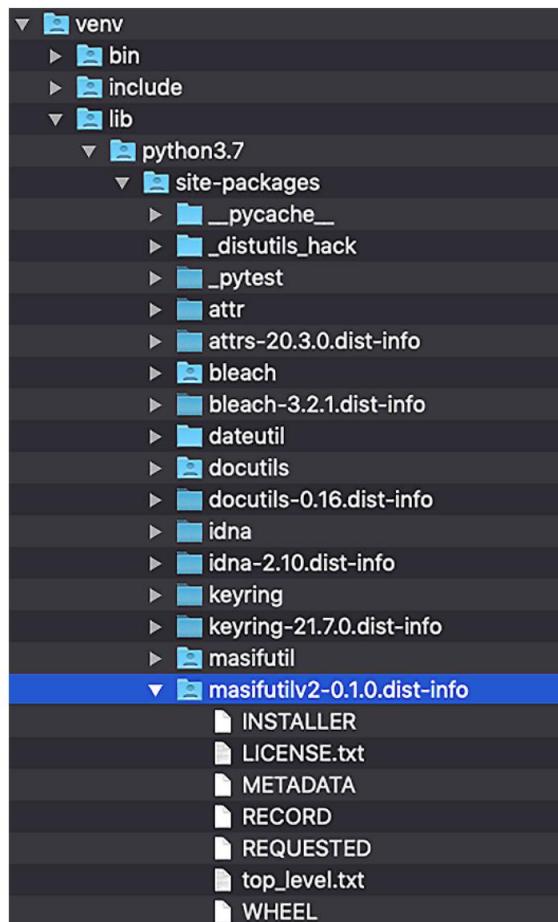


Figura 2.11 – Una vista del entorno virtual después de instalar masifutilv2 con rueda y usando pip

70 Uso de la modularización para manejar proyectos complejos

En esta sección, hemos instalado un paquete utilizando la utilidad pip del código fuente local.

En la siguiente sección, publicaremos el paquete en un repositorio centralizado (Test PyPI).

Publicación de un paquete para probar PyPI

Como siguiente paso, agregaremos nuestro paquete de muestra al repositorio de PyPI. Antes de ejecutar cualquier comando para publicar nuestro paquete, necesitaremos crear una cuenta en Test PyPI. Tenga en cuenta que Test PyPI es una instancia separada del índice del paquete específicamente para pruebas. Además de la cuenta con Test PyPI, también debemos agregar un **token API** a la cuenta. Le dejaremos los detalles para crear una cuenta y agregar un token API a la cuenta siguiendo las instrucciones disponibles en el sitio web de Test PyPI (<https://prueba.pypi.org/>).

Para enviar el paquete a Test PyPI, necesitaremos la utilidad Twine. Suponemos que Twine está instalado usando la utilidad pip. Para subir el paquete masifutilv2 ejecutaremos los siguientes pasos:

1. Cree una distribución usando el siguiente comando. Esta utilidad sdist creará un archivo TAR ZIP en una carpeta dist :

```
> python setup.py sdist
```

2. Suba el archivo de distribución a Test PyPI. Cuando se le solicite un nombre de usuario y contraseña, proporcione _token_ como nombre de usuario y el token de API como contraseña:

```
> subir hilo --repositorio testpypi dist/masifutilv2-  
0.1.0.tar.gz
```

Este comando enviará el archivo TAR ZIP del paquete al repositorio Test PyPI y la salida de la consola será similar a la siguiente:

```
Subiendo distribuciones a https://test.pypi.org/legacy/
```

```
Ingrese su nombre de usuario: _token_
```

```
Introduce tu contraseña:
```

```
Subiendo masifutilv2-0.1.0.tar.gz
```

```
100%|████████████████████████████████| 5,15 k/
```

```
5,15 k [00:02<00:00, 2,21 kB/s]
```

Podemos ver el archivo cargado en <https://test.pypi.org/project/masifutilv2/0.1.0/> después de una carga exitosa.

Instalando el paquete desde PyPI

Instalar el paquete desde Test PyPI es lo mismo que instalarlo desde un repositorio regular, excepto que necesitamos proporcionar la URL del repositorio usando los argumentos `index-url`. El comando y la salida de la consola serán similares a los siguientes:

```
> pip install --index-url https://test.pypi.org/simple/ --no deps masifutilv2
```

Este comando presentará una salida de consola similar a la siguiente:

```
Buscando en índices: https://test.pypi.org/simple/
Recoleciendo masifutilv2
Descargando https://test-files.pythonhosted.org/
paquetes/b7/e9/7afe390b4ec1e5842e8e62a6084505cbc6b9
f6adf0e37ac695cd23156844/masifutilv2-0.1.0.tar.gz (2,3 kB)
Construcción de ruedas para paquetes recogidos: masifutilv2
Rueda de construcción para masifutilv2 (setup.py) ... hecho
Rueda creada para masifutilv2: filename=masifutilv2-
0.1.0-py3-none-any.whl tamaño=3497
sha256=a3db8f04b118e16ae291bad9642483874
f5c9f447dbe57c0961b5f8fb99501
Almacenado en carpeta: /Users/muasif/Library/Caches/pip/
ruedas/1c/47/29/95b9edfe28f02a605757c1
f1735660a6f79807ece430f5b836
Masifutilv2 construido con éxito
Instalación de paquetes recopilados: masifutilv2
Masifutilv2-0.1.0 instalado con éxito
```

Como podemos ver en la salida de la consola, pip está buscando el módulo en Test PyPI. Una vez que encuentra el paquete con el nombre `masifutilv2`, comienza a descargarlo y luego lo instala en el entorno virtual.

En resumen, hemos observado que una vez que creamos un paquete usando el formato y el estilo recomendados, publicar y acceder al paquete es solo una cuestión de usar las utilidades de Python y seguir los pasos estándar.

Resumen

En este capítulo, presentamos el concepto de módulos y paquetes en Python.

Discutimos cómo construir módulos reutilizables y cómo pueden ser importados por otros módulos y programas. También cubrimos la carga e inicialización de módulos cuando están incluidos (mediante un proceso de importación) por otros programas. En la última parte de este capítulo, discutimos la construcción de paquetes simples y avanzados. También proporcionamos una gran cantidad de ejemplos de código para acceder a los paquetes, además de instalar y publicar el paquete para una reutilización eficiente.

Después de leer este capítulo, ha aprendido cómo crear módulos y paquetes y cómo compartir y publicar los paquetes (y módulos). Estas habilidades son importantes si está trabajando en un proyecto como equipo en una organización o está creando bibliotecas de Python para una comunidad más grande.

En el próximo capítulo, discutiremos el siguiente nivel de modularización usando programación orientada a objetos en Python. Esto abarcará la encapsulación, la herencia, el polimorfismo y la abstracción, que son herramientas clave para construir y administrar proyectos complejos en el mundo real.

Preguntas

1. ¿Cuál es la diferencia entre un módulo y un paquete?
2. ¿Qué son las importaciones absolutas y relativas en Python?
3. ¿Qué es PyPA?
4. ¿Qué es Test PyPI y por qué lo necesitamos?
5. ¿Es un archivo de inicio un requisito para construir un paquete?

Otras lecturas

- Programación Modular con Python por Erik Westra
- Programación Expertas en Python por Michaÿ Jaworski y Tarek Ziadé
- Guía del usuario de empaquetado de Python (<https://packaging.python.org/>)
- PEP 420: paquetes de espacios de nombres implícitos (<https://www.python.org/dev/peps/pep-0420/>)

respuestas

1. Un módulo está destinado a organizar funciones, variables y clases en Archivos de código Python. Un paquete de Python es como una carpeta para organizar varios módulos o subpaquetes.
2. La importación absoluta requiere el uso de la ruta absoluta de un paquete a partir del nivel superior, mientras que la importación relativa se basa en la ruta relativa del paquete según la ubicación actual del programa en el que se utilizará la declaración de importación .
3. La **Python Packaging Authority (PyPA)** es un grupo de trabajo que mantiene un núcleo conjunto de proyectos de software utilizados en el empaquetado de Python.
4. Test PyPI es un repositorio de software para el lenguaje de programación Python con fines de prueba.
5. El archivo de inicio es opcional desde la versión 3.3 de Python.

3

Objeto avanzado Python orientado Programación

Python se puede usar como un lenguaje de programación modular declarativo, como C, así como para la programación imperativa o la **programación orientada a objetos (POO)** completa con lenguajes de programación como Java. La **programación declarativa** es un paradigma en el que nos enfocamos en lo que queremos implementar, mientras que la **programación imperativa** es donde describimos los pasos exactos de cómo implementar lo que queremos. Python es adecuado para ambos tipos de paradigmas de programación. OOP es una forma de programación imperativa en la que agrupamos las propiedades y comportamientos de los objetos del mundo real en programas. Además, OOP también aborda las relaciones entre diferentes tipos de objetos del mundo real.

En este capítulo, exploraremos cómo se pueden implementar los conceptos avanzados de programación orientada a objetos utilizando Python. Suponemos que está familiarizado con conceptos generales como clases, objetos e instancias y tiene conocimientos básicos de herencia entre objetos.

Cubriremos los siguientes temas en este capítulo:

- Introducción de clases y objetos

• Comprensión de los principios de programación orientada a objetos

76 Programación Python avanzada orientada a objetos

- Usar la composición como un enfoque de diseño alternativo
- Introducción a la tipificación de patos en Python
- Aprender cuándo no usar OOP en Python

Requerimientos técnicos

Estos son los requisitos técnicos para este capítulo:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter03>.

Introducción a clases y objetos.

Una clase es un modelo de cómo debe definirse algo. En realidad, no contiene ningún dato: es una plantilla que se utiliza para crear instancias según las especificaciones definidas en una plantilla o un blueprint.

Un objeto de una clase es una instancia que se construye a partir de una clase, y por eso también se le llama instancia de una clase. En el resto de este capítulo y este libro, nos referiremos a objeto e instancia como sinónimos. Los objetos en programación orientada a objetos ocasionalmente se representan mediante objetos físicos como mesas, sillas o libros. En la mayoría de las ocasiones, los objetos de un programa de software representan entidades abstractas que pueden no ser físicas, como cuentas, nombres, direcciones y pagos.

Para refrescarnos con los conceptos básicos de clases y objetos, definiremos estas terminologías con ejemplos de código.

Distinguir entre atributos de clase y atributos de instancia

Los **atributos de clase** se definen como parte de la definición de clase y sus valores deben ser los mismos en todas las instancias creadas a partir de esa clase. Se puede acceder a los atributos de la clase usando el nombre de la clase o el nombre de la instancia, aunque se recomienda usar un nombre de clase para acceder a estos atributos (para lectura o actualización). Los **atributos de instancia** proporcionan el estado o los datos de un objeto .

La definición de una clase en Python se hace simplemente usando la palabra clave `class`. Como se discutió en el Capítulo 1, Ciclo de vida de desarrollo óptimo de Python, el nombre de la clase debe ser CamelCase. El siguiente fragmento de código crea una clase `Car`:

```
#carexample1.py
```

```
Coche de clase:
```

```
aprobado
```

Esta clase no tiene atributos ni métodos. Es una clase vacía, y puede pensar que esta clase es inútil hasta que le agreguemos más componentes. ¡No exactamente! En Python, puede agregar atributos sobre la marcha sin definirlos en la clase. El siguiente fragmento es un ejemplo válido de código en el que agregamos atributos a una instancia de clase en tiempo de ejecución:

```
#carexample1.py
```

```
Coche de clase:
```

```
aprobado
```

```
si __nombre__ == "__principal__":
```

```
coche = coche()
```

```
coche.color = "azul"
```

```
coche.millas = 1000
```

```
imprimir(coche.color)
```

```
imprimir(coche.millas)
```

En este ejemplo extendido, creamos una instancia (auto) de nuestra clase `Car` y luego agregamos dos atributos a esta instancia: `color` y `millas`. Tenga en cuenta que los atributos agregados con este enfoque son atributos de instancia.

A continuación, agregaremos atributos de clase y atributos de instancia mediante un método constructor (`__init__`), que se carga en el momento de la creación del objeto. A continuación se muestra un fragmento de código con dos atributos de instancia (`color` y `millas`) y el método `init`:

```
#carexample2.py
```

```
Coche de clase:
```

```
c_milla_unidades = "Mi"
```

```
def __init__(uno mismo, color, millas):
```

```
self.i_color = color
```

```
self.i_mileage = millas
```

```
si __nombre__ == "__principal__":
```

```
coche1 = Coche("azul", 1000)
```

78 Programación Python avanzada orientada a objetos

```
imprimir (coche.i_color)
imprimir (coche.i_kilometraje)
imprimir (car.c_mileage_units)
imprimir (Coche.c_millas_unidades)
```

En este programa hicimos lo siguiente:

1. Creamos una clase Car con un atributo de clase c_mileage_units y dos variables de instancia, i_color e i_mileage.
2. Creamos una instancia (car) de la clase Car .
3. Imprimimos los atributos de la instancia utilizando la variable de instancia del automóvil .
4. Imprimimos el atributo de clase utilizando la variable de instancia de coche , así como el Car nombre de la clase. La salida de la consola es la misma para ambos casos.

Nota IMPORTANTE

`self` es una referencia a la instancia que se está creando. El uso de `self` es común en Python para acceder a los atributos y métodos de instancia dentro del método de instancia, incluido el método `init` . `self` no es una palabra clave, y no es obligatorio usar la palabra `self`. Puede ser cualquier cosa como `esto` o `bla`, excepto que tiene que ser el primer parámetro de los métodos de instancia, pero la convención de usar `self` como nombre de argumento es demasiado fuerte.

Podemos actualizar los atributos de la clase usando una variable de instancia o un nombre de clase, pero el resultado puede ser diferente. Cuando actualizamos un atributo de clase usando el nombre de la clase, se actualiza para todas las instancias de esa clase. Pero si actualizamos un atributo de clase usando una variable de instancia, se actualizará solo para esa instancia en particular. Esto se demuestra en el siguiente fragmento de código, que usa la clase Car :

```
#carexample3.py
La definición de #class de Class Car es la misma que en carexample2.py
si __nombre__ == "__principal__":
    coche1 = Coche ("azul", 1000)
    coche2 = Coche("rojo", 2000)

imprimir("usando coche1: " + coche1.c_unidades_de_kilometraje)
```

```
print("usando coche2: " + coche2.c_unidades_kilometraje)
imprimir("usando Clase: " + Car.c_mileage_units)

coche1.c_millas_unidades = "km"

print("usando coche1: " + coche1.c_unidades_kilometraje)
print("usando coche2: " + coche2.c_unidades_kilometraje)
imprimir("usando Clase: " + Car.c_mileage_units)

Coche.c_mileage_units = "NP"
print("usando coche1: " + coche1.c_unidades_kilometraje)
print("usando coche2: " + coche2.c_unidades_kilometraje)
imprimir("usando Clase: " + Car.c_mileage_units)
```

La salida de la consola de este programa se puede analizar de la siguiente manera:

1. El primer conjunto de declaraciones de impresión generará el valor predeterminado del atributo de clase, que es mi.
2. Después de ejecutar la instrucción car1.c_mileage_units = "km" , el valor del atributo de clase será el mismo (Mi) para la instancia car2 y el atributo de nivel de clase.
3. Después de ejecutar la instrucción Car.c_mileage_units = "NP" , el valor de el atributo de clase para el coche 2 y el nivel de la clase cambiarán a NP, pero seguirán siendo los mismos (km) para el coche 1 tal como lo establecimos explícitamente nosotros.

Nota IMPORTANTE

Los nombres de atributos comienzan con c e i para indicar que son variables de clase e instancia, respectivamente, y no variables locales o globales regulares. El nombre de los atributos de la instancia no pública debe comenzar con un guión bajo simple o doble para que sean protegidos o privados. Esto se discutirá más adelante en el capítulo.

Uso de constructores y destructores con clases

Al igual que con cualquier otro lenguaje OOP, Python también tiene constructores y destructores, pero la convención de nomenclatura es diferente. El propósito de tener constructores en una clase es inicializar o asignar valores a los atributos de nivel de clase o instancia (principalmente atributos de instancia) cada vez que se crea una instancia de una clase. En Python, el `__init__` método se conoce como constructor y siempre se ejecuta cuando se crea una nueva instancia. Hay tres tipos de constructores admitidos en Python, enumerados a continuación:

- **Constructor por defecto:** Cuando no incluimos ningún constructor (el `__init__` method) en una clase u olvida declararlo, entonces esa clase usará un constructor predeterminado que está vacío. El constructor no hace nada más que inicializar la instancia de una clase.
- **Constructor no parametrizado:** Este tipo de constructor no toma ningún argumentos excepto una referencia a la instancia que se está creando. El siguiente ejemplo de código muestra un constructor no parametrizado para una clase Name ::

```
nombre de la clase:  
#constructor no parametrizado  
def __init__(uno mismo):  
    print("Una nueva instancia de la clase Nombre es \  
creado")
```

Dado que no se pasan argumentos con este constructor, tenemos una funcionalidad limitada para agregarle. Por ejemplo, en nuestro código de muestra, enviamos un mensaje a la consola de que se ha creado una nueva instancia para la clase Nombre

- **Constructor parametrizado:** un constructor parametrizado puede tomar uno o más argumentos, y el estado de la instancia se puede establecer según los argumentos de entrada proporcionados a través del método constructor. La clase Name se actualizará con un constructor parametrizado, de la siguiente manera:

```
nombre de la clase:  
#constructor parametrizado  
def __init__(uno mismo, primero, último):  
    self.i_first = primero  
    self.i_last = último
```

Los destructores son lo opuesto a los constructores: se ejecutan cuando se elimina o destruye una instancia. En Python, los destructores apenas se usan porque Python tiene un recolector de basura que maneja la eliminación de las instancias a las que ya no hace referencia ninguna otra instancia o programa. Si necesitamos agregar lógica dentro de un método destructor, podemos implementarlo usando un método especial `__del__`. Se llama automáticamente cuando se eliminan todas las referencias de una instancia. Aquí está la sintaxis de cómo definir un método destructor en Python:

```
def __del__(uno mismo):
    print("Objeto eliminado")
```

Distinguir entre métodos de clase y métodos de instancia

En Python podemos definir tres tipos de métodos en una clase, los cuales se describen a continuación:

- **Métodos de instancia:** Están asociados a una instancia y necesitan que se cree una instancia antes de ejecutarlos. Aceptan el primer atributo como referencia a la instancia (`self`) y pueden leer y actualizar el estado de la instancia. `__init__`, que es un método constructor, es un ejemplo de un método de instancia.

- **Métodos de clase:** estos métodos se declaran con el decorador `@classmethod` .
Estos métodos no necesitan una instancia de clase para su ejecución. Para este método, la referencia de clase (`cls` se usa como convención) se enviará automáticamente como primer argumento.

- **Métodos estáticos:** estos métodos se declaran con el decorador `@staticmethod` .
No tienen acceso a `cls` u objetos propios . Los métodos estáticos son como funciones de utilidad que toman ciertos argumentos y proporcionan la salida en función de los valores de los argumentos; por ejemplo, si necesitamos evaluar ciertos datos de entrada o analizar datos para su procesamiento, podemos escribir métodos estáticos para lograr estos objetivos. Los métodos estáticos funcionan como funciones regulares que definimos en módulos pero están disponibles en el contexto del espacio de nombres de la clase.

Para ilustrar cómo se pueden definir estos métodos y luego usarlos en Python, creamos un programa simple, que se muestra a continuación:

```
#métodosejemplo1.py
Coché de clase:
c_milla_unidades = "Mi"

def __init__(uno mismo, color, millas):
```

82 Programación Python avanzada orientada a objetos

```
self.i_color = color self.i_mileage = millas

def print_color (self): print (f"El color del
auto es {self.i_color}")

@métodoclase

def print_units(cls): print (f"la unidad de
kilometraje es {cls.c_mileage_unit}") print(f"el nombre de la clase es {cls.__name__}")

@métodoestático

def print_hello(): print ("Hola desde
un método estático")

si __nombre__ == "__principal__":
coche = Coche ("azul", 1000)
coche.imprimir_color()
coche.imprimir_unidades()
coche.imprimir_hola()

Coche.print_color(coche);
Coche.imprimir_unidades();
Coche.imprimir_hola()
```

En este programa hicimos lo siguiente:

1. Creamos una clase Car con un atributo de clase (c_mileage_units), un método de clase (print_units), un método estático (print_hello), atributos de instancia (i_color e i_mileage), un método de instancia (print_color) y un método constructor (__init__).
2. Creamos una instancia de la clase Car usando su constructor como car.
3. Usando la variable de instancia (coche en este ejemplo), llamamos al método de instancia, al método de clase y al método estático.

4. Usando el nombre de la clase (Car en este ejemplo), nuevamente activamos la instancia método, el método de clase y el método estático. Tenga en cuenta que podemos activar el método de instancia usando el nombre de la clase, pero debemos pasar la variable de instancia como primer argumento (esto también explica por qué necesitamos el argumento propio para cada método de instancia).

La salida de la consola de este programa se muestra a continuación como referencia:

```
El color del coche es azul.  
unidad de kilometraje son Mi  
el nombre de la clase es coche  
Hola desde un método estático  
El color del coche es azul.  
unidad de kilometraje son Mi  
el nombre de la clase es coche  
Hola desde un método estático
```

Métodos especiales

Cuando definimos una clase en Python e intentamos imprimir una de sus instancias usando una impresión declaración, obtendremos una cadena que contiene el nombre de la clase y la referencia de la instancia del objeto, que es la dirección de memoria del objeto. No hay una implementación predeterminada de la funcionalidad de cadena disponible con una instancia u objeto. El fragmento de código que muestra este comportamiento se presenta aquí:

```
#carexampl4.py  
Coche de clase:  
  
def __init__(uno mismo, color, millas):  
    self.i_color = color  
    self.i_mileage = millas  
  
    si __nombre__ == "__principal__":  
        coche = Coche ("azul", 1000)  
    imprimir (coche)
```

Obtendremos una salida de consola similar a la siguiente, que no es lo que se espera de una declaración de impresión :

```
<__main__.Objeto de coche en 0x100caae80>
```

84 Programación Python avanzada orientada a objetos

Para obtener algo significativo de una declaración de impresión , debemos implementar un método `__str__` especial que devolverá una cadena con información sobre la instancia y que se puede personalizar según sea necesario. Aquí hay un fragmento de código que muestra el archivo `carexample4.py` con el método `__str__` :

```
#carexample4.py
Coche de clase:
c_milla_unidades = "Mi"
def __init__(uno mismo, color, millas):
    self.i_color = color
    self.i_mileage = millas

def __str__(uno mismo):
    devuelve f"car con color {self.i_color} y \
kilometraje {self.i_mileage}"

si __nombre__ == "__principal__":
    coche = Coche ("azul", 1000)
imprimir (coche)
```

Y la salida de la consola de la declaración de impresión se muestra aquí:

```
coche con color azul y kilometraje 1000
```

Con una implementación adecuada de `__str__` , podemos usar una declaración de impresión sin implementar funciones especiales como `to_string()`. Es la forma Pythonic de controlar la conversión de cadenas. Otro método popular usado por razones similares es `__repr__`, que es usado por un intérprete de Python para inspeccionar un objeto. El método `__repr__` es más para fines de depuración.

Estos métodos (y algunos más) se denominan métodos especiales o **dunders**, ya que siempre comienzan y terminan con guiones bajos dobles. Los métodos normales no deberían usar esta convención.

Estos métodos también se conocen como **métodos** mágicos en alguna literatura, pero no es la terminología oficial. Hay varias docenas de métodos especiales disponibles para implementar con una clase. Una lista completa de métodos especiales está disponible con la documentación oficial de Python 3 en <https://docs.python.org/3/reference/datamodel.html#nombresspeciales>.

Revisamos las clases y los objetos con ejemplos de código en esta sección. En la siguiente sección, estudiaremos diferentes principios orientados a objetos disponibles en Python.

Comprender los principios de la programación orientada a objetos

OOP es una forma de agrupar propiedades y comportamiento en una sola entidad, a la que llamamos objetos. Para hacer que esta agrupación sea más eficiente y modular, hay varios principios disponibles en Python, que se describen a continuación:

- Encapsulación de datos
- Herencia
- Polimorfismo
- Abstracción

En las siguientes subsecciones, estudiaremos cada uno de estos principios en detalle.

Encapsulación de datos

La encapsulación es un concepto fundamental en OOP y, a veces, también se denomina abstracción. Pero en realidad, la encapsulación es más que la abstracción. En OOP, la agrupación de datos y las acciones asociadas con los datos en una sola unidad se conoce como encapsulación. La encapsulación es en realidad más que simplemente agrupar datos y las acciones asociadas. Podemos enumerar tres objetivos principales de la encapsulación aquí, de la siguiente manera:

- Englobar datos y acciones asociadas en una sola unidad.
- Ocultar la estructura interna y los detalles de implementación del objeto.
- Restringir el acceso a ciertos componentes (atributos o métodos) del objeto.

La encapsulación simplifica el uso de los objetos sin conocer los detalles internos sobre cómo se implementa, y también ayuda a controlar las actualizaciones del estado del objeto.

En las siguientes subsecciones, discutiremos estos objetivos en detalle.

Abarcando datos y acciones.

Para abarcar datos y acciones en un inicio, definimos atributos y métodos en una clase.

Una clase en Python puede tener los siguientes tipos de elementos:

- Constructor y destructor
- Métodos y atributos de clase
- Métodos y atributos de instancia
- Clases **anidadas**

86 Programación Python avanzada orientada a objetos

Ya hemos discutido estos elementos de clase en la sección anterior, excepto las clases anidadas o **internas**. Ya proporcionamos los ejemplos de código de Python para ilustrar la implementación de constructores y destructores. Hemos utilizado atributos de instancia para encapsular datos en nuestras instancias u objetos. También hemos discutido los métodos de clase, los métodos estáticos y los atributos de clase con ejemplos de código en la sección anterior.

Para completar el tema, analizaremos el siguiente fragmento de código de Python con una clase anidada. Tomemos un ejemplo de nuestra clase Car y una clase interna Engine dentro de ella. Cada automóvil necesita un motor, por lo que tiene sentido convertirlo en una clase anidada o interna:

```
#carwithinnerexample1.py
Coche de clase:
"""
clase exterior"""

c_millas_unidades = "Mi" def
__init__(self, color, millas, tamaño_ing): self.i_color = color self.i_mileage
= millas self.i_engine = self.Engine(eng_size)

def __str__(self): devuelve
f"coche con color {self.i_color}, kilometraje \ {self.i_mileage} y motor de {self.i_engine}"

motor de clase:
"""
clase interna"""

def __init__(self, tamaño): self.i_size = tamaño

def __str__(self): return self.i_size

if __name__ == "__main__": coche =
Coche ("azul", 1000, "2.5L")
imprimir (coche)
imprimir (coche.i_motor.i_tamaño)
```

En este ejemplo, definimos una clase interna Engine dentro de nuestra clase Car normal . La clase Engine tiene solo un atributo: `i_size`, el método constructor (`__init__`) y el método `__str__` . Para la clase Car , actualizamos lo siguiente en comparación con nuestros ejemplos anteriores:

- El método `__init__` incluye un nuevo atributo para el tamaño del motor y se agregó una nueva línea para crear una nueva instancia de Engine asociada con la instancia de Car .
- El método `__str__` de la clase Car incluye los atributos de la clase interna `i_size` en eso.

El programa principal utiliza una declaración de impresión en la instancia de Car y también tiene una línea para imprimir el valor del atributo `i_size` de la clase Engine . La salida de la consola de este programa será similar a la que se muestra aquí:

```
auto con color azul, kilometraje 1000 y motor de 2.5L
```

```
2.5L
```

La salida de la consola del programa principal muestra que tenemos acceso a la clase interna desde dentro de la implementación de la clase y podemos acceder a los atributos de la clase interna desde el exterior.

En la siguiente subsección, discutiremos cómo podemos ocultar algunos de los atributos y métodos para que no sean accesibles o visibles desde fuera de la clase.

ocultar información

Hemos visto en nuestros ejemplos de código anteriores que tenemos acceso a todos los atributos de nivel de clase y de instancia sin ninguna restricción. Tal enfoque nos llevó a un diseño plano, y la clase simplemente se convertirá en un envoltorio alrededor de las variables y los métodos. Un mejor enfoque de diseño orientado a objetos es ocultar algunos de los atributos de la instancia y hacer que solo los atributos necesarios sean visibles para el mundo exterior. Para analizar cómo se logra esto en Python, presentamos dos términos: **privado** y **protegido**.

Variables y métodos privados

Una **variable** o atributo privado se puede definir utilizando un guion bajo doble como prefijo antes del nombre de una variable. En Python, no existe una palabra clave como `private`, como ocurre en otros lenguajes de programación. Tanto las variables de clase como las de instancia se pueden marcar como privadas.

También se puede definir un **método** privado usando un guion bajo doble antes del nombre de un método. Un método privado solo se puede llamar dentro de la clase y no está disponible fuera de la clase.

88 Programación Python avanzada orientada a objetos

Cada vez que definimos un atributo o un método como privado, el intérprete de Python no permite el acceso a dicho atributo o método fuera de la definición de la clase. La restricción también se aplica a las subclases; por lo tanto, solo el código dentro de una clase puede acceder a dichos atributos y métodos.

Variables y métodos protegidos

Una variable **protegida** o un método se pueden marcar agregando un solo guion bajo antes del nombre del atributo o del método. El código escrito dentro de la definición de clase y dentro de las subclases debe acceder a una variable o método protegido o utilizarlo; por ejemplo, si queremos convertir el atributo `i_color` de un atributo público a uno protegido, solo tenemos que cambiar su nombre a `_i_color`. . El intérprete de Python no impone este uso de los elementos protegidos dentro de una clase o subclase. Es más respetar la convención de nomenclatura y usar o acceder al atributo o métodos según la definición de las variables y métodos protegidos.

Mediante el uso de variables y métodos privados y protegidos, podemos ocultar algunos de los detalles de la implementación de un objeto. Esto es útil, ya que nos permite tener un código fuente limpio y ajustado dentro de una clase de gran tamaño sin exponer todo al mundo exterior. Otro motivo para ocultar atributos es controlar la forma en que se puede acceder a ellos o actualizarlos. Este es un tema para la siguiente subsección. Para concluir esta sección, discutiremos una versión actualizada de nuestra clase Car con variables privadas y protegidas y un método privado, que se muestra a continuación:

```
#carexample5.py
Coche de clase:
c_milla_unidades = "Mi"
__máx_velocidad = 200

def __init__(yo, color, millas, modelo):
    self.i_color = color
    self.i_mileage = millas
    self.__no_doors = 4
    self.__modelo = modelo

def __str__(uno mismo):
    devolver f"coche con color {self.i_color}, kilometraje
    {self.i_mileage}, modelo {self.__model} y puertas
    {auto.__puertas()}"
```

```
def __puertas(uno mismo):
    volver self.__no_doors

    si __nombre__ == "__principal__":
        coche = Coche ("azul", 1000, "Camry")
        imprimir (coche)
```

En esta clase de automóvil actualizada , hemos actualizado o agregado lo siguiente según el ejemplo anterior:

- Una variable de clase privada __max_speed con un valor predeterminado
- Una variable de instancia privada __no_doors con un valor predeterminado dentro de __init__ método constructor
- Una variable de instancia protegida __model , agregada solo con fines ilustrativos
- Un método de instancia privada __doors() para obtener el número de puertas
- El método __str__ se actualiza para obtener la puerta usando __doors() método privado

La salida de la consola de este programa funciona como se esperaba, pero si intentamos acceder a cualquiera de los métodos privados o variables privadas del programa principal, no está disponible y el El intérprete de Python arrojará un error. Esto es según el diseño, ya que el propósito previsto de estas variables privadas y métodos privados es que solo estén disponibles dentro de una clase.

Nota IMPORTANTE

Python realmente no hace que las variables y los métodos sean privados, pero pretende hacerlos privados. Python en realidad mezcla los nombres de las variables con el nombre de la clase para que no sean fácilmente visibles fuera de la clase que los contiene.

Para el ejemplo de la clase Car , podemos acceder a las variables privadas y métodos privados. Python proporciona acceso a estos atributos y métodos fuera de la definición de la clase con un nombre de atributo diferente que se compone de un guión bajo seguido del nombre de la clase y luego un nombre de atributo privado. De la misma manera, también podemos acceder a los métodos privados.

90 Programación Python avanzada orientada a objetos

Las siguientes líneas de códigos son válidas pero no recomendadas y están en contra de la definición de privado y protegido:

```
imprimir (Car._Car__max_speed)  
imprimir (coche._Car__puertas())  
imprimir (coche._modelo)
```

Como podemos ver, `_Car` se agrega antes del nombre de la variable privada real. Esto se hace para minimizar los conflictos con las variables en las clases internas también.

Protegiendo los datos

Hemos visto en nuestros ejemplos de código anteriores que podemos acceder a los atributos de la instancia sin ninguna restricción. También implementamos métodos de instancia y no tenemos restricciones en el uso de estos. Emulamos para definirlos como privados o protegidos, lo que funciona para ocultar los datos y las acciones del mundo exterior.

Pero en los problemas del mundo real, necesitamos brindar acceso a las variables de una manera que sea controlable y fácil de mantener. Esto se logra en muchos lenguajes orientados a objetos a través de **modificadores de acceso** como getters y setters, que se definen a continuación:

- **Getters:** Estos son métodos utilizados para acceder a los atributos privados de una clase o su instancia
- **Setters:** Son métodos utilizados para establecer los atributos privados de una clase o su instancia.

Los métodos getters y setters también se pueden usar para implementar una lógica adicional de acceso o configuración de los atributos, y es conveniente mantener dicha lógica adicional en un solo lugar. Hay dos formas de implementar los métodos getters y setters: una forma tradicional y una forma decorativa.

Uso de getters y setters tradicionales

Tradicionalmente, escribimos los métodos de instancia con un prefijo `get` y `set`, seguido del guion bajo y el nombre de la variable. Podemos transformar nuestra clase `Car` para usar los métodos `getter` y `setter` para atributos de instancia, de la siguiente manera:

```
#carexample6.py  
Coche de clase:  
__unidades_de_kilometraje = "Mi"  
def __init__(self, col, mil):
```

```
self.__color = col self.__kilometraje = mil

def __str__(self): devuelve f"coche
con color {self.get_color()} y \ kilometraje {self.get_mileage()}"


def get_color(self): return self.__color


def get_mileage(self): return self.__mileage


def set_mileage (self, new_mil): self.__mileage = new_mil


si __nombre__ == "__principal__":
    coche = Coche ("azul", 1000)

imprimir (automóvil)
imprimir (automóvil.obtener_color())
imprimir(automóvil.obtener_kilómetro())
auto.establecer_kilómetro(2000)
imprimir (automóvil.obtener_color())
imprimir(automóvil.obtener_kilómetro())
```

En esta clase de automóvil actualizada , agregamos lo siguiente:

- Los atributos de instancia de color y kilometraje se agregaron como variables privadas. • Métodos getter para atributos de instancia de color y kilometraje .
- Un método de establecimiento solo para el atributo de millaje porque el color generalmente no cambia una vez que se establece en el momento de la creación del objeto.
- En el programa principal, obtenemos datos para la instancia recién creada de la clase utilizando métodos getter. A continuación, actualizamos el kilometraje utilizando un método de establecimiento y luego obtuvimos datos nuevamente para los atributos de color y kilometraje .

92 Programación Python avanzada orientada a objetos

La salida de la consola de cada declaración en este ejemplo es trivial y según las expectativas. Como se mencionó, no definimos un setter para cada atributo, sino solo para aquellos atributos donde tiene sentido y lo exige el diseño. Usar getters y setters es una buena práctica en OOP, pero no son muy populares en Python. La cultura de los desarrolladores de Python (también conocida como la forma Pythonic) sigue siendo acceder a los atributos directamente.

Uso de decoradores de propiedades

Usar un **decorador** para definir getters y setters es un enfoque moderno que ayuda a lograr la forma de programación de Python.

Si le gusta usar decoradores, entonces tenemos un decorador `@property` en Python para hacer que el código sea más simple y limpio. La clase Car con getters y setters tradicionales se actualiza con decoradores, y aquí hay un fragmento de código que muestra esto:

```
ejemplo7.py
Coche de clase:
__unidades_de_kilometraje = "Mi"

def __init__(self, col, mil):
    self.__color = col
    auto.__kilometraje = mil

def __str__(uno mismo):
    devolver f"coche con color {self.color} y kilometraje \
{auto.kilometraje}"

@propiedad
color def (uno mismo):
    volver self.__color

@propiedad
def kilometraje(auto):
    devolver auto.__kilometraje

@mileage.setter
```

```
def kilometraje (self, new_mil):
    self.__kilometraje = new_mil

    if __nombre__ == "__principal__":
        coche = Coche ("azul", 1000)

        imprimir (coche)
        imprimir (coche.color)
imprimir (coche.kilometraje)
        coche.kilometraje = 2000
        imprimir (coche.color)
imprimir (coche.kilometraje)
```

En esta definición de clase actualizada, actualizamos o agregamos lo siguiente:

- Atributos de instancia como variables privadas
- Métodos Getter para el color y el kilometraje usando el nombre del atributo como nombre del método y usando @property
- Métodos de establecimiento para el kilometraje usando el decorador @mileage.setter , dándole al método el mismo nombre que el nombre del atributo

En el script principal, accedemos a los atributos de color y kilometraje usando el nombre de la instancia seguido de un punto y el nombre del atributo (la forma Pythonic). Esto hace que la sintaxis del código sea concisa y legible. El uso de decoradores también simplifica el nombre de los métodos.

En conclusión, discutimos todos los aspectos de la encapsulación en Python, el uso de clases para agrupar datos y acciones, ocultar información innecesaria del mundo exterior de una clase y cómo proteger los datos en una clase usando getters, setters y características de propiedad de Pitón. En la siguiente sección, discutiremos cómo se implementa la herencia en Python.

Ampliación de clases con herencia

El concepto de herencia en OOP es similar al concepto de herencia en el mundo real, donde los niños heredan algunas de las características de sus padres además de sus propias características.

94 Programación Python avanzada orientada a objetos

De manera similar, una clase puede heredar elementos de otra clase. Estos elementos incluyen atributos y métodos. La clase de la que heredamos otra clase se conoce comúnmente como clase padre, **superclase** o clase **base**. La clase que heredamos de otra clase se denomina **clase derivada**, **clase secundaria** o **subclase**. La siguiente captura de pantalla muestra una relación simple entre una clase principal y una clase secundaria:



Figura 3.1 – Relación de clase padre-hijo

En Python, cuando una clase hereda de otra clase, normalmente hereda todos los elementos que componen la clase principal, pero esto se puede controlar usando convenciones de nomenclatura (como el doble guión bajo) y modificadores de acceso.

La herencia puede ser de dos tipos: **simple** o **múltiple**. Discutiremos estos en el próximas secciones.

herencia simple

En herencia simple o básica, una clase se deriva de un solo parent. Esta es una forma de herencia de uso común en OOP y está más cerca del árbol genealógico de los seres humanos. La sintaxis de una clase principal y una clase secundaria usando herencia simple se muestra a continuación:

```

clase ClaseBase:
    <atributos y métodos de la clase base>

clase ChildClass (ClaseBase):
    <atributos y métodos de la clase secundaria>
  
```

Para esta herencia simple, modificaremos nuestro ejemplo de la clase Car para que se derive de una clase parent Vehicle. También agregaremos una clase secundaria Camión para elaborar el concepto de herencia. Aquí está el código con modificaciones:

```

#herencia1.py
clase de vehículo:
    def __init__(uno mismo, color):
        self.i_color = color

    def print_vehicle_info(self):
  
```

```
print(f"Este es un vehículo y sé que mi color es \ {self.i_color}")
```

clase Coche (Vehículo):

```
def __init__(self, color, asientos): self.i_color = color self.i_seats  
= asientos
```

```
def print_me(self): print( f"Auto con  
color {self.i_color} y no de \ asientos {self.i_seats}")
```

clase Camión (Vehículo):

```
def __init__(self, color, capacidad): self.i_color = color self.i_capacity  
= capacidad
```

```
def print_me(self): print( f"Camión con  
color {self.i_color} y \ capacidad de carga {self.i_capacity} toneladas")
```

```
if __nombre__ == "__principal__": coche =  
Coche ("azul", 5)  
coche.print_vehicle_info() coche.print_me()  
camión = Camión("blanco", 1000)  
  
camión.print_vehicle_info() camión.print_me()
```

En este ejemplo, creamos una clase principal Vehicle con un atributo i_color y un método print_vehicle_info . Ambos elementos son candidatos a la herencia.

A continuación, creamos dos clases secundarias, Auto y Camión. Cada clase secundaria tiene un atributo adicional (i_seats e i_capacity) y un método adicional (print_me). En los métodos print_me de cada clase secundaria, accedemos al atributo de la instancia de la clase principal, así como a los atributos de la instancia de la clase secundaria.

96 Programación Python avanzada orientada a objetos

Este diseño fue intencional, para elaborar la idea de heredar algunos elementos de la clase principal y agregar algunos elementos propios en una clase secundaria. Las dos clases secundarias se utilizan en este ejemplo para demostrar el papel de la herencia hacia la reutilización.

En nuestro programa principal, creamos instancias de Car and Truck e intentamos acceder al método principal así como al método de instancia. La salida de la consola de este programa es la esperada y se muestra a continuación:

```
Este es un vehículo y sé que mi color es azul.  
Coche con color azul y no de asientos 5  
Este es un vehículo y sé que mi color es blanco.  
Camión con color blanco y capacidad de carga 1000 toneladas
```

Herencia múltiple

En la herencia múltiple, una clase secundaria se puede derivar de varios padres. El concepto de herencia múltiple es aplicable en diseños avanzados orientados a objetos donde los objetos tienen relaciones con múltiples objetos, pero debemos tener cuidado al heredar de múltiples clases, especialmente si esas clases se heredan de una superclase común.

Esto nos puede llevar a problemas como el problema del diamante. El problema del diamante es una situación en la que creamos una clase X al heredar de dos clases, Y y Z, y las clases Y y Z se heredan de una clase común, A. La clase X tendrá ambigüedad sobre el código común de la clase A. class, que hereda de las clases Y y Z. No se recomienda la herencia múltiple debido a los posibles problemas que puede traer consigo.

Para ilustrar el concepto, modificaremos nuestras clases Vehicle y Car y agregaremos una clase Engine como uno de los padres. El código completo con herencia múltiple de clases se muestra en el siguiente fragmento:

```
#herencia2.py  
clase de vehículo:  
  
def __init__(uno mismo, color):  
    self.i_color = color  
  
def print_vehicle_info(self):  
    print(f"Este es un vehículo y sé que mi color es \\\n{self.i_color}")  
  
motor de clase:  
def __init__(uno mismo, tamaño):
```

```
self.i_size = tamaño

def print_engine_info(self): print(f"Este es
Engine y sé que mi tamaño es \ {self.i_size}")

clase Coche (Vehículo, Motor): def
__init__(self, color, tamaño, asiento): self.i_color = color
self.i_size = tamaño self.i_seat = asiento

def print_car_info(self): print(f"Este auto
de color {self.i_color} con \ asientos {self.i_seat} con motor de tamaño \ {self.i_size}")

if __name__ == "__main__": coche =
Coche ("azul", "2.5L", 5 )
coche.print_vehicle_info()
coche.print_engine_info()
coche.print_car_info()
```

En este ejemplo de herencia múltiple, creamos dos clases principales como principal: Vehículo y Motor. La clase principal Vehicle es la misma que en el ejemplo anterior. La clase Engine tiene un atributo (*i_size*) y un método (*print_engine_info*). La clase Car se deriva tanto de Vehicle como de Engine y agrega un atributo adicional (*i_seats*) y un método adicional (*print_car_info*). En el método de instancia, podemos acceder a los atributos de instancia de ambas clases principales.

En el programa principal, creamos una instancia de la clase Car . Con esta instancia, podemos acceder a los métodos de instancia de las clases principales, así como a las clases secundarias. La salida de la consola del programa principal se muestra aquí y es como se esperaba:

```
Este es un vehículo y sé que mi color es azul.
Coche con color azul y no de asientos 5
Este es un vehículo y sé que mi color es blanco.
Camión con color blanco y capacidad de carga 1000 toneladas
```

98 Programación Python avanzada orientada a objetos

En esta sección, presentamos la herencia y sus tipos como simple y múltiple. A continuación, estudiaremos el concepto de polimorfismo en Python.

Polimorfismo

En su significado literal, un proceso de tener múltiples formas se llama polimorfismo. En OOP, el **polimorfismo** es la capacidad de una instancia de comportarse de múltiples maneras y una forma de usar el mismo método con el mismo nombre y los mismos argumentos, para comportarse de manera diferente de acuerdo con la clase a la que pertenece.

El polimorfismo se puede implementar de dos maneras: **sobrecarga** de métodos y **método primordial**. Discutiremos cada uno en las siguientes subsecciones.

Sobrecarga de métodos

La sobrecarga de métodos es una forma de lograr el polimorfismo al tener varios métodos con el mismo nombre, pero con un tipo o número de argumentos diferente. No existe una forma limpia de implementar la sobrecarga de métodos en Python. Dos métodos no pueden tener el mismo nombre en Python. En Python, todo es un objeto, incluidas las clases y los métodos. Cuando escribimos métodos para una clase, de hecho son atributos de una clase desde la perspectiva del espacio de nombres y, por lo tanto, no pueden tener el mismo nombre. Si escribimos dos métodos con el mismo nombre, no habrá ningún error de sintaxis, y el segundo simplemente reemplazará al primero.

Dentro de una clase, un método puede sobrecargarse estableciendo el valor predeterminado para los argumentos. Esta no es la forma perfecta de implementar la sobrecarga de métodos, pero funciona. Aquí hay un ejemplo de sobrecarga de métodos dentro de una clase en Python:

```
#métodosobrecarga1.py
Coche de clase:
def __init__(auto, color, asientos):
    self.i_color = color
    self.i_seat = asientos

def print_me(self, i='básico'):
    if(i =='básico'):
        print(f"Este carro es de color {self.i_color}")
    demás:
        print(f"Este auto es de color {self.i_color} \\"
```

```
con asientos {self.i_seat})"  
  
si __nombre__ == "__principal__":  
    coche = Coche("azul", 5 )  
    coche.print_me()  
    coche.print_me('bla')  
    coche.print_me('detalle')
```

En este ejemplo, agregamos un método print_me con un argumento que tiene un valor predeterminado. El valor predeterminado se utilizará cuando no se pase ningún parámetro. Cuando no se pasa ningún parámetro al método print_me , la salida de la consola solo proporcionará el color de la instancia de Car . Cuando se pasa un argumento a este método (independientemente del valor), tenemos un comportamiento diferente de este método, que proporciona tanto el color como el número de asientos de la instancia de Car . Aquí está la salida de la consola de este programa como referencia:

```
Este carro es de color azul.  
Este carro es de color azul con asientos 5  
Este carro es de color azul con asientos 5
```

Nota IMPORTANTE

Hay bibliotecas de terceros (por ejemplo, sobrecarga) disponibles que se pueden usar para implementar la sobrecarga de métodos de una manera más limpia.

Anulación de método

Tener el mismo nombre de método en una clase secundaria que en una clase principal se conoce como anulación de métodos. Se espera que la implementación de un método en una clase principal y una clase secundaria sea diferente. Cuando llamamos a un método de anulación en una instancia de una clase secundaria, el intérprete de Python busca el método en la definición de la clase secundaria, que es el método anulado. El intérprete ejecuta el método de nivel de clase hijo. Si el intérprete no encuentra un método a nivel de instancia secundaria, lo busca en una clase principal. Si tenemos que ejecutar específicamente un método en una clase principal que se anula en una clase secundaria usando la instancia de la clase secundaria, podemos usar el método super() para acceder al método de nivel de clase principal. Este es un concepto de polimorfismo más popular en Python, ya que va de la mano con la herencia y es una de las formas poderosas de implementar la herencia.

100 programación Python avanzada orientada a objetos

Para ilustrar cómo implementar la anulación de métodos, actualizaremos inheritance1.py cambiando el nombre del método print_vehicle_info como print_me.

Como sabemos, los métodos print_me ya están en las dos clases secundarias con diferentes implementaciones. Aquí está el código actualizado con los cambios resaltados:

```
#methodoverriding1.py
clase de vehículo:
def __init__(self, color): self.i_color = color

def print_me(self): print(f"Este es un
vehículo y sé que mi color es \ {self.i_color}")

clase Coche (Vehículo):
def __init__(self, color, asientos): self.i_color = color self.i_seats
= asientos

def print_me(self): print( f"Auto con
color {self.i_color} y no de \ asientos {self.i_seats}")

clase Camión (Vehículo):
def __init__(self, color, capacidad): self.i_color = color self.i_capacity
= capacidad

def print_me(self): print( f"Camión
con color {self.i_color} y \ capacidad de carga {self.i_capacity} toneladas")

if __nombre__ == "__principal__": vehículo
= Vehículo("rojo")
vehículo.print_me() coche = Coche
("azul", 5)
coche.print_me()
```

```
camion = Camion("blanco", 1000)  
camion.print_me()
```

En este ejemplo, anulamos el método print_me en las clases secundarias. Cuando creamos tres instancias diferentes de las clases Vehicle, Car y Truck y ejecutamos el mismo método, obtenemos un comportamiento diferente. Aquí está la salida de la consola como referencia:

```
Este es un vehículo y sé que mi color es rojo.  
Coche con color azul y no de asientos 5  
Camión con color blanco y capacidad de carga 1000 toneladas
```

La anulación de métodos tiene muchas aplicaciones prácticas en problemas del mundo real; por ejemplo, podemos heredar la clase de lista integrada y anular sus métodos para agregar nuestra funcionalidad.

La introducción de un enfoque de ordenación personalizado es un ejemplo de anulación de métodos para un objeto de lista . Cubriremos algunos ejemplos de anulación de métodos en los próximos capítulos.

Abstracción

La abstracción es otra característica poderosa de OOP y se relaciona principalmente con ocultar los detalles de la implementación y mostrar solo las características esenciales o de alto nivel de un objeto.

Un ejemplo del mundo real es un automóvil que derivamos con las funciones principales disponibles para nosotros como conductor, sin conocer los detalles reales de cómo funciona la función y qué otros objetos están involucrados para proporcionar estas funciones.

La abstracción es un concepto que está relacionado con la encapsulación y la herencia juntas, y es por eso que hemos mantenido este tema hasta el final para comprender primero la encapsulación y la herencia. Otra razón para tener esto como un tema aparte es enfatizar el uso de clases abstractas en Python.

Clases abstractas en Python

Una clase abstracta actúa como un modelo para otras clases. Una clase abstracta le permite crear un conjunto de métodos abstractos (vacíos) que serán implementados por una clase secundaria. En términos simples, una clase que contiene uno o más métodos abstractos se denomina **clase abstracta**. Por otro lado, un **método abstracto** es aquel que solo tiene una declaración pero no una implementación.

Puede haber métodos en una clase abstracta que ya estén implementados y que puedan ser aprovechados por una clase secundaria (tal cual) mediante la herencia. El concepto de clases abstractas es útil para implementar interfaces comunes como **las interfaces de programación de aplicaciones (API)** y también para definir una base de código común en un lugar que las clases secundarias pueden reutilizar.

102 Programación Python avanzada orientada a objetos

Consejo

Las clases abstractas no se pueden instanciar.

Se puede implementar una clase abstracta utilizando un módulo integrado de Python llamado **Clases base abstractas (ABC)** del paquete abc . El paquete abc también incluye el módulo Abstractmethod , que utiliza decoradores para declarar los métodos abstractos.

A continuación se muestra un ejemplo simple de Python con el uso del módulo ABC y el decorador de método abstracto :

```
#abstracción1.py
de abc importar ABC, método abstracto

clase de vehículo (ABC):
    def hola(yo):
        print(f"Hola desde la clase abstracta")
    @metodoabstracto
    def print_me(auto):
        aprobar

    clase Coche (Vehículo):
        def __init__(self, color, asientos): self.i_color = color self.i_seats =
            asientos

        """Se debe implementar este método"""\n        def print_me(self): print( f"Auto con color
            {self.i_color} y no de \ asientos {self.i_seats}" )

    if __name__ == "__main__": # vehículo =
        Vehículo() #no es posible # vehículo.hola()

    coche = Coche ("azul", 5)
    coche.print_me()
    coche.hola()
```

En este ejemplo, hicimos lo siguiente:

- Hicimos que la clase Vehicle sea abstracta heredándola de la clase ABC y también declarando uno de los métodos (print_me) como un método abstracto. Usamos el decorador @abstractmethod para declarar un método abstracto.
- A continuación, actualizamos nuestra famosa clase Car implementando el método print_me y manteniendo el resto del código igual que en el ejemplo anterior.
- En la parte principal del programa, intentamos crear una instancia del Clase de vehículo (código comentado en la ilustración). Creamos una instancia de la clase Car y ejecutamos los métodos print_me y hello .

Cuando intentamos crear una instancia de la clase Vehicle , nos da un error como este:

No se puede crear una instancia de clase abstracta Vehículo con métodos abstractos
imprimir

Además, si intentamos no implementar el método print_me en la clase secundaria Car , obtenemos un error. Para una instancia de la clase Car , obtenemos la salida de consola esperada de los métodos print_me y hello .

Usar la composición como un enfoque de diseño alternativo

La composición es otro concepto popular en OOP que nuevamente es algo relevante para la encapsulación. En palabras simples, composición significa incluir uno o más objetos dentro de un objeto para formar un objeto del mundo real. Una clase que incluye otros objetos de clase se denomina clase **compuesta** , y las clases cuyos objetos se incluyen en una clase compuesta se conocen como clases de **componentes** . En la siguiente captura de pantalla, mostramos un ejemplo de una clase compuesta que tiene tres objetos de clase de componente, A, B y C:

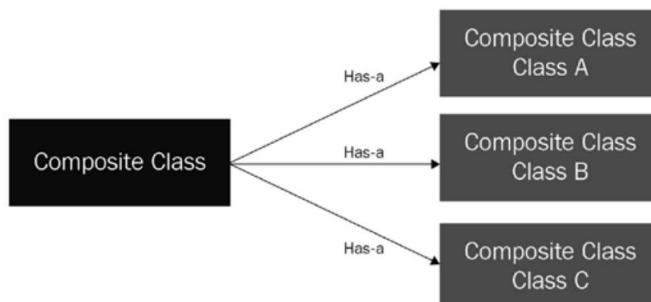


Figura 3.2 – Relación entre una clase compuesta y sus clases componentes

104 Programación Python avanzada orientada a objetos

La composición se considera un enfoque alternativo a la herencia. Ambos enfoques de diseño están destinados a establecer una relación entre los objetos. En el caso de la herencia, los objetos están estrechamente acoplados porque cualquier cambio en las clases principales puede romper el código de las clases secundarias. Por otro lado, los objetos están débilmente acoplados en el caso de la composición, lo que facilita cambios en una clase sin romper nuestro código en otra clase. Debido a la flexibilidad, el enfoque de composición es bastante popular, pero esto no significa que sea la elección correcta para cada problema. Entonces, ¿cómo podemos determinar cuál usar para qué problema? Hay una regla general para esto. Cuando tenemos una relación entre objetos, la herencia es la elección correcta; por ejemplo, un automóvil es un vehículo y un gato es un animal. En el caso de la herencia, una clase secundaria es una extensión de una clase principal, con funcionalidad adicional y la capacidad de reutilizar la funcionalidad de la clase principal. Si la relación entre los objetos es que un objeto tiene otro objeto, entonces es mejor usar la composición; por ejemplo, un automóvil tiene una batería.

Tomaremos nuestro ejemplo anterior de la clase Car y la clase Engine . En el código de ejemplo para herencia múltiple, implementamos la clase Car como un elemento secundario de Engine class, que no es realmente un buen caso de uso de la herencia. Es hora de usar la composición implementando la clase Car con el objeto Engine dentro de la clase Car . Podemos tener otra clase para Seat y también podemos incluirla dentro de la clase Car .

Ilustraremos más este concepto en el siguiente ejemplo, en el que construimos un automóvil class al incluir las clases Engine y Seat en él:

```
#composición1.py
Asiento de clase:
def __init__(uno mismo, tipo):
    self.i_type = tipo

def __str__(uno mismo):
    return f"Tipo de asiento: {self.i_type}"

motor de clase:
def __init__(uno mismo, tamaño):
    self.i_size = tamaño

def __str__(uno mismo):
    volver f"Motor: {self.i_size}"

Coche de clase:
def __init__(self, color, ing_size, seat_type):
```

```
self.i_color = color self.engine =
Engine(eng_size) self.seat = Seat(seat_type)

def print_me(self): print(f"Este auto
de color {self.i_color} con \ {self.engine} y {self.seat}")

si __nombre__ == "__principal__":
    coche = Coche ("azul", "2.5L", "cuero")
    car.print_me() print(car.motor)
    print(car.seat) print(car.i_color)
    print(car.engine.i_size)
    print(car.seat.i_type)
```

Podemos analizar este código de ejemplo de la siguiente manera:

1. Definimos clases de Motor y Asiento con un atributo en cada clase: i_size para la clase Engine y i_type para la clase Seat .
2. Más tarde, definimos una clase de automóvil agregando el atributo i_color , un motor instancia, y una instancia de Seat en ella. Las instancias de Engine y Seat se crearon en el momento de crear una instancia de Car .
3. En este programa principal, creamos una instancia de Car y realizamos el siguientes acciones:
 - a) car.print_me: Esto accede al método print_me en la instancia de Car . b) print(car.engine): Esto ejecuta el método __str__ de la clase Engine .
 - c) print(car.seat): Esto ejecuta el método __str__ de la clase Seat . d) print(car.i_color): Accede al atributo i_color del Instancia de coche .
 - e) print(car.engine.i_size): Esto accede al atributo i_size del Instancia de motor dentro de la instancia de automóvil .
 - f) print(car.seat.i_type): Accede al atributo i_type del Seat instancia dentro de la instancia de Car

106 Programación Python avanzada orientada a objetos

La salida de la consola de este programa se muestra aquí:

```
Este auto de color azul con Motor: 2.5L y Tipo de asiento: cuero
```

```
Motor: 2.5L
```

```
Tipo de asiento: cuero
```

```
azul
```

```
2.5L
```

```
cuero
```

A continuación, analizaremos la tipificación de pato, que es una alternativa al polimorfismo.

Introducción a la escritura de patos en Python

La **tipificación pato**, a veces denominada **tipificación dinámica**, se adopta principalmente en lenguajes de programación que admiten la tipificación dinámica, como Python y JavaScript. El nombre de escritura de pato se toma prestado en función de la siguiente cita:

"Si parece un pato, nada como un pato y grazna como un pato, entonces probablemente sea un pato".

Esto significa que si un pájaro se comporta como un pato, probablemente será un pato. El punto de mencionar esta cita es que es posible identificar un objeto por su comportamiento, que es el principio central de la tipificación pato en Python.

En el tipo de pato, el tipo de clase de un objeto es menos importante que el método (comportamiento) que define. Usando el tipo de pato, los tipos del objeto no se verifican, pero se ejecuta el método que se espera.

Para ilustrar este concepto, tomamos un ejemplo simple con tres clases, Car, Cycle y Horse, e intentamos implementar un método de inicio en cada una de ellas. En la clase Horse , en lugar de nombrar el método start, lo llamamos push. Aquí hay un fragmento de código con las tres clases y el programa principal al final:

```
#ducttype1.py
```

```
Coche de clase:
```

```
def inicio(auto):
```

```
    imprimir ("arrancar el motor por encendido/batería")
```

```
Ciclo de clase:
```

```
def inicio(auto):
```

```
    imprimir ("comenzar empujando paletas")
```

```
clase de caballo:  
def empujar(auto):  
    imprimir ("comenzar tirando/soltando las riendas")  
  
      
  
    si __nombre__ == "__principal__":  
        para obj en Car(), Cycle(), Horse():  
            obj.start()
```

En el programa principal, tratamos de iterar las instancias de estas clases dinámicamente y llamamos al método de inicio . Como era de esperar, la línea obj.start() falló para el objeto Horse porque la clase no tiene dicho método. Como podemos ver en este ejemplo, podemos poner diferentes clases o tipos de instancias en una instrucción y ejecutar los métodos a través de ellas.

Si cambiamos el método llamado push para comenzar dentro de la clase Horse , el programa principal se ejecutará sin ningún error. La escritura de pato tiene muchos casos de uso, donde simplifica las soluciones. El uso del método len en muchos objetos y el uso de iteradores son un par de muchos ejemplos. Exploraremos los iteradores en detalle en el próximo capítulo.

Hasta ahora, hemos revisado diferentes conceptos y principios orientados a objetos y sus beneficios. En la siguiente sección, también discutiremos brevemente cuándo no es muy beneficioso

para usar la programación orientada a objetos.

Aprendiendo cuándo no usar OOP en Python

Python tiene la flexibilidad de desarrollar programas usando lenguajes OOP como Java o usando programación declarativa como C. OOP siempre es atractivo para los desarrolladores porque proporciona herramientas poderosas como encapsulación, abstracción, herencia y polimorfismo, pero estas herramientas pueden no encajar. cada escenario y caso de uso. Estas herramientas son más beneficiosas cuando se usan para crear una aplicación grande y compleja, especialmente una que involucra **interfaces de usuario (UI)** e interacciones de usuario.

Si su programa es más como un script que tiene que ejecutar ciertas tareas y no hay necesidad de mantener el estado de los objetos, usar OOP es una exageración. Las aplicaciones de ciencia de datos y el procesamiento intensivo de datos son ejemplos en los que es menos importante usar OOP pero más importante definir cómo ejecutar tareas en un orden determinado para lograr objetivos. Un ejemplo del mundo real es escribir programas de cliente para ejecutar trabajos de uso intensivo de datos en un grupo de nodos, como Apache Spark para procesamiento paralelo. Cubriremos este tipo de aplicaciones en capítulos posteriores. Aquí hay algunos escenarios más en los que no es necesario usar OOP:

108 Programación Python avanzada orientada a objetos

- Leer un archivo, aplicar lógica y volver a escribir en un nuevo archivo es un tipo de programa que es más fácil de implementar usando funciones en un módulo en lugar de usar OOP.
- La configuración de dispositivos usando Python es muy popular y es otro candidato para ser hecho usando funciones regulares.
- Analizar y transformar datos de un formato a otro también es un caso de uso que se puede programar mediante programación declarativa en lugar de programación orientada a objetos.
- Portar un código base antiguo a uno nuevo con OOP no es una buena idea. Debemos recordar que es posible que el código anterior no se construya utilizando patrones de diseño de programación orientada a objetos y que podemos terminar con funciones que no son de programación orientada a objetos envueltas en clases y objetos que son difíciles de mantener y ampliar.

En resumen, es importante analizar primero la declaración del problema y los requisitos antes de elegir si usar OOP o no. También depende de qué bibliotecas de terceros utilizará con su programa. Si debe ampliar las clases de bibliotecas de terceros, tendrá que aceptar OOP en ese caso.

Resumen

En este capítulo, aprendimos el concepto de clases y objetos en Python y también discutimos cómo construir clases y usarlas para crear objetos e instancias. Más tarde, profundizamos en los cuatro pilares de la programación orientada a objetos: encapsulación, herencia, polimorfismo y abstracción. También trabajamos con ejemplos de código simples y claros para que a los lectores les resulte más fácil comprender los conceptos de programación orientada a objetos. Estos cuatro pilares son fundamentales para usar OOP en Python.

En las secciones posteriores, también cubrimos el tipo de pato, que es importante para aclarar su no dependencia de las clases, antes de terminar el capítulo revisando cuándo no es significativamente beneficioso usar OOP.

Al leer este capítulo, no solo actualizó su conocimiento de los conceptos principales de OOP, sino que también aprendió cómo aplicar los conceptos usando la sintaxis de Python. Revisaremos algunas bibliotecas de Python para programación avanzada en el próximo capítulo.

Preguntas 1. ¿Qué

- son una clase y un objeto?
2. ¿Qué son los dunders?
3. ¿Python admite la herencia de una clase de varias clases?
4. ¿Podemos crear una instancia de una clase abstracta?
5. El tipo de una clase es importante en la tipificación de conductos: ¿verdadero o falso?

Otras lecturas

- Programación Modular con Python, por Erik Westra
- Programación orientada a objetos de Python 3, por Dusty Phillips •
- Programación orientada a objetos de aprendizaje, por Gaston C. Hillar •
- Python para todos: tercera edición, por Cay Horstmann y Rance Necaise

respuestas

1. Una clase es un plano o una plantilla para decirle al intérprete de Python cómo algo necesita ser definido. Un objeto es una instancia que se crea a partir de una clase basada en lo que se define en esa clase.
2. Dunders son métodos especiales que siempre comienzan y terminan con guiones bajos dobles. Hay algunas docenas de métodos especiales disponibles para implementar con cada clase.
3. Sí: Python admite la herencia de una clase de varias clases.
4. No, no podemos crear una instancia de una clase abstracta.
5. Falso. Son los métodos los que son más importantes que la clase.

Sección 2: Avanzado Programación Conceptos

Continuamos nuestro viaje aprendiendo los conceptos avanzados del lenguaje Python en esta sección. Esto incluye un repaso de algunos conceptos para usted con una introducción a temas avanzados como iteradores, generadores, errores y manejo de excepciones. Esto te ayudará a pasar al siguiente nivel de programación en Python. Además de escribir programas de Python, también exploramos cómo escribir y automatizar pruebas unitarias y pruebas de integración utilizando marcos de prueba como unittest y pytest. En la última parte de esta sección, analizamos algunos conceptos de funciones avanzadas para la transformación de datos y la construcción de decoradores en Python, y cómo usar estructuras de datos, incluidos pandas DataFrames para aplicaciones de análisis.

Esta sección contiene los siguientes capítulos:

- Capítulo 4, Bibliotecas de Python para programación avanzada
- Capítulo 5, Pruebas y automatización con Python
- Capítulo 6, Sugerencias y trucos avanzados en Python

4

Bibliotecas de Python para avanzada Programación

En capítulos anteriores, hemos discutido diferentes enfoques para construir programas modulares y reutilizables en Python. En este capítulo, investigaremos algunos conceptos avanzados del lenguaje de programación Python, como iteradores, generadores, registro y manejo de errores. Estos conceptos son importantes para escribir código eficiente y reutilizable. Para este capítulo, asumimos que está familiarizado con la sintaxis del lenguaje Python y sabe cómo escribir estructuras de control y de bucle.

En este capítulo, aprenderemos cómo funcionan los bucles en Python, cómo se manejan los archivos y cuál es la mejor práctica para abrir y acceder a los archivos, y cómo manejar situaciones erróneas, que pueden ser esperadas o inesperadas. También investigaremos el soporte de registro en Python y las diferentes formas de configurar el sistema de registro. Este capítulo también lo ayudará a aprender a usar las bibliotecas avanzadas de Python para crear proyectos complejos.

Cubriremos los siguientes temas en este capítulo:

- Introducción a los contenedores de datos de Python
- Uso de iteradores y generadores para el procesamiento de datos

114 bibliotecas de Python para programación avanzada

- Manejo de archivos en Python
- Manejo de errores y excepciones
- Uso del módulo de registro de Python

Al final de este capítulo, habrá aprendido cómo construir iteradores y generadores, cómo manejar errores y excepciones en su programa y cómo implementar el registro para su proyecto de Python de manera eficiente.

Requerimientos técnicos

El requisito técnico para este capítulo es que debe tener instalado Python 3.7 o posterior en su computadora. El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Capítulo04>.

Comencemos por actualizar nuestro conocimiento sobre los contenedores de datos disponibles en Python, que serán útiles para los temas de seguimiento de este capítulo.

Introducción a los contenedores de datos de Python

Python admite varios tipos de datos, tanto numéricos como de colecciones. La definición de tipos de datos numéricos, como números enteros y números de coma flotante, se basa en la asignación de un valor a una variable. El valor que asignamos a una variable determina el tipo de dato numérico. Tenga en cuenta que un constructor específico (por ejemplo, int() y float()) también se puede usar para crear una variable de un tipo de datos específico. Los tipos de datos de contenedor también se pueden definir mediante la asignación de valores en un formato apropiado o mediante el uso de un constructor específico para cada tipo de datos de colección. Estudiaremos cinco tipos diferentes de datos de contenedores en esta sección: **cadenas, listas, tuplas, diccionarios y conjuntos**.

Instrumentos de cuerda

Las cadenas no son directamente un tipo de datos de contenedor. Pero es importante analizar el tipo de datos de cadena debido a su amplio uso en la programación de Python y también al hecho de que el tipo de datos de cadena se implementa mediante una **secuencia inmutable** de puntos de código Unicode. El hecho de que use una secuencia (un tipo de colección) lo convierte en un candidato para ser discutido en esta sección.

Los objetos de cadena son objetos inmutables en Python. Con la inmutabilidad, los objetos de cadena brindan una solución segura para programas concurrentes donde varias funciones pueden acceder al mismo objeto de cadena y obtendrán el mismo resultado. Esta seguridad no es posible con objetos mutables. Al ser objetos inmutables, los objetos de cadena son populares para usar como claves para el tipo de datos del diccionario o como elementos de datos para el tipo de datos del conjunto. El inconveniente de la inmutabilidad es que es necesario crear una nueva instancia incluso si se va a realizar un pequeño cambio en una instancia de cadena existente.

Objetos mutables versus inmutables

Un objeto mutable se puede cambiar después de su creación, pero no es posible cambiar un objeto inmutable.

Los literales de cadena se pueden encerrar usando comillas simples coincidentes (por ejemplo, 'bla'), comillas dobles (por ejemplo, "bla, bla") o comillas simples o dobles triples (por ejemplo, """ninguno"""" o 'ninguna'). También vale la pena mencionar que los objetos de cadena se manejan de manera diferente en Python 3 que en Python 2. En Python 3, los objetos de cadena solo pueden contener secuencias de texto en forma de puntos de datos Unicode, pero en Python 2 pueden contener tanto texto como datos de bytes. . En Python 3, los datos de bytes son manejados por el tipo de datos de bytes .

Separar el texto de los bytes en Python 3 lo hace limpio y eficiente, pero a costa de la portabilidad de los datos. El texto Unicode en cadenas no se puede guardar en el disco ni enviar a una ubicación remota en la red sin convertirlo a un formato binario. Esta conversión requiere codificar los datos de la cadena en una secuencia de bytes, lo que se puede lograr de una de las siguientes maneras:

- **Usando el método str.encode (codificación, errores):** Este método está disponible en el objeto de cadena y puede tomar dos argumentos. Un usuario puede proporcionar el tipo de códec que se utilizará (UTF-8 es el predeterminado) y cómo manejar los errores.
- **Conversión al tipo de datos bytes:** un objeto de cadena se puede convertir al tipo de datos Bytes tipo de datos pasando la instancia de cadena al constructor de bytes junto con el esquema de codificación y el esquema de manejo de errores.

Los detalles de los métodos y los atributos disponibles con cualquier objeto de cadena se pueden encontrar en la documentación oficial de Python según el lanzamiento de Python.

Liza

La lista es uno de los tipos de colección básicos en Python, que se usa para almacenar múltiples objetos usando una sola variable. Las listas son dinámicas y mutables, lo que significa que los objetos de una lista se pueden cambiar y la lista puede crecer o reducirse.

116 bibliotecas de Python para programación avanzada

Los objetos de lista en Python no se implementan usando ningún concepto de lista enlazada sino usando una matriz de longitud variable. La matriz contiene referencias a los objetos que está almacenando. El puntero de esta matriz y su longitud se almacenan en la estructura del encabezado de la lista, que se mantiene actualizada a medida que se agregan o eliminan objetos de una lista. El comportamiento de una matriz de este tipo parece una lista, pero en realidad no es una lista real. Es por eso que algunas de las operaciones en una lista de Python no están optimizadas. Por ejemplo, insertar un nuevo objeto en una lista y eliminar objetos de una lista tendrá una complejidad de n .

Para salvar la situación, Python proporciona un tipo de datos deque en el módulo integrado de colecciones . El tipo de datos deque proporciona la funcionalidad de pilas y colas y es una buena opción alternativa para los casos en los que una declaración de problema exige un comportamiento similar a una lista enlazada.

Las listas se pueden crear vacías o con un valor inicial usando corchetes. A continuación, presentamos un fragmento de código que demuestra cómo crear un objeto de lista vacío o no vacío usando solo los corchetes o usando el constructor de objetos de lista:

```
e1 = [] #una lista vacía
e2 = lista() #una lista vacía a través del constructor
g1 = ['a', 'b'] #una lista con 2 elementos
g2 = list(['a', 'b']) #una lista con 2 elementos usando \
constructor
g3 = list(g1) #una lista creada a partir de una lista
```

Los detalles de las operaciones disponibles con un objeto de lista, como agregar, insertar, agregar y eliminar , se pueden revisar en la documentación oficial de Python. Introduciremos las tuplas en la siguiente sección.

tuplas

Una tupla es una lista inmutable, lo que significa que no se puede modificar después de su creación. Las tuplas generalmente se usan para un pequeño número de entradas y cuando la posición y secuencia de las entradas en una colección es importante. Para preservar la secuencia de entradas, las tuplas están diseñadas como inmutables, y aquí es donde las tuplas se diferencian de las listas.

Las operaciones en una tupla suelen ser más rápidas que un tipo de datos de lista regular. En los casos en que se requiere que los valores de una colección sean constantes en un orden particular, el uso de tuplas es la opción preferida debido a su rendimiento superior.

Las tuplas normalmente se inicializan con valores porque son inmutables. Se puede crear una tupla simple usando paréntesis. En el siguiente fragmento de código se muestran algunas formas de crear instancias de tupla:

```
w = () #una tupla vacía  
x = (2, 3) #tupla con dos elementos  
y = ("Hola mundo") #no es una tupla, se requiere coma \  
para tupla de entrada única  
z = ("Hola mundo") #Una coma lo convierte en una tupla
```

En este fragmento de código, creamos una tupla vacía (w), una tupla con números (x) y una tupla con el texto Hello World, que es z. La variable y no es una tupla ya que, para una tupla de 1 (una tupla de un solo objeto), necesitamos una coma final para indicar que es una tupla.

Después de presentar listas y tuplas, presentaremos brevemente los diccionarios.

Diccionarios

Los diccionarios son uno de los tipos de datos más utilizados y versátiles en Python. Un diccionario es una colección que se utiliza para almacenar valores de datos en el formato clave:valor. Los diccionarios son tipos de datos mutables y desordenados. En otros lenguajes de programación, se denominan matrices asociativas o tablas hash.

Se puede crear un diccionario usando corchetes con una lista de pares clave:valor. La clave está separada de su valor por dos puntos ':' y los pares clave:valor están separados por una coma ','. A continuación se muestra un fragmento de código para una definición de diccionario:

```
midict = {  
    "marca": "BMW",  
    "modelo": "330i",  
    "color azul"  
}
```

No se permiten claves duplicadas en un diccionario. Una clave debe ser un tipo de objeto inmutable, como una cadena, una tupla o un número. Los valores en un diccionario pueden ser de cualquier tipo de datos, que incluso incluye listas, conjuntos, objetos personalizados e incluso otro diccionario en sí.

Cuando se trata de diccionarios, tres objetos o listas son importantes:

- **Claves:** se utiliza una lista de claves en un diccionario para iterar a través de los elementos del diccionario.
Se puede obtener una lista de claves utilizando el método keys() :

```
dict_objeto.keys()
```

118 bibliotecas de Python para programación avanzada

- **Valores:** Los valores son los objetos almacenados contra diferentes claves. Una lista de objetos de valor se puede obtener usando el método de valores () :

```
dict_objeto.valores()
```

- **Elementos:** los elementos son los pares clave-valor que se almacenan en un diccionario. Una lista de elementos se puede obtener usando el método items() :

```
dict_objeto.elementos()
```

A continuación, analizaremos los conjuntos, que también son estructuras de datos clave en Python.

Conjuntos

Un conjunto es una colección única de objetos. Un conjunto es una colección mutable y desordenada. No se permite la duplicación de objetos en un conjunto. Python usa una estructura de datos de tabla hash para implementar la unicidad en un conjunto, que es el mismo enfoque que se usa para garantizar la unicidad de las claves en un diccionario. El comportamiento de los conjuntos en Python es muy similar al de los conjuntos en matemáticas. Este tipo de datos encuentra su aplicación en situaciones donde el orden de los objetos no es importante, pero su unicidad sí lo es. Esto ayuda a probar si una determinada colección contiene un determinado objeto o no.

Consejo

Si se requiere el comportamiento de un conjunto como un tipo de datos inmutable, Python tiene una variante de implementación de conjuntos llamada frozenset.

Es posible crear un nuevo objeto de conjunto usando corchetes o usando el constructor de conjuntos (set()). El siguiente fragmento de código muestra algunos ejemplos de cómo crear un conjunto:

```
s1 = conjunto () # conjunto vacío
s2 = {} # un conjunto vacío usando curly
s3 = set(['a', 'b']) # un conjunto creado a partir de una lista con
# const.
s3 = {1,2} # un conjunto creado usando llaves
s4 = {1, 2, 1} # se creará un conjunto con solo 1 y 2 # objetos. El objeto duplicado será ignorado
```

No es posible acceder a los objetos establecidos utilizando un enfoque de indexación. Necesitamos sacar un objeto del conjunto como una lista o podemos iterar en un conjunto para obtener objetos uno por uno. Al igual que los conjuntos matemáticos, los conjuntos en Python también admiten operaciones como unión, intersección y diferencia.

En esta sección, revisamos los conceptos clave de cadenas y tipos de datos de colección en Python 3, que son importantes para comprender el próximo tema: iteradores y generadores.

Uso de iteradores y generadores para el procesamiento de datos

La iteración es una de las herramientas clave utilizadas para el procesamiento y la transformación de datos.

Las iteraciones son especialmente útiles cuando se trata de grandes conjuntos de datos y cuando no es posible o eficiente traer todo el conjunto de datos a la memoria. Los iteradores proporcionan una forma de llevar los datos a la memoria, un elemento a la vez.

Los iteradores se pueden crear definiéndolos con una clase separada e implementando métodos especiales como `__iter__` y `__next__`. Pero también hay una nueva forma de crear iteradores usando la operación `yield`, conocida como generadores. En las siguientes subsecciones, estudiaremos tanto los iteradores como los generadores.

iteradores

Los iteradores son los objetos que se utilizan para iterar sobre otros objetos. Un objeto sobre el que un iterador puede iterar se denomina **iterable**. En teoría, los dos objetos son diferentes, pero es posible implementar un iterador dentro de la clase de objeto iterable . Esto no se recomienda pero es técnicamente posible y discutiremos con un ejemplo por qué este enfoque no es un buen enfoque de diseño. En el siguiente fragmento de código, proporcionamos algunos ejemplos del uso del bucle `for` con fines de iteración en Python :

#iterador1.py

#ejemplo 1: iterando en una lista

```
para x en [1,2,3]:
```

```
    imprimir (x)
```

#ejemplo 2: iterando en una cadena

```
para x en "Python para Geeks":
```

```
    imprimir (x, fin = "")
```

```
    imprimir("")
```

#ejemplo 3: iterando en un diccionario

```
días_semana = {1:'Lun', 2:'Mar',
```

```
3: 'miércoles', 4: 'jueves',
```

120 bibliotecas de Python para programación avanzada

```
5: 'vie', 6: 'sábado', 7: 'dom'}
para k en días_semana:
    imprimir(k, días_semana[k])
```

#ejemplo 4: iterando en un archivo

```
para fila en abierto('abc.txt'):
    imprimir (fila, final = "")
```

En estos ejemplos de código, usamos diferentes bucles for para iterar en una lista, una cadena, un diccionario y un archivo. Todos estos tipos de datos son iterables y, por lo tanto, usaremos una sintaxis simple con el ciclo for para recorrer los elementos en estas colecciones o secuencias. A continuación, estudiaremos qué ingredientes hacen que un objeto sea iterable, lo que también se conoce como **Protocolo Iterador**.

Nota IMPORTANTE

Cada colección en Python es iterable por defecto.

En Python, un objeto iterador debe implementar dos métodos especiales: `__iter__` y `__next__`. Para iterar sobre un objeto, el objeto tiene que implementar al menos el método `__iter__`. Una vez que el objeto implementa el método `__iter__`, podemos llamar al objeto iterable. Estos métodos se describen a continuación:

- `__iter__`: este método devuelve el objeto iterador. Este método se llama al comienzo de un ciclo para obtener el objeto iterador.
- `__next__`: este método se llama en cada iteración del bucle y devuelve el siguiente elemento en el objeto iterable.

Para explicar cómo crear un objeto personalizado que sea iterable, implementaremos la clase Week , que almacena los números y nombres de todos los días de la semana en un diccionario. Esta clase no será iterable por defecto. Para hacerlo iterable, agregaremos `__iter__`. Para simplificar el ejemplo, también agregaremos el método `__next__` en la misma clase. Aquí está el fragmento de código con la clase Week y el programa principal, que itera para obtener los nombres de los días de la semana:

```
#iterador2.py
semana de clase:
def __init__(uno mismo):
    self.días = {1:'Lunes', 2: "Martes",
3: "miércoles", 4: "jueves",
5:"Viernes", 6:"Sábado", 7:"Domingo"}
```

```
self._índice = 1

def __iter__(self): self._index = 1

regresar a sí mismo

def __siguiente__(uno mismo):

    si self._index < 1 | self._index > 7: aumentar StopIteration

    demás:

    ret_value = self.días[self._index] self._index +=1 return ret_value

if(__nombre__ == "__principal__"):
    semana = Semana()
    para el día en la semana:
        imprimir (día)
```

Compartimos este ejemplo de código solo para demostrar cómo se pueden implementar los métodos `__iter__` y `__next__` en la misma clase de objeto. Este estilo de implementar un iterador se encuentra comúnmente en Internet, pero no es un enfoque recomendado y se considera un mal diseño. La razón es que cuando lo usamos en el bucle `for`, recuperamos el objeto principal como iterador, ya que implementamos `__iter__` y `__next__` en la misma clase. Esto puede dar resultados impredecibles. Podemos demostrar esto ejecutando el siguiente fragmento de código para la misma clase, `Semana`:

```
#iterador3.py
semana de clase:

La definición de #clase es la misma que se muestra en la anterior \
ejemplo de código

if(__name__ == "__main__"): wk = Week()

iter1 = iter(semana)
iter2 = iter(semana)

imprimir(iter1.__siguiente__())
```

122 bibliotecas de Python para programación avanzada

```
imprimir(iter2.__siguiente__())
imprimir (siguiente (iter1))
imprimir (siguiente (iter2))
```

En este nuevo programa principal, estamos iterando sobre el mismo objeto usando dos iteradores diferentes. Los resultados de este programa principal no son los esperados. Esto se debe a un atributo `_index` común compartido por los dos iteradores. Aquí hay una salida de consola como referencia:

```
Lunes
martes
miércoles
jueves
```

Tenga en cuenta que en este nuevo programa principal deliberadamente no usamos un bucle `for`. Creamos dos objetos iteradores para el mismo objeto de la clase `Week` usando la función `iter`. La función `iter` es una función estándar de Python que llama al método `__iter__`. Para obtener el siguiente elemento en el objeto iterable, usamos directamente el método `__next__` y la siguiente función. La siguiente función también es una función general, como la función `iter`. Este enfoque de usar un iterable como iterador tampoco se considera seguro para subprocessos.

El mejor enfoque es siempre usar una clase de iterador separada y siempre crear una nueva instancia de un iterador a través del método `__iter__`. Cada instancia de iterador tiene que gestionar su propio estado interno. Una versión revisada del mismo ejemplo de código de la Semana La clase se muestra a continuación con una clase de iterador separada:

```
#iterador4.py
semana de clase:
def __init__(uno mismo):
    self.días = {1: 'Lunes', 2: "Martes",
            3: "miércoles", 4: "jueves",
            5: "viernes", 6: "sábado", 7: "domingo"}

def __iter__(uno mismo):
    return WeekIterator(self.días)

iterador de semana de clase :
def __init__(self, días):
    self.days_ref = diáss
```

```
self._índice = 1

def __siguiente__(uno mismo):
    si self._índice < 1 | self._índice > 8:
        subir StopIteration
    demás:
        ret_value = self.days_ref[self._índice]
        auto._índice +=1
        devolver ret_valor

    if(__nombre__ == "__principal__"):
        semana = Semana()
        iter1 = iter(semana)
        iter2 = iter(semana)
        imprimir(iter1.__siguiente__())
        imprimir(iter2.__siguiente__())
        imprimir (siguiente (iter1))
        imprimir (siguiente (iter2))
```

En este ejemplo de código revisado, tenemos una clase de iterador separada con el método `__next__` y tiene su propio atributo `_index` para administrar el estado del iterador. La instancia del iterador tendrá una referencia al objeto contenedor (diccionario). La salida de la consola del ejemplo revisado da los resultados esperados: cada iterador está iterando por separado en la misma instancia de la clase `Semana`. La salida de la consola se muestra a continuación como referencia:

```
Lunes
Lunes
martes
martes
```

En resumen, para crear un iterador, necesitamos implementar `__iter__` y `__next__` métodos, administrar el estado interno y generar una excepción `StopIteration` cuando no hay valores disponibles. A continuación, estudiaremos los generadores, lo que simplificará la forma en que devolvemos los iteradores.

Generadores

Un generador es una forma simple de devolver una instancia de iterador que se puede usar para la iteración, lo que se logra implementando solo una función de generador. Una función de generador es similar a una función normal pero con una declaración de rendimiento en lugar de una declaración de retorno . La declaración de devolución todavía está permitida en una función de generador, pero no se utilizará para devolver el siguiente elemento en un objeto iterable.

Por definición, una función será una función generadora si tiene al menos una declaración de rendimiento . La principal diferencia cuando se usa la declaración de rendimiento es que pausa la función y guarda su estado interno, y cuando se llama a la función la próxima vez, comienza desde la línea que arrojó la última vez. Este patrón de diseño hace que la funcionalidad del iterador sea simple y eficiente.

Internamente, los métodos como `__iter__` y `__next__` se implementan automáticamente y la excepción `StopIteration` también se genera automáticamente. Los atributos locales y sus valores se conservan entre las llamadas sucesivas y el desarrollador no debe implementar ninguna lógica adicional. El intérprete de Python proporciona toda esta funcionalidad cada vez que identifica una función generadora (una función con una declaración de rendimiento).

Para comprender cómo funciona el generador, comenzaremos con un ejemplo de generador simple que se utiliza para generar una secuencia de las tres primeras letras del alfabeto:

```
#generadores1.py
def mi_gen():
    rendimiento 'A'
    rendimiento 'B'
    rendimiento 'C'

    if(__nombre__ == "__principal__"):
        iter1 = mi_gen()
        imprimir(iter1.__siguiente__())
        imprimir (siguiente (iter1))
        imprimir(iter1.__siguiente__())
```

En este ejemplo de código, implementamos una función de generador simple usando tres rendimiento declaraciones sin declaración de retorno . En la parte principal del programa, hicimos lo siguiente:

1. Llamamos a la función generadora, que nos devuelve una instancia de iterador. En esta etapa, no se ejecuta ninguna línea dentro de la función generadora `my_gen()` .

2. Usando la instancia del iterador, llamamos al método `__next__`, que inicia la ejecución de la función `my_gen()`, se detiene después de ejecutar el primer rendimiento declaración, y devuelve A.
3. A continuación, llamamos a la función `next()` en la instancia del iterador. El resultado es el mismo que obtenemos con el método `__next__`. Pero esta vez, la función `my_gen()` inicia la ejecución desde la siguiente línea desde donde se detuvo la última vez debido a la declaración de rendimiento. La siguiente línea es otra declaración de rendimiento, lo que da como resultado otra pausa después de devolver la letra B.
4. El siguiente método `__next__` dará como resultado la ejecución del siguiente rendimiento instrucción, que devolverá la letra C.

A continuación, revisaremos la clase Semana y su implementación iteradora y usaremos un generador en lugar de una clase iteradora. El código de ejemplo se presenta a continuación:

```
#generador2.py
semana de clase:
def __init__(uno mismo):
    self.dias = {1:'Lunes', 2: "Martes",
3: "miércoles", 4: "jueves",
5:"Viernes", 6:"Sábado", 7:"Domingo"}

def semana_gen(auto):
    para x en self.days:
        rendimiento self.days[x]

    if(__nombre__ == "__principal__"):
        semana = Semana()
    iter1 = semana.semana_gen()
    iter2 = iter(sem.week_gen())
    imprimir(iter1.__siguiente__())
    imprimir(iter2.__siguiente__())
    imprimir (siguiente (iter1))
    imprimir (siguiente (iter2))
```

126 bibliotecas de Python para programación avanzada

En comparación con iterator4.py, la implementación de la clase Week con un generador es mucho más simple y limpia y podemos lograr los mismos resultados. Este es el poder de los generadores y por eso son muy populares en Python. Antes de concluir este tema, es importante destacar algunas otras características clave de los generadores:

- **Expresiones generadoras :** las expresiones generadoras se pueden usar para crear generadores simples (también conocidos como **funciones anónimas**) sobre la marcha sin escribir un método especial. La sintaxis es similar a la comprensión de listas, excepto que usamos paréntesis en lugar de corchetes. El siguiente ejemplo de código (una extensión del ejemplo que presentamos para la comprensión de listas) muestra cómo se puede usar una expresión de generador para crear un generador, su uso y también una comparación con la comprensión de listas:

#generador3.py

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
f1 = [x+1 para x en L]
g1 = (x+1 para x en L)

imprimir(g1.__siguiente__())
imprimir(g1.__siguiente__())
```

- **Flujos infinitos:** los generadores también se pueden usar para implementar un flujo infinito de datos.

Siempre es un desafío llevar un flujo infinito a la memoria, lo cual se resuelve fácilmente con generadores, ya que solo devuelven un elemento de datos a la vez.

- **Generadores de canalización:** cuando se trabaja con problemas complejos, se pueden utilizar varios generadores como canalización para lograr cualquier objetivo. El concepto de canalización de múltiples generadores se puede explicar con un ejemplo. Asumimos un problema, que es sacar la suma de los cuadrados de números primos. Este problema se puede resolver con bucles for tradicionales , pero intentaremos resolverlo usando dos generadores: el prime_gen para generar números primos y el generador x2_gen para tomar el cuadrado de los números primos alimentados a este generador por el generador prime_gen . Introducimos los dos generadores canalizados en la función de suma para obtener el resultado deseado. Aquí está el fragmento de código para la solución de este problema:

#generador4.py

```
def prime_gen(núm):
    para cand en rango (2, num+1):
        para i en rango (2, cand):
            si (can % i) == 0:
                romper
```

```
demás:  
rendimiento cand  
  
def x2_gen(lista2):  
    para num en list2:  
        número de rendimiento * número  
  
imprimir (suma (x2_gen (prime_gen (5))))
```

Los generadores funcionan bajo demanda, lo que los hace no solo eficientes en términos de memoria, sino que también proporciona una forma de generar valores cuando se necesitan. Esto ayuda a evitar la generación de datos innecesarios, que pueden no utilizarse en absoluto. Los generadores son adecuados para una gran cantidad de procesamiento de datos, para canalizar los datos de una función a otras y también para simular la concurrencia.

En la siguiente sección, investigaremos cómo manejar archivos en Python.

Manejo de archivos en Python

Leer datos de un archivo o escribir datos en un archivo es una de las operaciones fundamentales admitidas por cualquier lenguaje de programación. Python proporciona un amplio soporte para el manejo de operaciones con archivos, que en su mayoría están disponibles en su biblioteca estándar. En esta sección, analizaremos las operaciones de archivos principales, como abrir un archivo, cerrar un archivo, leer de un archivo, escribir en un archivo, administrar archivos con administradores de contexto y abrir varios archivos con un identificador utilizando la biblioteca estándar de Python. Comenzaremos nuestra discusión con las operaciones de archivo en la siguiente subsección.

Operaciones de archivos

Las operaciones de archivo generalmente comienzan con la apertura de un archivo y luego la lectura o actualización del contenido de ese archivo. Las operaciones del archivo principal son las siguientes:

Abrir y cerrar un archivo

Para aplicar cualquier operación de lectura o actualización a un archivo, necesitamos un puntero o referencia al archivo. Se puede obtener una referencia de archivo abriendo un archivo usando la función abierta incorporada. Esta función devuelve una referencia al objeto de archivo , que también se conoce como **identificador de archivo** en alguna literatura. El requisito mínimo con la función abrir es el nombre del archivo con una ruta absoluta o relativa. Un parámetro opcional es el modo de acceso para indicar en qué modo se debe abrir un archivo. El modo de acceso puede ser lectura, escritura, anexar u otros. Una lista completa de las opciones de modo de acceso es la siguiente:

- r: esta opción es para abrir un archivo en modo de sólo lectura. Esta es una opción predeterminada si no se proporciona la opción de modo de acceso:

```
f = abierto ('abc.txt')
```

- a: esta opción es para abrir un archivo y agregar una nueva línea al final del archivo:

```
f = abierto ('abc.txt', 'a')
```

- w: Esta opción es para abrir un archivo para escribir. Si un archivo no existe, creará un nuevo archivo. Si el archivo existe, esta opción lo anulará y cualquier contenido existente en ese archivo será destruido:

```
f = abierto ('abc.txt', 'w')
```

- x: Esta opción es para abrir un archivo para escritura exclusiva. Si el archivo ya existe, arrojará un error:

```
f = abierto ('abc.txt', 'x')
```

- t: Esta opción es para abrir un archivo en modo texto. Esta es la opción por defecto.
- b: Esta opción es para abrir un archivo en modo binario.
- +: Esta opción es para abrir un archivo para lectura y escritura:

```
f = abierto ('abc.txt', 'r+')
```

Las opciones de modo se pueden combinar para obtener múltiples opciones. Además del nombre del archivo y las opciones del modo de acceso, también podemos pasar el tipo de codificación, especialmente para archivos de texto. Aquí hay un ejemplo de cómo abrir un archivo con utf-8:

```
f = abrir ("abc.txt", modo = 'r', codificación = 'utf-8')
```

Cuando completamos nuestras operaciones con un archivo, es imprescindible cerrar el archivo para liberar los recursos para que otros procesos utilicen el archivo. Un archivo se puede cerrar usando el cierre en la instancia del archivo o en el identificador del archivo. Aquí hay un fragmento de código que muestra el uso del método close :

```
archivo = abrir("abc.txt", 'r+w')
#operaciones en archivo
archivo.cerrar()
```

Una vez que se cierra un archivo, el sistema operativo liberará los recursos asociados con la instancia del archivo y los bloqueos (si los hay), lo cual es una buena práctica en cualquier lenguaje de programación.

Lectura y escritura de archivos.

Se puede leer un archivo abriendo el archivo en el modo de acceso r y luego usando uno de los métodos de lectura. A continuación, resumimos diferentes métodos disponibles para operaciones de lectura:

- read(n): este método lee n caracteres de un archivo.
- readline(): este método devuelve una línea de un archivo.
- readlines(): este método devuelve todas las líneas de un archivo en forma de lista.

De manera similar, podemos agregar o escribir en un archivo una vez que se abre en un modo de acceso apropiado. Los métodos que son relevantes para agregar un archivo son los siguientes:

- escribir (x): este método escribe una cadena o una secuencia de bytes en un archivo y devuelve el número de caracteres añadidos al archivo.
- writelines (líneas): este método escribe una lista de líneas en un archivo.

En el siguiente ejemplo de código, crearemos un nuevo archivo, le agregaremos algunas líneas de texto y luego leeremos los datos de texto usando las operaciones de lectura discutidas anteriormente:

```
#writereadfile.py: escribe en un archivo y luego lee de él
f1 = abrir("miarchivo.txt",'w')
f1.write("Este es un archivo de muestra\n")
líneas =[ "Estos son datos de prueba\n", "en dos líneas\n"]
f1.writelines(líneas)
f1.cerrar()

f2 = abrir("miarchivo.txt",'r')
```

130 bibliotecas de Python para programación avanzada

```
imprimir (f2. leer (4))
imprimir(f2.readline())
imprimir(f2.readline())

f2.buscar(0)
para línea en f2.readlines():
imprimir (línea)
f2.cerrar()
```

En este ejemplo de código, primero escribimos tres líneas en un archivo. En las operaciones de lectura, primero, leer cuatro caracteres, seguido de la lectura de dos líneas usando el método `readline` . Al final, volvemos a mover el puntero a la parte superior del archivo usando el método de búsqueda y accedemos a todas las líneas del archivo usando el método `readlines` .

En la siguiente sección, veremos cómo el uso de un administrador de contexto facilita el manejo de archivos.

Usar un administrador de contexto

El uso correcto y justo de los recursos es fundamental en cualquier lenguaje de programación. Un controlador de archivos y una conexión a la base de datos son un par de muchos ejemplos en los que es una práctica común no liberar los recursos a tiempo después de trabajar con los objetos. Si los recursos no se liberan en absoluto, terminará en una situación llamada **fuga de memoria** y puede afectar el rendimiento del sistema y, en última instancia, puede provocar que el sistema se bloquee.

Para resolver este problema de pérdida de memoria y liberación oportuna de recursos, a Python se le ocurrió el concepto de administradores de contexto. Un administrador de contexto está diseñado para reservar y liberar recursos precisamente según el diseño. Cuando se usa un administrador de contexto con la palabra clave `with` , se espera que una declaración después de la palabra clave `with` devuelva un objeto que debe implementar el **protocolo de administración de contexto**. Este protocolo requiere que el objeto devuelto implemente dos métodos especiales. Estos métodos especiales son los siguientes:

- `__enter__()`: este método se llama con la palabra clave `with` y se utiliza para reservar los recursos necesarios según la instrucción que sigue a la palabra clave `with` .
- `__exit__()`: Este método se llama después de la ejecución del bloque `with` y se utiliza para liberar los recursos que están reservados en el método `__enter__()` .

Por ejemplo, cuando se abre un archivo utilizando el administrador de contexto con declaración (bloque), no es necesario cerrar el archivo. La declaración de apertura de archivo devolverá el objeto controlador de archivo, que ya ha implementado el protocolo de administración de contexto y el archivo se cerrará automáticamente tan pronto como se complete la ejecución del bloque `with`. Una versión revisada del ejemplo de código para escribir y leer un archivo usando el administrador de contexto es la siguiente:

```
#contextmgr1.py
con open("myfile.txt",'w') como f1:
    f1.write("Este es un archivo de muestra\n")
    líneas = ["Estos son datos de prueba\n", "en dos líneas\n"]
    f1.writelines(líneas)

    con open("myfile.txt",'r') como f2:
        para línea en f2.readlines():
            imprimir (línea)
```

El código con el administrador de contexto es simple y fácil de leer. El uso de un administrador de contexto es un enfoque recomendado para abrir y trabajar con archivos.

Operando en varios archivos

Python admite abrir y operar en varios archivos al mismo tiempo. Podemos abrir estos archivos en diferentes modos y operar sobre ellos. No hay límite en el número de archivos. Podemos abrir dos archivos en modo de lectura usando el siguiente código de muestra y acceder a los archivos en cualquier orden:

```
1.txt
Este es un archivo de muestra 1
Estos son datos de prueba 1

2.txt
Este es un archivo de muestra 2
Estos son datos de prueba 2

#multifilesread1.py
con abrir("1.txt") como archivo1, abrir("2.txt") como archivo2:
    imprimir(archivo2.readline())
    imprimir(archivo1.readline())
```

132 bibliotecas de Python para programación avanzada

También podemos leer de un archivo y escribir en otro archivo usando esta opción de operación multiarchivo. El código de muestra para transferir contenido de un archivo a otro archivo es el siguiente:

```
#multifilesread2.py

con open("1.txt",'r') como archivo1, open("3.txt",'w') como archivo2:
    para línea en file1.readlines():
        archivo2.escribir(línea)
```

Python también tiene una solución más elegante para operar en múltiples archivos usando la entrada de archivo módulo. La función de entrada de este módulo puede tomar una lista de múltiples archivos y luego tratar todos esos archivos como una sola entrada. El código de muestra con dos archivos de entrada, 1.txt y 2.txt , y usando el módulo de entrada de archivos se presenta a continuación:

```
#multifilesread1.py

importar entrada de archivo
con fileinput.input(files = ("1.txt",'2.txt')) como f:
    para línea en f:
        imprimir (f. nombre de archivo ())
        imprimir (línea)
```

Con este enfoque, obtenemos un identificador de archivo que opera en varios archivos de forma secuencial. A continuación, discutiremos el manejo de errores y excepciones en Python.

Manejo de errores y excepciones

Hay muchos tipos de errores posibles en Python. El más común está relacionado con la sintaxis del programa y normalmente se conoce como **error de sintaxis**. En muchas ocasiones se reportan errores durante la ejecución de un programa. Estos errores se denominan **errores de tiempo de ejecución**. Los errores de tiempo de ejecución que se pueden manejar dentro de nuestro programa se denominan **excepciones**.

Esta sección se centrará en cómo manejar errores o excepciones en tiempo de ejecución. Antes de pasar al manejo de errores, presentaremos brevemente los errores de tiempo de ejecución más comunes de la siguiente manera:

- **IndexError**: este error se produce cuando un programa intenta acceder a un elemento en un índice inválido (ubicación en la memoria).
- **ModuleNotFoundError**: este error aparecerá cuando un módulo específico no se encuentra en la ruta del sistema.
- **ZeroDivisionError**: este error se produce cuando un programa intenta dividir un número por cero.

- **KeyError:** este error se produce cuando un programa intenta obtener un valor de un diccionario utilizando una clave no válida.
- **StopIteration:** Este error se lanza cuando el método `__next__` no encuentra cualquier artículo adicional en un contenedor.
- **TypeError:** este error ocurre cuando un programa intenta aplicar una operación en un objeto de un tipo inapropiado.

Una lista completa de errores está disponible en la documentación oficial de Python. En las siguientes subsecciones, discutiremos cómo manejar los errores, a veces también llamados excepciones, usando construcciones apropiadas en Python.

Trabajando con excepciones en Python

Cuando surgen errores de tiempo de ejecución, el programa puede terminar abruptamente y causar daños a los recursos del sistema, como archivos corruptos y tablas de bases de datos. Esta es la razón por la que el manejo de errores o excepciones es uno de los ingredientes clave para escribir programas robustos en cualquier lenguaje. La idea es anticipar que pueden ocurrir errores de tiempo de ejecución y si tal error ocurre, cuál sería el comportamiento de nuestro programa como respuesta a ese error en particular.

Como muchos otros lenguajes, Python usa las palabras clave `try` y `except`. Las dos palabras clave van seguidas de bloques separados de código que se ejecutarán. El bloque `try` es un conjunto regular de declaraciones para las que anticipamos que puede ocurrir un error. El `except` el bloque se ejecutará solo si hay un error en un bloque de prueba . La siguiente es la sintaxis de escribir código Python con bloques de prueba y excepción :

```
tratar:  
#una serie de declaraciones  
  
excepto:  
#instrucciones a ejecutar si hay un error en \  
intentar bloquear
```

Si anticipamos un tipo de error en particular o múltiples tipos de error, podemos definir una excepción bloque con el nombre del error y puede agregar tantos bloques `except` como necesitemos. Dichos bloques de excepción con nombre se ejecutan solo si la excepción con nombre se genera en el bloque de prueba . Con la declaración de bloque de excepción , también podemos agregar una declaración `as` para almacenar el objeto de excepción como una variable que se genera durante el bloque de prueba . El bloque de prueba en el siguiente ejemplo de código tiene muchos posibles errores de tiempo de ejecución y es por eso que tiene varios bloques de excepción :

```
#excepción1.py  
tratar:  
imprimir (x)
```

134 bibliotecas de Python para programación avanzada

```
x = 5
y = 0
z = x/y
imprimir('x'+ y)
excepto NameError como e:
    imprimir (e)
excepto ZeroDivisionError:
    print("No se permite dividir por 0")
excepto Excepción como e:
    imprimir("Ha ocurrido un error")
    imprimir (e)
```

Para ilustrar un mejor uso de uno o más bloques de excepción , agregamos varios bloques de excepción que se explican a continuación:

- **El bloque NameError:** este bloque se ejecutará cuando una declaración en el intento bloque intente acceder a una variable indefinida. En nuestro ejemplo de código, este bloque se ejecutará cuando el intérprete intente ejecutar la instrucción print(x) . Además, nombramos el objeto de excepción como e y lo usamos con la impresión declaración para obtener el detalle oficial del error asociado con este tipo de error.
- **El bloque ZeroDivisionError:** Este bloque se ejecutará cuando intentemos ejecutar z = x/y e y = 0. Para que este bloque se ejecute, necesitamos corregir el NameError bloque primero.
- **El bloque de excepción predeterminado:** este es un bloque de excepción general , lo que significa que si no se encuentra una coincidencia con los dos bloques anteriores , excepto que se ejecutará este bloque. La última instrucción print('x'+ y) también generará un error de tipo TypeError y será manejado por este bloque. Dado que no recibimos ningún tipo particular de excepción en este bloque, podemos usar la palabra clave Exception para almacenar el objeto de excepción en una variable.

Tenga en cuenta que tan pronto como ocurre un error en cualquier declaración en el bloque de prueba , el resto de las declaraciones se ignoran y el control va a uno de los bloques de excepción . En nuestro ejemplo de código, primero debemos corregir el error NameError para ver el siguiente nivel de excepción y así sucesivamente. Agregamos tres tipos diferentes de errores en nuestro ejemplo para demostrar cómo definir varios bloques excepto para el mismo bloque de prueba . El orden de los bloques de excepción es importante porque los bloques de excepción más específicos con nombres de error deben definirse primero y un bloque de excepción sin especificar un nombre de error siempre debe estar al final.

La siguiente figura muestra todos los bloques de manejo de excepciones:

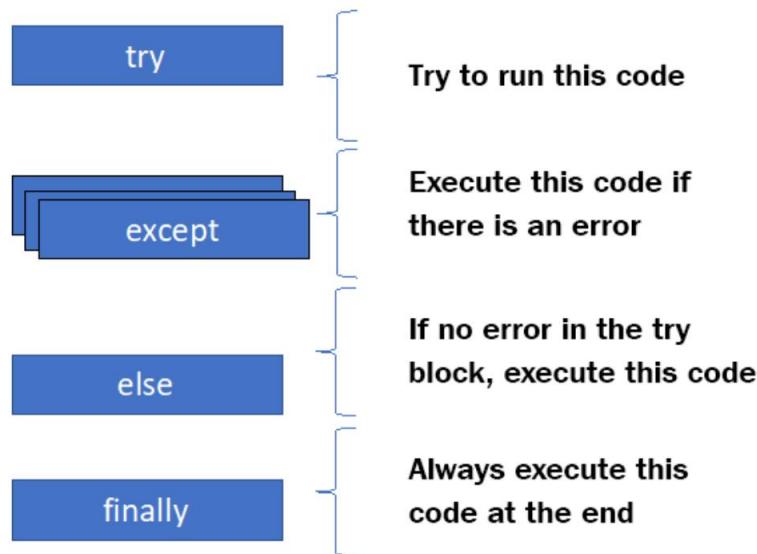


Figura 4.1 – Diferentes bloques de manejo de excepciones en Python

Como se muestra en el diagrama anterior, además de los bloques de prueba y excepción , Python también admite los bloques else y finalmente para mejorar la funcionalidad de manejo de errores. El bloque else se ejecuta si no se generaron errores durante el bloque try . El código de este bloque se ejecutará con normalidad y no se producirá ninguna excepción si se produce algún error dentro de este bloque. Los bloques de prueba y excepción anidados se pueden agregar dentro del bloque else si es necesario. Tenga en cuenta que este bloque es opcional.

El bloque " finally " se ejecuta independientemente de si hay un error en el bloque " try " o no. El código dentro del bloque finalmente se ejecuta sin ningún mecanismo de manejo de excepciones. Este bloque se utiliza principalmente para liberar recursos cerrando las conexiones o archivos abiertos. Aunque es un bloque opcional, es muy recomendable implementar este bloque.

A continuación, veremos el uso de estos bloques con un ejemplo de código. En este ejemplo, abriremos un nuevo archivo para escribir en el bloque de prueba . Si ocurre un error al abrir el archivo, se lanzará una excepción y enviaremos los detalles del error a la consola usando la declaración de impresión en el bloque de excepción . Si no ocurre ningún error, ejecutaremos el código en el bloque else que está escribiendo algún texto en el archivo. En ambos casos (con error o sin error), cerraremos el archivo en el bloque finalmente . El código de ejemplo completo es el siguiente:

```
#excepción2.py
```

```
tratar:
```

136 bibliotecas de Python para programación avanzada

```
f = abrir("abc.txt", "w")
excepto Excepción como e:
    imprimir("Error:" + e)
demás:
    f.write("Hola mundo")
    f.write("Fin")
por fin:
    f.cerrar()
```

Hemos cubierto extensamente cómo manejar una excepción en Python. A continuación, discutiremos cómo generar una excepción desde el código de Python.

Generar excepciones

El intérprete de Python genera excepciones o errores en tiempo de ejecución cuando se produce un error.

También podemos generar errores o excepciones nosotros mismos si ocurre una condición que puede darnos una mala salida o bloquear el programa si seguimos adelante. Generar un error o una excepción proporcionará una salida elegante del programa.

Se puede lanzar una excepción (objeto) a la persona que llama usando la palabra clave `raise`.

Una excepción puede ser de uno de los siguientes tipos:

- Una excepción incorporada
- Una excepción personalizada
- Un objeto de excepción genérico

En el siguiente ejemplo de código, llamaremos a una función simple para calcular una raíz cuadrada y la implementaremos para generar una excepción si el parámetro de entrada no es un número positivo válido:

```
#excepción3.py
importar matematicas
def sqrt(num):

    si no es instancia (num, (int, float)) :
        aumentar TypeError("solo se permiten números")
    si numero < 0:
        aumentar la excepción ("Número negativo no admitido")
```

```
devuelve matemáticas.sqrt(num)

si __nombre__ == "__principal__":
    tratar:
        imprimir (raíz cuadrada (9))
imprimir (raíz cuadrada ('a'))
imprimir (raíz cuadrada (-9))
    excepto Excepción como e:
        imprimir (e)
```

En este ejemplo de código, generamos una excepción integrada al crear una nueva instancia de la clase `TypeError` cuando el número pasado a la función `sqrt` no es un número.

También lanzamos una excepción genérica cuando el número pasado es inferior a 0. En ambos casos, pasamos nuestro texto personalizado a su constructor. En la siguiente sección, estudiaremos cómo definir nuestra propia excepción personalizada y luego lanzarla a la persona que llama.

Definición de excepciones personalizadas

En Python, podemos definir nuestras propias excepciones personalizadas creando una nueva clase que debe derivarse de la clase `Exception` integrada o su subclase. Para ilustrar el concepto, revisaremos nuestro ejemplo anterior mediante la definición de dos clases de excepción personalizadas para reemplazar los tipos de error integrados `TypeError` y `Exception`. Las nuevas clases de excepción personalizadas se derivarán de las clases `TypeError` y `Exception`. Aquí hay un código de muestra para referencia con excepciones personalizadas:

```
#excepción4.py
importar matematicas

clase NumTypeError(TypeError):
    aprobar

clase NegativeNumError (Excepción):
    def __init__(uno mismo):
        super().__init__("Número negativo no admitido")

    def sqrt(num):
        si no es instancia (num, (int, float)) :
            aumentar NumTypeError("solo se permiten números")
```

138 bibliotecas de Python para programación avanzada

```
si numero < 0:  
    aumentar NegativeNumError  
  
devuelve matemáticas.sqrt(num)  
  
si __nombre__ == "__principal__":  
    tratar:  
        imprimir (raíz cuadrada (9))  
        imprimir (raíz cuadrada ('a'))  
        imprimir (raíz cuadrada (-9))  
    excepto NumTypeError como e:  
        imprimir (e)  
    excepto NegativeNumError como e:  
        imprimir (e)
```

En este ejemplo de código, la clase NumTypeError se deriva de la clase TypeError y no hemos agregado nada en esta clase. La clase NegativeNumError se hereda de la clase Exception y anulamos su constructor y agregamos un mensaje personalizado para esta excepción como parte del constructor. Cuando lanzamos estas excepciones personalizadas en la función sqrt() , no pasamos ningún texto con la clase de excepción NegativeNumError . Cuando usamos el programa principal, obtenemos el mensaje con la letra (e)

tal como lo hemos establecido como parte de la definición de clase.

En esta sección, cubrimos cómo manejar los tipos de errores integrados usando Try y Except bloques, cómo definir excepciones personalizadas y cómo generar una excepción de forma declarativa. En la siguiente sección, cubriremos el inicio de sesión en Python.

Usando el módulo de registro de Python

El registro es un requisito fundamental para cualquier aplicación de tamaño razonable. El registro no solo ayuda a depurar y solucionar problemas, sino que también proporciona información sobre los detalles de los problemas internos de una aplicación. Algunas ventajas del registro son las siguientes:

- Código de depuración, especialmente para diagnosticar por qué y cuándo una aplicación falló o se bloqueó
- Diagnóstico del comportamiento inusual de la aplicación
- Proporcionar datos de auditoría para cuestiones de cumplimiento normativo o legal
- Identificar los comportamientos de los usuarios y los intentos maliciosos de acceder a recursos no autorizados

Antes de discutir cualquier ejemplo práctico de registro, primero discutiremos los componentes clave del sistema de registro en Python.

Presentación de los componentes principales de registro

Los siguientes componentes son fundamentales para configurar el registro de una aplicación en Python:

- Registrador
- Niveles de registro
- Formateador de registro
- Controlador de registro

Aquí se puede resumir una arquitectura de alto nivel del sistema de registro de Python:

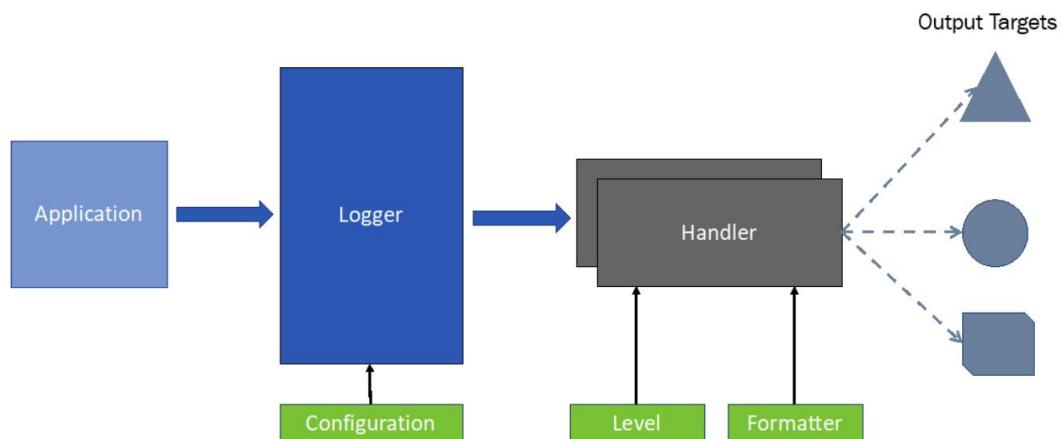


Figura 4.2 – Componentes de registro en Python

Cada uno de estos componentes se analiza en detalle en las siguientes subsecciones.

el registrador

El registrador es el punto de entrada al sistema de registro de Python. Es la interfaz para el programador de aplicaciones. La clase Logger disponible en Python proporciona varios métodos para registrar mensajes con diferentes prioridades. Estudiaremos los métodos de la clase Logger con ejemplos de código más adelante en esta sección.

140 bibliotecas de Python para programación avanzada

Una aplicación interactúa con la instancia de Logger , que se configura mediante la configuración de registro, como el nivel de registro. Al recibir eventos de registro, la instancia de Logger selecciona uno o más controladores de registro apropiados y delega los eventos a los controladores.

Cada controlador suele estar diseñado para un destino de salida específico. Un controlador envía los mensajes después de aplicar un filtro y formato al destino de salida previsto.

Niveles de registro

Todos los eventos y mensajes de un sistema de registro no tienen la misma prioridad. Por ejemplo, los mensajes sobre errores son más urgentes que los mensajes de advertencia. Los niveles de registro son una forma de establecer diferentes prioridades para diferentes eventos de registro. Hay seis niveles definidos en Python. Cada nivel está asociado con un valor entero que indica la gravedad. Estos niveles son NOTSET, DEBUG, INFO, WARNING, ERROR y CRITICAL. Estos se resumen aquí:

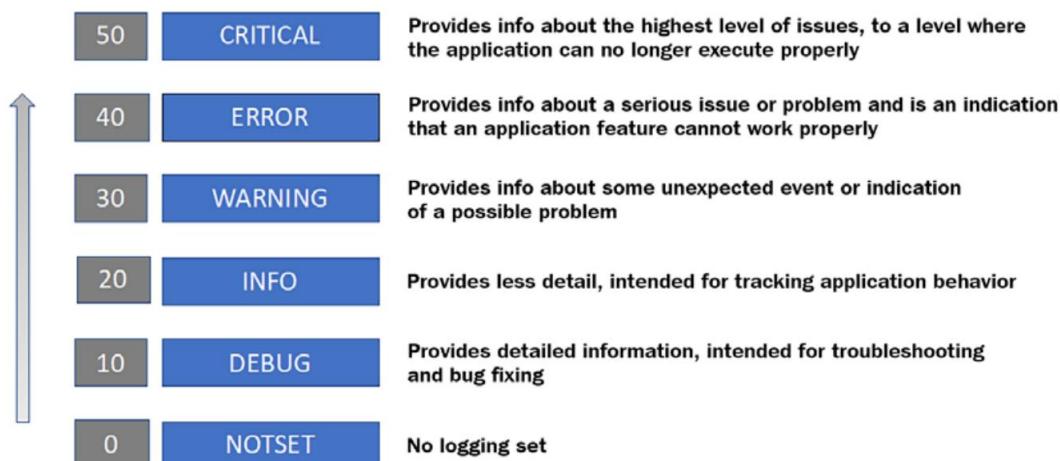


Figura 4.3 – Niveles de registro en Python

El formateador de registro

El componente del formateador de registro ayuda a mejorar el formato de los mensajes, lo cual es importante para la coherencia y la legibilidad humana y mecánica. El formateador de registro también agrega contexto adicional a los mensajes, como la hora, el nombre del módulo, el número de línea, los subprocessos y el proceso, lo que es extremadamente útil para fines de depuración. Un ejemplo de expresión del formateador es el siguiente:

```
%(asctime)s — %(name)s — %(levelname)s — %(funcName)
s:%(lineno)d — %(mensaje)s"
```

Cuando se usa una expresión de formateador de este tipo, el mensaje de registro hola Geeks of level INFO se mostrará similar a la salida de la consola que sigue:

```
2021-06-10 19:20:10,864 - abc - INFORMACIÓN - <nombre del módulo>: 10 -  
hola frikis
```

El controlador de registro

La función de un controlador de registro es escribir datos de registro en un destino adecuado, que puede ser una consola, un archivo o incluso un correo electrónico. Hay muchos tipos de controladores de registro integrados disponibles en Python. Aquí se presentan algunos controladores populares:

- StreamHandler para mostrar los registros en una consola
- FileHandler para escribir los registros en un archivo
- SMTPHandler para enviar los registros a un correo electrónico
- SocketHandler para enviar los registros a un socket de red
- SyslogHandler para enviar los registros a un servidor syslog de Unix local o remoto
- HTTPHandler para enviar los registros a un servidor web mediante GET
o métodos POST

El controlador de registro utiliza el formateador de registro para agregar más información de contexto a los registros y el nivel de registro para filtrar los datos de registro.

Trabajando con el módulo de registro

En esta sección, discutiremos cómo usar el módulo de registro con ejemplos de código.

Comenzaremos con las opciones básicas de registro y las llevaremos a un nivel avanzado de manera gradual.

Uso del registrador predeterminado

Sin crear una instancia de ninguna clase de registrador, ya hay un registrador predeterminado disponible en Python. El registrador predeterminado, también conocido como **registrador raíz**, se puede usar importando el módulo de registro y usando sus métodos para enviar eventos de registro. El siguiente fragmento de código muestra el uso del registrador raíz para capturar eventos de registro:

```
#registro1.py  
registro de importación  
  
logging.debug("Este es un mensaje de depuración")
```

142 bibliotecas de Python para programación avanzada

```
logging.warning("Este es un mensaje de advertencia")
logging.info("Este es un mensaje de información")
```

Los métodos de depuración, advertencia e información se utilizan para enviar eventos de registro al registrador según su gravedad. El nivel de registro predeterminado para este registrador se establece en ADVERTENCIA y la salida predeterminada se establece en stderr, lo que significa que todos los mensajes irán a la consola o terminal únicamente. Esta configuración bloqueará los mensajes DEBUG e INFO que se mostrarán en la salida de la consola, que serán los siguientes:

ADVERTENCIA:raíz:Este es un mensaje de advertencia

El nivel del registrador raíz se puede cambiar agregando la siguiente línea después de la declaración de importación :

```
registro.basicConfig(nivel=registro.DEBUG)
```

Después de cambiar el nivel de registro a DEBUG, la salida de la consola ahora mostrará todos los mensajes de registro:

DEBUG:root:Este es un mensaje de depuración

ADVERTENCIA:raíz:Este es un mensaje de advertencia

INFO:root:Este es un mensaje de información

Aunque discutimos el registrador raíz o predeterminado en esta subsección, no se recomienda usarlo para otros fines que no sean el registro básico. Como práctica recomendada, debemos crear un nuevo registrador con un nombre, que analizaremos en los siguientes ejemplos de código.

Uso de un registrador con nombre

Podemos crear un registrador separado con su propio nombre y posiblemente con su propio nivel de registro, controladores y formateadores. El siguiente fragmento de código es un ejemplo de cómo crear un registrador con un nombre personalizado y también usar un nivel de registro diferente al del registrador raíz:

```
#registro2.py
registro de importación
registrador1 = registro.getLogger("mi_registrador")
registro.basicConfig()
logger1.setLevel(registro.INFO)
logger1.warning("Este es un mensaje de advertencia")
logger1.info("Este es un mensaje de información")
```

```
logger1.debug("Este es un mensaje de depuración")
logging.info("Este es un mensaje de información")
```

Cuando creamos una instancia de registrador usando el método getLogger con un nombre de cadena o usando el nombre del módulo (usando la variable global `__name__`), solo se administra una instancia para un nombre. Esto significa que si intentamos usar el método getLogger con el mismo nombre en cualquier parte de la aplicación, el intérprete de Python verificará si ya existe una instancia creada para este nombre. Si ya hay uno creado, devolverá la misma instancia.

Después de crear una instancia de registrador, debemos realizar una llamada al registrador raíz (basicConfig()) para proporcionar un controlador y un formateador para nuestro registrador. Sin ninguna configuración de controlador, obtendremos un controlador interno como último recurso, que solo generará mensajes sin formato y el nivel de registro será ADVERTENCIA , independientemente del nivel de registro que establezcamos para nuestro registrador. La salida de la consola de este fragmento de código se muestra a continuación y es como se esperaba:

```
ADVERTENCIA: my_logger: Este es un mensaje de advertencia
INFO:my_logger:Este es un mensaje de información
```

También es importante tener en cuenta lo siguiente:

- Establecimos el nivel de registro de nuestro registrador en INFO y pudimos registrar una advertencia y mensajes de información , pero no el mensaje de depuración.
- Cuando usamos el registrador raíz (usando la instancia de registro), no pudimos enviar el mensaje de información . Esto se debió a que el registrador raíz todavía estaba usando el nivel de registro predeterminado, que es ADVERTENCIA.

Uso de un registrador con un controlador incorporado y un formateador personalizado

Podemos crear un objeto registrador utilizando un controlador integrado pero con un formateador personalizado. En este caso, el objeto del controlador puede usar un objeto de formateador personalizado y el objeto del controlador se puede agregar al objeto del registrador como su controlador antes de que comencemos a usar el registrador para cualquier evento de registro. Aquí hay un fragmento de código para ilustrar cómo crear un controlador y un formateador mediante programación y luego agregar el controlador al registrador:

```
#registro3.py
registro de importación
registrador = registro.getLogger('mi_registrador')
mi_manejador = registro.StreamHandler()
mi_formateador = logging.Formatter("%(asctime)s - \
'%(nombre)s - %(nombrenivel)s - %(mensaje)s'")
```

144 bibliotecas de Python para programación avanzada

```
mi_manejador.setFormatter(mi_formateador)
logger.addHandler(mi_manejador)
regrador.setLevel(registro.INFO)
logger.warning("Este es un mensaje de advertencia")
logger.info("Este es un mensaje de información")
logger.debug("Este es un mensaje de depuración")
```

Podemos crear un registrador con la misma configuración usando el método basicConfig también con los argumentos apropiados. El siguiente fragmento de código es una versión revisada de logging3.py con la configuración basicConfig :

```
#registro3A.py
registro de importación
regrador = registro.getLogger('mi_regrador')
logging.basicConfig(handlers=[logging.StreamHandler()],
format="%(asctime)s - %(nombre)s -
"%(nombre de nivel)s - %(mensaje)s",
nivel=registro.INFO)

logger.warning("Este es un mensaje de advertencia")
logger.info("Este es un mensaje de información")
logger.debug("Este es un mensaje de depuración")
```

Hasta ahora, hemos cubierto casos en los que usamos clases y objetos integrados para configurar nuestros registradores. A continuación, configuraremos un registrador con controladores y formateadores personalizados.

Uso de un registrador con un controlador de archivos

El controlador de registro envía los mensajes de registro a su destino final. De manera predeterminada, cada registrador está configurado para enviar mensajes de registro a la consola o terminal asociada con el programa en ejecución. Pero esto se puede cambiar configurando un registrador con un nuevo controlador con un destino diferente. Se puede crear un controlador de archivos utilizando uno de los dos enfoques que ya discutimos en la subsección anterior. En esta sección, utilizaremos un tercer enfoque para crear un controlador de archivos automáticamente con el método basicConfig proporcionando el nombre del archivo como un atributo de este método. Esto se muestra en el siguiente fragmento de código:

```
#registro4.py
registro de importación
```

```
logging.basicConfig(filename='logs/logging4.log'  
,nivel=registro.DEBUG)  
registrador = registro.getLogger('mi_registrador')  
registrador.setLevel(registro.INFO)  
logger.warning("Este es un mensaje de advertencia")  
logger.info("Este es un mensaje de información")  
logger.debug("Este es un mensaje de depuración")
```

Esto generará mensajes de registro en el archivo que especificamos con el método basicConfig y según el nivel de registro, que se establece en INFO.

Uso de un registrador con múltiples controladores mediante programación

La creación de un registrador con múltiples controladores es bastante sencillo y se puede lograr usando el método basicConfig o adjuntando controladores manualmente a un registrador. Con fines ilustrativos, revisaremos nuestro ejemplo de código logging3.py para hacer lo siguiente:

1. Crearemos dos controladores (uno para la salida de la consola y otro para la salida del archivo) que son instancias de las clases streamHandler y fileHandler .
2. Crearemos dos formateadores separados, uno para cada controlador. no incluiremos la información de tiempo para el formateador del controlador de la consola.
3. Estableceremos niveles de registro separados para los dos controladores. Es importante comprender que el nivel de registro en el nivel del controlador no puede anular el controlador del nivel raíz.

Aquí está el ejemplo de código completo:

```
#registro5.py  
registro de importación  
registrador = registro.getLogger('mi_registrador')  
registrador.setLevel(registro.DEBUG)  
controlador_consola = registro.StreamHandler()  
manejador_de_archivos = logging.FileHandler("logs/logging5.log")  
#establecimiento de niveles de registro en el nivel del controlador  
controlador_consola.setLevel(registro.DEBUG)  
manejador_archivo.setLevel(registro.INFO)  
  
#creando un formateador separado para dos manejadores  
console_formatter = logging.Formatter(
```

```
'%(nombre)s - %(nombrenivel)s - %(mensaje)s')
file_formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s
- %(message)s')

#añadiendo formateadores al controlador
controlador_consola.setFormatter(formateador_consola)
manejador_archivo.setFormatter(formateador_archivo)

#agregar controladores al registrador
registrador.addHandler(console_handler)
registrador.addHandler(file_handler)

logger.error("Este es un mensaje de error")
logger.warning("Este es un mensaje de advertencia")
logger.info("Este es un mensaje de información")
logger.debug("Este es un mensaje de depuración")
```

Aunque establecemos diferentes niveles de registro para los dos controladores, que son INFO y DEBUG, solo serán efectivos si el nivel de registro del registrador tiene un valor más bajo (el valor predeterminado es ADVERTENCIA). Es por eso que tenemos que establecer el nivel de registro de nuestro registrador en DEBUG al comienzo del programa. El nivel de registro en el nivel del controlador puede ser DEBUG o cualquier nivel superior. Este es un punto muy importante a tener en cuenta al diseñar una estrategia de registro para su aplicación.

En el ejemplo de código compartido en esta sección, básicamente configuramos el registrador mediante programación. En la siguiente sección, trabajaremos en cómo configurar un registrador a través de un archivo de configuración.

Configuración de un registrador con múltiples controladores mediante un archivo de configuración

Configurar un registrador programáticamente es atractivo pero no práctico para entornos de producción. En entornos de producción, tenemos que establecer la configuración del registrador de manera diferente en comparación con la configuración de desarrollo y, a veces, tenemos que mejorar el nivel de registro para solucionar problemas que encontramos solo en entornos en vivo. Es por eso que tenemos la opción de proporcionar la configuración del registrador a través de un archivo que es fácil de cambiar según el entorno de destino. El archivo de configuración para un registrador se puede escribir usando **JSON (notación de objetos de JavaScript)** o **YAML (otro lenguaje de marcado más)** o como una lista de pares clave:valor en un archivo .conf . Con fines ilustrativos, demostrarímos la configuración del registrador usando un archivo YAML, que es exactamente lo mismo que logramos mediante programación en la sección anterior. El archivo YAML completo y el código de Python es el siguiente:

El siguiente es el archivo de configuración YAML:

```
versión 1

formateadores:

formateador_consola: formato:
'%(nombre)s - %(nombrenivel)s - %(mensaje)s'

formateador_archivo:
formato: '%(asctime)s - %(nombre)s - %(nombrenivel)s - %(mensaje)s'

manipuladores:

controlador_consola: clase:
registro.StreamHandler
nivel: DEPURAR

formateador: flujo de formateador de consola: ext://
sys.stdout

manejador_archivo:
clase: logging.FileHandler
nivel: INFORMACIÓN

formateador: file_formatter nombre de archivo: logs/
logging6.log loggers: my_logger:

nivel: DEPURAR

manejadores: [console_handler, file_handler] propagar: no

raíz:
nivel: ERROR

controladores: [console_handler]
```

El siguiente es el programa Python que usa el archivo YAML para configurar el registrador:

```
#logging6.py
import registro importar
logging.config importar yaml

con open('logging6.conf.yaml', 'r') como f:
```

148 bibliotecas de Python para programación avanzada

```
config = yaml.safe_load(f.read())
registro.config.dictConfig(config)

registrador = registro.getLogger('mi_registrador')

logger.error("Este es un mensaje de error")
logger.warning("Este es un mensaje de advertencia")
logger.info("Este es un mensaje de información")
logger.debug("Este es un mensaje de depuración")
```

Para cargar la configuración desde un archivo, usamos el método dictConfig en lugar del método basicConfig . El resultado de la configuración del registrador basado en YAML es exactamente el mismo que logramos con las declaraciones de Python. Hay otras opciones de configuración adicionales disponibles para un registrador con todas las funciones.

En esta sección, presentamos diferentes escenarios de configuración de una o más instancias de registrador para una aplicación. A continuación, discutiremos qué tipo de eventos registrar y cuáles no.

Qué registrar y qué no registrar

Siempre hay un debate sobre qué información debemos registrar y qué no registrar.

Como práctica recomendada, la siguiente información es importante para el registro:

- Una aplicación debe registrar todos los errores y excepciones y la forma más adecuada es registrar estos eventos en el módulo de origen.
- Las excepciones que se manejan con un flujo de código alternativo se pueden registrar como advertencias.
- Para fines de depuración, la entrada y salida de una función es información útil para iniciar sesión
- También es útil registrar puntos de decisión en el código porque puede ser útil para solucionar problemas.
- Las actividades y acciones de los usuarios, especialmente las relacionadas con el acceso a ciertos recursos y funciones de la aplicación, son importantes para registrar con fines de seguridad y auditoría.

Al registrar mensajes, la información de contexto también es importante, que incluye la hora, el nombre del registrador, el nombre del módulo, el nombre de la función, el número de línea, el nivel de registro, etc. Esta información es crítica para identificar el análisis de la causa de la ruta.

Una discusión de seguimiento sobre este tema es qué no capturar para el registro. No debemos registrar ninguna información confidencial, como identificación de usuario, dirección de correo electrónico, contraseñas y cualquier dato privado y confidencial. También debemos evitar registrar cualquier dato de registro personal y comercial, como registros de salud, detalles de documentos emitidos por el gobierno y detalles de la organización.

Resumen

En este capítulo, discutimos una variedad de temas que requieren el uso de módulos y bibliotecas avanzados de Python. Comenzamos actualizando nuestro conocimiento sobre contenedores de datos en Python. A continuación, aprendimos a usar y crear iteradores para objetos iterables. También cubrimos los generadores, que son más eficientes y fáciles de construir y usar que los iteradores.

Discutimos cómo abrir y leer archivos y cómo escribir en archivos, seguido del uso de un administrador de contexto con archivos. En el siguiente tema, discutimos cómo manejar errores y excepciones en Python, cómo generar excepciones a través de la programación y cómo definir excepciones personalizadas. El manejo de excepciones es fundamental para cualquier aplicación Python decente. En la última sección, cubrimos cómo configurar el marco de registro en Python usando diferentes opciones para controladores y formateadores.

Después de leer este capítulo, ahora sabe cómo crear sus propios iteradores y diseñar funciones generadoras para iterar en cualquier objeto iterable, y cómo manejar archivos, errores y excepciones en Python. También aprendió cómo configurar registradores con uno o más controladores para administrar el registro de una aplicación utilizando diferentes niveles de registro.

Las habilidades que ha aprendido en este capítulo son clave para crear aplicaciones comerciales o de código abierto.

En el próximo capítulo, cambiaremos nuestro enfoque a cómo construir y automatizar pruebas unitarias y pruebas de integración.

Preguntas

1. ¿Cuál es la diferencia entre una lista y una tupla?
2. ¿Qué declaración de Python se usará siempre cuando se trabaje con un administrador de contexto?
3. ¿Cuál es el uso de la sentencia else con el bloque try-except ?
4. Es mejor usar generadores que iteradores. ¿Por qué?
5. ¿Cuál es el uso de múltiples controladores para el registro?

Otras lecturas

- Python fluido por Luciano Ramalho
- Guía avanzada para la programación de Python 3 por John Hunt
- La biblioteca estándar de Python 3 por ejemplo por Doug Hellmann
- Documentación de Python 3.7.10 (<https://docs.python.org/3.7/>)
- Para obtener más información sobre las opciones adicionales disponibles para configurar un registrador, puede consultar la documentación oficial de Python en <https://docs.python.org/3/biblioteca/registro.config.html>

respuestas

1. Una lista es un objeto mutable mientras que una tupla es inmutable. Esto significa que podemos actualizar una lista después de crearla. Esto no es cierto para las tuplas.
2. La declaración `with` se usa con un administrador de contexto.
3. El bloque `else` se ejecuta solo cuando se ejecuta el código en el bloque `try` sin ningún error. Se puede codificar una acción de seguimiento en el bloque `else` una vez que la funcionalidad principal se ejecuta sin ningún problema en el bloque `try`.
4. Los generadores son eficientes en memoria y también fáciles de programar en comparación con iteradores. Una función de generador proporciona automáticamente una instancia de iterador y la siguiente implementación de función lista para usar.
5. El uso de múltiples controladores es común porque un controlador generalmente se enfoca en un tipo de destino. Si necesitamos enviar eventos de registro a varios destinos y quizás con diferentes niveles de prioridad, necesitaremos varios controladores. Además, si necesitamos registrar mensajes en varios archivos con diferentes niveles de registro, podemos crear diferentes controladores de archivos para coordinar con varios archivos.

5

Pruebas y

Automatización c

La prueba de software es el proceso de validar una aplicación o un programa según los requisitos del usuario o las especificaciones deseadas y evaluar el software para los objetivos de escalabilidad y optimización. Validar el software como un usuario real lleva mucho tiempo y no es un uso eficiente de los recursos humanos. Además, las pruebas no se realizan solo una o dos veces, sino que es un proceso continuo como parte del desarrollo de software. Para salvar la situación, se recomienda la automatización de pruebas para todo tipo de pruebas. **La automatización de pruebas** es un conjunto de programas escritos para validar el comportamiento de una aplicación usando diferentes escenarios como entrada a estos programas. Para entornos profesionales de desarrollo de software, es imprescindible que las pruebas de automatización se ejecuten cada vez que se actualiza el código fuente (también llamado **operación de confirmación**) en un repositorio central.

152 Pruebas y Automatización con Python

En este capítulo, estudiaremos diferentes enfoques para las pruebas automatizadas, y luego veremos diferentes tipos de marcos de trabajo y bibliotecas de prueba que están disponibles para las aplicaciones de Python. Luego, nos centraremos en las pruebas unitarias y veremos diferentes formas de implementar las pruebas unitarias en Python. A continuación, estudiaremos la utilidad del **desarrollo dirigido por pruebas (TDD)** y la forma correcta de implementarlo. Finalmente, nos centraremos en la **integración continua (CI)** automatizada y analizaremos los desafíos de implementarla de manera sólida y eficiente. Este capítulo lo ayudará a comprender los conceptos de pruebas automatizadas en Python en varios niveles.

Cubriremos los siguientes temas en este capítulo:

- Comprensión de varios niveles de prueba
- Trabajar con marcos de prueba de Python
- Ejecutando TDD
- Introducción de CI automatizado

Al final de este capítulo, no solo comprenderá los diferentes tipos de automatización de pruebas, sino que también podrá escribir pruebas unitarias utilizando uno de los dos marcos de prueba populares.

Requerimientos técnicos

Estos son los requisitos técnicos para este capítulo:

- Debe tener instalado Python 3.7 o posterior en su computadora.
- Debe registrar una cuenta con Test PyPI y crear un token **de interfaz de programación de aplicaciones (API)** en su cuenta.

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter05>.

Comprensión de varios niveles de prueba

Las pruebas se realizan en varios niveles según el tipo de aplicación, su nivel de complejidad y el rol del equipo que está trabajando en la aplicación. Los diferentes niveles de prueba incluyen lo siguiente:

- Examen de la unidad
- Pruebas de integración

- Pruebas del sistema
- Test de aceptación

Estos diferentes niveles de prueba se aplican en el orden que se muestra aquí:

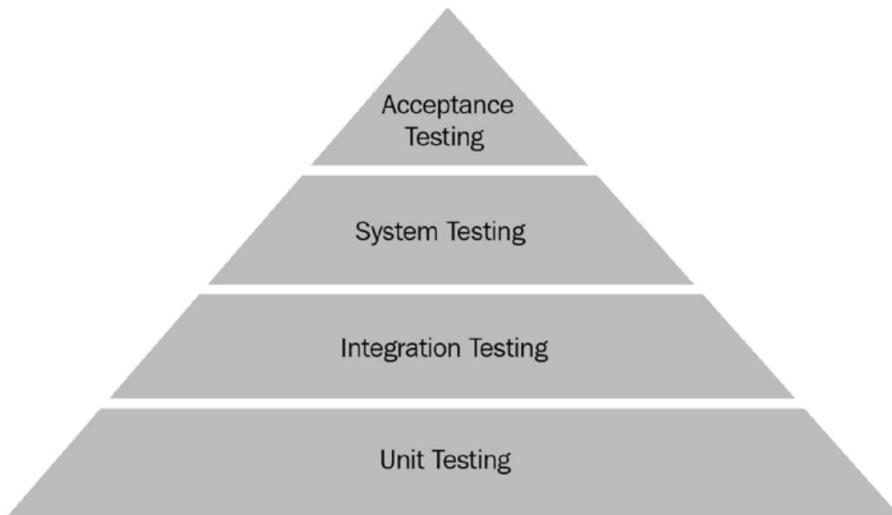


Figura 5.1 – Diferentes niveles de prueba durante el desarrollo de software

Estos niveles de prueba se describen en las siguientes subsecciones.

Examen de la unidad

La prueba unitaria es un tipo de prueba que se centra en el nivel de unidad más pequeño posible. Una unidad corresponde a una unidad de código que puede ser una función en un módulo o un método en una clase, o puede ser un módulo en una aplicación. Una prueba de unidad ejecuta una sola unidad de código de forma aislada y valida que el código funciona como se esperaba. La prueba unitaria es una técnica utilizada por los desarrolladores para identificar errores en las primeras etapas del desarrollo del código y corregirlos como parte de la primera iteración del proceso de desarrollo. En Python, las pruebas unitarias se dirigen principalmente a una clase o módulo en particular sin involucrar dependencias.

Las pruebas unitarias son desarrolladas por desarrolladores de aplicaciones y se pueden realizar en cualquier momento. Las pruebas unitarias son una especie de **pruebas de caja blanca**. Las bibliotecas y herramientas disponibles para pruebas unitarias en Python son pyunit (unittest), pytest, doctest, nose y algunas otras.

Pruebas de integración

Las pruebas de integración consisten en probar unidades individuales de un programa colectivamente en forma de grupo. La idea detrás de este tipo de pruebas es probar la combinación de diferentes funciones o módulos de una aplicación para validar las interfaces entre los componentes y el intercambio de datos entre ellos.

Las pruebas de integración generalmente las realizan los evaluadores y no los desarrolladores. Este tipo de prueba comienza después del proceso de prueba unitaria, y el enfoque de esta prueba es identificar el problema de integración cuando se usan juntos diferentes módulos o funciones. En algunos casos, las pruebas de integración requieren recursos o datos externos que pueden no ser posibles de proporcionar en un entorno de desarrollo. Esta limitación se puede gestionar mediante pruebas simuladas, que proporcionan objetos simulados de reemplazo para dependencias internas o externas. Los objetos simulados simulan el comportamiento de las dependencias reales. Ejemplos de pruebas simuladas pueden ser enviar un correo electrónico o realizar un pago con una tarjeta de crédito.

Las pruebas de integración son una especie de **pruebas de caja negra**. Las bibliotecas y las herramientas utilizadas para las pruebas de integración son prácticamente las mismas que para las pruebas unitarias, con la diferencia de que los límites de las pruebas se amplían más para incluir varias unidades en una sola prueba.

Pruebas del sistema

Los límites de las pruebas del sistema se extienden aún más al nivel del sistema, que puede ser un módulo completo o una aplicación. Este tipo de prueba valida la funcionalidad de la aplicación desde una perspectiva de **extremo a extremo (E2E)**.

Las pruebas del sistema también las desarrollan los evaluadores, pero después de completar el proceso de prueba de integración. Podemos decir que las pruebas de integración son un requisito previo para las pruebas del sistema; de lo contrario, se repetirá mucho esfuerzo mientras se realizan las pruebas del sistema. Las pruebas del sistema pueden identificar problemas potenciales, pero no señalan la ubicación del problema. La causa raíz exacta del problema generalmente se identifica mediante pruebas de integración o incluso agregando más pruebas unitarias.

La prueba del sistema también es un tipo de prueba de caja negra y puede aprovechar las mismas bibliotecas que están disponibles para la prueba de integración.

Test de aceptación

La prueba de aceptación es la prueba del usuario final antes de aceptar el software para el uso diario.

Las pruebas de aceptación no suelen ser candidatas para las pruebas de automatización, pero vale la pena usar la automatización para las pruebas de aceptación en situaciones en las que los usuarios de la aplicación tienen que interactuar con el producto mediante una API. Esta prueba también se llama **prueba de aceptación del usuario**.

(UAT). Este tipo de prueba se puede confundir fácilmente con la prueba del sistema, pero es diferente porque garantiza la usabilidad de la aplicación desde el punto de vista de un usuario real. También hay otros dos tipos de pruebas de aceptación: pruebas de aceptación de **fábrica (FAT)** y **pruebas de aceptación operativas (OAT)**.

El primero es más popular desde el punto de vista del hardware, y el segundo lo realizan los equipos de operación, que son los responsables de utilizar el producto en entornos de producción.

Además, también escuchamos sobre las pruebas **alfa** y **beta**. Estas también son pruebas a nivel de usuario. enfoques y no están destinados a la automatización de pruebas. Las pruebas alfa las realizan los desarrolladores y el personal interno para emular el comportamiento real del usuario. Las pruebas beta las realizan los clientes o usuarios reales para obtener comentarios tempranos antes de declarar la **disponibilidad general (GA)** del software.

También usamos el término **prueba de regresión** en el desarrollo de software. Esto es básicamente la ejecución de pruebas cada vez que hacemos un cambio en el código fuente o cualquier cambio de dependencia interno o externo. Esta práctica garantiza que nuestro producto funcione de la misma manera que antes de realizar el cambio. Dado que las pruebas de regresión se repiten muchas veces, la automatización de las pruebas es imprescindible para este tipo de pruebas.

En la siguiente sección, investigaremos cómo crear casos de prueba utilizando los marcos de prueba en Python.

Trabajar con marcos de pruebas de Python

Python viene con bibliotecas estándar y de terceros para la automatización de pruebas. Los marcos más populares se enumeran aquí:

- pytest
- prueba unitaria
- prueba de documento
- nariz

156 Pruebas y Automatización con Python

Estos marcos se pueden usar para pruebas unitarias, así como para pruebas de integración y sistemas. En esta sección, evaluaremos dos de estos marcos: unittest, que forma parte de la biblioteca estándar de Python, y pytest, que está disponible como biblioteca externa.

El enfoque de esta evaluación estará en la construcción de casos de prueba (principalmente pruebas unitarias) usando estos dos marcos, aunque las pruebas de integración y del sistema también se pueden construir usando las mismas bibliotecas y patrones de diseño.

Antes de comenzar a escribir cualquier caso de prueba, es importante comprender qué es un caso de prueba. En el contexto de este capítulo y libro, podemos definir un caso de prueba como una forma de validar los resultados de un comportamiento particular de un código de programación según los resultados esperados. El desarrollo de un caso de prueba se puede dividir en las siguientes cuatro etapas:

1. **Organizar:** esta es una etapa en la que preparamos el entorno para nuestros casos de prueba. Esto no incluye ninguna acción o paso de validación. En la comunidad de automatización de pruebas, esta etapa se conoce más comúnmente como preparación de **dispositivos de prueba**.
2. **Actuar:** Esta es la etapa de acción que desencadena el sistema que queremos probar. Esta acción La etapa da como resultado un cambio en el comportamiento del sistema, y el cambio de estado del sistema es algo que queremos evaluar con fines de validación. Tenga en cuenta que no validamos nada en esta etapa.
3. **Afirmar:** en esta etapa, evaluamos los resultados de la etapa de actuar y validamos los resultados contra el resultado esperado. En función de esta validación, las herramientas de automatización de pruebas marcan el caso de prueba como fallido o aprobado. En la mayoría de las herramientas, esta validación se logra utilizando declaraciones o funciones de aserción integradas.
4. **Limpieza:** en esta etapa, el entorno se limpia para asegurarse de que las otras pruebas no se vean afectadas por los cambios de estado causados por la etapa de actuación.

Las etapas centrales de un caso de prueba son actuar y afirmar. Las etapas de organización y limpieza son opcionales pero muy recomendables. Estas dos etapas proporcionan principalmente accesorios de prueba de software. Un dispositivo de prueba es un tipo de equipo, dispositivo o software que proporciona un entorno para probar un dispositivo, una máquina o un software de manera consistente. El término accesorio de prueba se usa en el mismo contexto para pruebas unitarias y pruebas de integración.

Los marcos o bibliotecas de prueba proporcionan métodos auxiliares o declaraciones para facilitar la implementación de estas etapas de manera conveniente. En las siguientes secciones, evaluaremos los frameworks unittest y pytest para los siguientes temas:

- Cómo construir casos de prueba de nivel básico para las etapas de actuar y afirmar
- Cómo construir casos de prueba con accesorios de prueba
- Cómo construir casos de prueba para la validación de excepciones y errores

- Cómo ejecutar casos de prueba de forma masiva
- Cómo incluir y excluir casos de prueba en ejecución

Estos temas no solo cubren el desarrollo de una variedad de casos de prueba, sino que también incluyen diferentes formas de ejecutarlos. Comenzaremos nuestra evaluación con el marco unittest .

Trabajando con el framework unittest

Antes de comenzar a discutir ejemplos prácticos con el marco o la biblioteca unittest , es importante presentar algunos términos y nombres de métodos tradicionales relacionados con las pruebas unitarias y, en particular, con la biblioteca unittest . Esta terminología se usa más o menos en todos los marcos de prueba y se describe aquí:

- **Caso de prueba:** Una prueba o caso de prueba o método de prueba es un conjunto de instrucciones de código que se basan en una comparación de la condición actual con las condiciones posteriores a la ejecución después de ejecutar una unidad de código de aplicación.
- **Conjunto de pruebas:** un conjunto de pruebas es una colección de casos de prueba que pueden tener condiciones previas, pasos de inicialización y quizás los mismos pasos de limpieza. Esto fomenta la reutilización del código de automatización de pruebas y reduce el tiempo de ejecución. • **Test runner:** Esta es una aplicación de Python que ejecuta las pruebas (pruebas unitarias), valida todas las afirmaciones definidas en el código y nos devuelve los resultados como un éxito o un fracaso.
- **Configuración:** este es un método especial en un conjunto de pruebas que se ejecutará antes de cada caso de prueba.
- **setupClass:** este es un método especial en un conjunto de pruebas que se ejecutará solo una vez al comienzo de la ejecución de las pruebas en un conjunto de pruebas.
- **desmontaje:** este es otro método especial en un conjunto de pruebas que se ejecuta después de completar cada prueba, independientemente de si la prueba pasa o falla.
- **teardownClass:** este es otro método especial en un conjunto de pruebas que se ejecuta solo una vez cuando se completan todas las pruebas en un conjunto.

158 Pruebas y Automatización con Python

Para escribir casos de prueba utilizando la biblioteca unittest , debemos implementar los casos de prueba como métodos de instancia de una clase que debe heredarse de la clase base TestCase .

La clase TestCase viene con varios métodos para facilitar la escritura y la ejecución de los casos de prueba. Estos métodos se agrupan en tres categorías, que se analizan a continuación:

- **Métodos relacionados con la ejecución:** los métodos incluidos en esta categoría son setUp, tearDown, setupClass, teardownClass, run, skipTest, skipTestIf, subTest y debug. El ejecutor de pruebas utiliza estas pruebas para ejecutar un fragmento de código antes o después de un caso de prueba o ejecutar un conjunto de casos de prueba, ejecutar una prueba, omitir una prueba o ejecutar cualquier bloque de código como subprueba. En nuestra clase de implementación de caso de prueba, podemos anular estos métodos. Los detalles exactos de estos métodos están disponibles como parte de la documentación de Python en <https://docs.python.org/3/library/unittest.html>.
- Métodos de **validación** (métodos de afirmación): estos métodos se utilizan para implementar pruebas casos para verificar las condiciones de éxito o falla e informar el éxito o las fallas para un caso de prueba automáticamente. El nombre de estos métodos generalmente comienza con un prefijo de aserción. La lista de métodos de afirmación es muy larga. Proporcionamos métodos de afirmación de uso común aquí como ejemplos:

Method name	Evaluating condition
assertEqual (x, y)	Check if x is equal to y
assertTrue (x)	Check if x is Boolean true
assertFalse (x)	Check if x is Boolean false
assertNotEqual (x)	Check if x is not equal to y
assertIn (a, c)	Check if a is in collection c
assertNotIn (a, c)	Check if a is not in collection c
assertIs (x, y)	Check if x is y
assertIsNot (x, y)	Check if x is not y
assertIsNone (x)	Check if x is none

Figura 5.2 – Algunos ejemplos de métodos de afirmación de la clase TestCase

- **Métodos y atributos relacionados con la información adicional:** Estos métodos y

Los atributos proporcionan información adicional relacionada con los casos de prueba que se van a ejecutar. o ya ejecutado. Algunos de los métodos y atributos clave en esta categoría se resumen a continuación:

- a) failException: este atributo proporciona una excepción provocada por un método de prueba. Esta excepción se puede usar como una superclase para definir una excepción de falla personalizada con información adicional.
- b) longMessage: este atributo determina qué hacer con un mensaje personalizado que se pasa como argumento con un método de afirmación . Si el valor de este atributo se establece en True, el mensaje se adjunta al mensaje de error estándar. Si este atributo se establece en falso, un mensaje personalizado reemplaza el mensaje estándar.
- c) countTestCases(): este método devuelve el número de pruebas adjuntas a un objeto de prueba.
- d) shortDescription(): este método devuelve una descripción de un método de prueba si hay alguna descripción agregada, utilizando una cadena de documentación.

Hemos revisado los métodos principales de la clase TestCase en esta sección. En la siguiente sección, exploraremos cómo usar unittest para crear pruebas unitarias para un módulo de muestra o una aplicación.

Creación de casos de prueba utilizando la clase base TestCase

La biblioteca unittest es un marco de prueba estándar de Python que está muy inspirado en el marco **JUnit** , un marco de prueba popular en la comunidad de Java. Las pruebas unitarias se escriben en archivos de Python separados y se recomienda hacer que los archivos formen parte del proyecto principal. Como discutimos en el Capítulo 2, Uso de la modularización para manejar proyectos complejos, en la sección Creación de un paquete, las pautas de **Python Packaging Authority (PyPA)** recomiendan tener una carpeta separada para las pruebas al crear paquetes para un proyecto o una biblioteca. En nuestros ejemplos de código para esta sección, seguiremos una estructura similar a la que se muestra aquí:

```
Nombre del proyecto
|-- origen
| -- __init__.py
| -- myadd/myadd.py |-- pruebas
|
| -- __init__.py
| -- tests_myadd/test_myadd1.py | -- tests_myadd/
| test_myadd2.py
|-- LÉAME.md
```

160 Pruebas y Automatización con Python

En nuestro primer ejemplo de código, crearemos un conjunto de pruebas para la función de agregar en myadd.py módulo, de la siguiente manera:

```
# myadd.py con suma de dos números
def suma(x, y):
    """Esta función suma dos números"""
    volver x + y
```

Es importante comprender que puede haber más de un caso de prueba para la misma pieza de código (una función de suma , en nuestro caso).

Para la función de suma , implementamos cuatro casos de prueba variando los valores de los parámetros de entrada. El siguiente es un ejemplo de código con cuatro casos de prueba para la función de suma , de la siguiente manera:

#test_myadd1.py conjunto de pruebas para la función myadd

```
prueba unitaria de importación
desde myunittest.src.myadd.myadd importar añadir

clase MyAddTestSuite(unittest.TestCase):

    def test_add1(auto):
        """ caso de prueba para validar dos números positivos"""
        self.assertEqual(15, add(10 , 5), "debería ser 15")

    def test_add2(auto):
        """ caso de prueba para validar positivo y negativo \
números"""
        self.assertEqual(5, add(10 , -5), "debería ser 5")

    def test_add3(auto):
        """ caso de prueba para validar positivo y negativo \
números"""
        self.assertEqual(-5, add(-10 , 5), "debería ser -5")

    def test_add4(auto):
        """ caso de prueba para validar dos números negativos"""
        self.assertEqual(-15, add(-10 , -5), "debería ser -15")

    si __nombre__ == '__principal__':
        unittest.principal()
```

Todos los puntos clave del conjunto de pruebas anterior se analizan a continuación, de la siguiente manera:

- Para implementar pruebas unitarias utilizando el marco unittest , necesitamos importar una biblioteca estándar con el mismo nombre, unittest.
- Necesitamos importar el módulo o módulos que queremos probar en nuestro banco de pruebas. En este caso, importamos la función de agregar desde el módulo myadd.py utilizando el enfoque de importación relativa (consulte la sección Importación de módulos del Capítulo 2, Uso de la modularización para manejar proyectos complejos, para obtener más información)
- Implementaremos una clase de conjunto de pruebas que se hereda de unittest.

Clase base de caso de prueba. Los casos de prueba se implementan en la subclase, que en este caso es la clase MyAddTestSuite . El constructor de la clase unittest.TestCase puede tomar un nombre de método como entrada que se puede usar para ejecutar los casos de prueba. De forma predeterminada, hay un método runTest ya implementado que utiliza el ejecutor de pruebas para ejecutar las pruebas. En la mayoría de los casos, no necesitamos proporcionar nuestro propio método o volver a implementar el método runTest .
- Para implementar un caso de prueba, necesitamos escribir un método que comience con la prueba prefijo y va seguido de un guión bajo. Esto ayuda al corredor de pruebas a buscar los casos de prueba que se van a ejecutar. Usando esta convención de nomenclatura, agregamos cuatro métodos para nuestro conjunto de pruebas.
- En cada método de caso de prueba, usamos un método assertEquals especial , que es disponible desde la clase base. Este método representa la etapa de aserción de un caso de prueba y se utiliza para decidir si nuestra prueba se declarará como aprobada o fallida. El primer parámetro de este método son los resultados esperados de la prueba unitaria, el segundo parámetro es el valor que obtenemos después de ejecutar el código bajo prueba, y el tercer parámetro (opcional) es el mensaje que se proporcionará en el informe en caso de que la prueba ha fallado.
- Al final del conjunto de pruebas, agregamos el método unittest.main para activar el ejecutor de pruebas para ejecutar el método runTest , lo que facilita la ejecución de las pruebas sin usar los comandos en la consola. Este método principal (un TestProgram clase bajo el capó) primero descubrirá todas las pruebas que se ejecutarán y luego ejecutará las pruebas.

Nota IMPORTANTE

Las pruebas unitarias se pueden ejecutar mediante un comando como Python -m unittest <conjunto de pruebas o módulo>, pero los ejemplos de código que proporcionamos en este capítulo supondrán que estamos ejecutando los casos de prueba mediante el **entorno de desarrollo integrado (IDE) de PyCharm**.

A continuación, construiremos el siguiente nivel de casos de prueba utilizando los accesorios de prueba.

162 Pruebas y Automatización con Python

Creación de casos de prueba con dispositivos de prueba

Hemos discutido los métodos de configuración y desmontaje que ejecutan automáticamente los ejecutores de pruebas antes y después de ejecutar un caso de prueba. Estos métodos (junto con setUpClass y los métodos tearDownClass) proporcionan los accesorios de prueba y son útiles para implementar las pruebas unitarias de manera eficiente.

Primero, revisaremos la implementación de nuestra función de suma . En la nueva implementación, haremos que esta unidad de código forme parte de la clase MyAdd . También estamos manejando la situación lanzando una excepción TypeError en caso de que los argumentos de entrada no sean válidos.

El siguiente es el fragmento de código completo con el nuevo método de agregar :

```
# myadd2.py es una clase con el método de sumar dos números
clase MiAregar:
    def suma(self, x, y):
        """Esta función suma dos números"""
        if (no es una instancia (x, (int, float))) | \
            (no es una instancia (y, (int, float))):
            aumentar TypeError("solo se permiten números")
        volver x + y
```

En la sección anterior, construimos casos de prueba utilizando solo la etapa de actuar y la etapa de afirmación. En esta sección, revisaremos el ejemplo de código anterior agregando setUp y tearDown métodos. El siguiente es el conjunto de pruebas para esta clase myAdd , de la siguiente manera:

```
#test_myadd2.py conjunto de pruebas para el método de clase myadd2
prueba unitaria de importación
de myunittest.src.myadd.myadd2 importar MyAdd

clase MyAddTestSuite(unittest.TestCase):
    def configurar(auto):
        self.myadd = MyAdd()

    def tearDown(self):
        del (self.myadd)

    def test_add1(auto):
        """ caso de prueba para validar dos números positivos"""
```

```
self.assertEqual(15, self.myadd.add(10 , 5), \
"debería ser 15")

def test_add2(auto):
    Caso de prueba """ para validar positivo y negativo
    números"""

    self.assertEqual(5, self.myadd.add(10 , -5), \
"debe ser 5")

#test_add3 y test_add4 se omiten porque son muy \
igual que test_add1 y test_add2
```

En este conjunto de pruebas, agregamos o cambiamos lo siguiente:

- Agregamos un método de configuración en el que creamos una nueva instancia de la clase MyAdd y guardamos su referencia como un atributo de instancia. Esto significa que crearemos una nueva instancia de la clase MyAdd antes de ejecutar cualquier caso de prueba. Esto puede no ser ideal para este conjunto de pruebas, ya que un mejor enfoque podría ser usar el método setUpClass y crear una sola instancia de la clase MyAdd para todo el conjunto de pruebas, pero lo hemos implementado de esta manera con fines ilustrativos.
- También agregamos un método de desmontaje . Para demostrar cómo implementarlo, simplemente llamó al destructor (usando la función del) en la instancia de MyAdd que creamos en el método de configuración . Al igual que con el método setUp , el método tearDown
El método se ejecuta después de cada caso de prueba. Si tenemos la intención de utilizar el setUpClass método, hay un método equivalente para desmontaje, que es tearDownClass.

En la siguiente sección, presentaremos ejemplos de código que crearán casos de prueba para manejar una excepción TypeError .

Creación de casos de prueba con manejo de errores

En los ejemplos de código anteriores, solo comparamos los resultados del caso de prueba con los resultados esperados. resultados. No consideraremos ningún manejo de excepción, como cuál sería el comportamiento de nuestro programa si se pasaran los tipos de argumentos incorrectos como entrada a nuestra función de agregar . Las pruebas unitarias también deben cubrir estos aspectos de la programación.

164 Pruebas y Automatización con Python

En el siguiente ejemplo de código, crearemos casos de prueba para manejar errores o excepciones que se esperan de una unidad de código. Para este ejemplo, usaremos la misma función de agregar , que lanza una excepción TypeError si el argumento no es un número. Los casos de prueba se construirán pasando argumentos no numéricos a la función de suma . El siguiente fragmento de código muestra los casos de prueba:

```
#test_myadd3.py conjunto de pruebas para el método de clase myadd2 para validar errores

prueba unitaria de importación
de myunittest.src.myadd.myadd2 importar MyAdd

clase MyAddTestSuite(unittest.TestCase):
    def configurar(auto):
        self.myadd = MyAdd()

    def prueba_tipoerror1(uno mismo):
        """ caso de prueba para verificar si podemos manejar no \
            entrada de número"""
        self.assertRaises(TypeError, self.myadd.add, \
            'un' , -5)

    def test_typeerror2(auto):
        """ caso de prueba para verificar si podemos manejar no \
            entrada de número"""
        self.assertRaises(TypeError, self.myadd.add, \
            'un' , 'b')
```

En el fragmento de código anterior, agregamos dos casos de prueba adicionales a test_add3.py módulo. Estos casos de prueba utilizan el método assertRaises para validar si se lanza o no un tipo particular de excepción. En nuestros casos de prueba, usamos una sola letra (a) o dos letras (a yb) como argumentos para los dos casos de prueba. En ambos casos, esperamos que se produzca la excepción prevista (TypeError) . Es importante tener en cuenta los argumentos del método assertRaises . Este método espera solo el nombre del método o función como segundo argumento. Los parámetros del método o función deben pasarse por separado como argumentos de la función assertRaises .

Hasta ahora, hemos ejecutado varios casos de prueba en un solo conjunto de pruebas. En la siguiente sección, discutiremos cómo podemos ejecutar varios conjuntos de pruebas simultáneamente, utilizando la línea de comandos y también mediante programación.

Ejecución de múltiples conjuntos de pruebas

A medida que creamos casos de prueba para cada unidad de código, la cantidad de casos de prueba (casos de prueba unitaria) crece muy rápidamente. La idea de usar suites de prueba es traer modularidad al desarrollo de casos de prueba. Los conjuntos de pruebas también facilitan el mantenimiento y la ampliación de los casos de prueba a medida que agregamos más funciones a una aplicación. El siguiente aspecto que nos viene a la mente es cómo ejecutar múltiples conjuntos de pruebas a través de un script maestro o un flujo de trabajo. Las herramientas de CI, como Jenkins, brindan dicha funcionalidad lista para usar. Los marcos de prueba como unittest, nose o pytest también brindan características similares.

En esta sección, construiremos una aplicación de calculadora simple (una clase MyCalc) con métodos de suma, resta, multiplicación y división . Más tarde, agregaremos un conjunto de pruebas para cada método en esta clase. De esta manera, agregaremos cuatro suites de prueba para esta aplicación de calculadora. Una estructura de directorios es importante para implementar los conjuntos de pruebas y los casos de prueba. Para esta aplicación, utilizaremos la siguiente estructura de directorios:

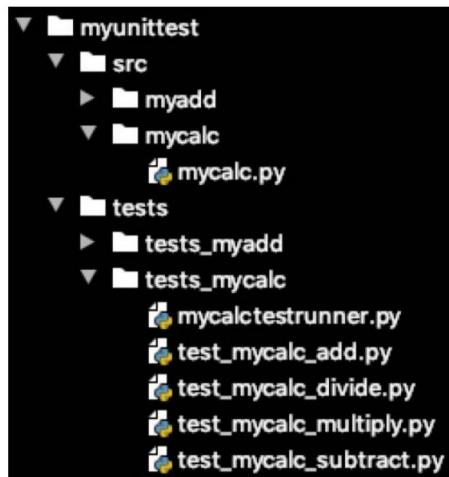


Figura 5.3 – Estructura de directorios para la aplicación mycalc y conjuntos de pruebas asociados con esta aplicación

El código de Python está escrito en el módulo mycalc.py y los archivos del conjunto de pruebas (test_mycalc*.py) se muestran a continuación. Tenga en cuenta que mostramos solo un caso de prueba en cada conjunto de pruebas en los ejemplos de código que se muestran a continuación. En realidad, habrá múltiples casos de prueba en cada conjunto de pruebas. Comenzaremos con las funciones de la calculadora en el archivo mycalc.py , de la siguiente manera:

```
# mycalc.py con funciones de suma, resta, multiplicación y división
```

```
clase MiCalc:
```

166 Pruebas y Automatización con Python

```
def suma(self, x, y):
    """Esta función suma dos números"""
    volver x + y

def restar(self, x, y):
    """Esta función resta dos números"""
    volver x - y

def multiplicar(self, x, y):
    """Esta función resta dos números"""
    volver x * y

def divide(self, x, y):
    """Esta función resta dos números"""
    volver x / y
```

A continuación, tenemos un conjunto de pruebas para probar la función de agregar en el archivo test_mycalc_add.py , como se ilustra en el siguiente fragmento de código:

```
# test_mycalc_add.py conjunto de pruebas para agregar método de clase
import unittest from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcAddTestSuite(unittest.TestCase):
    def configurar(self):
        self.calc = MiCalc()

    def test_add(self): """ caso de prueba
        para validar dos números positivos"""\n        self.assertEqual(15, self.calc.add(10, 5), \
            "debería ser 15")
```

A continuación, tenemos un conjunto de pruebas para probar la función restar en el archivo `test_mycalc_subtract.py`, como se ilustra en el siguiente fragmento de código:

```
#test_mycalc_subtract.py conjunto de pruebas para el método de clase de resta import unittest from  
myunittest.src.mycalc.mycalc import MyCalc  
  
  
clase MyCalcSubtractTestSuite(unittest.TestCase):  
def configurar(auto):  
self.calc = MiCalc()  
  
  
def test_subtract(self): """ caso de prueba  
para validar dos números positivos""" self.assertEqual(5, self.calc.subtract(10,5), \  
  
"debe ser 5")
```

A continuación, tenemos un conjunto de pruebas para probar la función de multiplicación en el archivo `test_mycalc_multiple.py`, como se ilustra en el siguiente fragmento de código:

```
#test_mycalc_multiply.py conjunto de pruebas para el método de clase de multiplicación  
import unittest from myunittest.src.mycalc.mycalc import MyCalc  
  
  
class MyCalcMultiplyTestSuite(unittest.TestCase): def setUp(self): self.calc =  
MyCalc()  
  
  
def test_multiply(self): """ caso de prueba  
para validar dos números positivos"""  
self.assertEqual(50, self.calc.multiply(10, 5), "debería  
ser 50")
```

A continuación, tenemos un conjunto de pruebas para probar la función de división en `test_mycalc_divide.py`, como se ilustra en el siguiente fragmento de código:

```
#test_mycalc_divide.py conjunto de pruebas para la prueba unitaria de importación del  
método de división de clases  
de myunittest.src.mycalc.mycalc importar MyCalc
```

168 Pruebas y Automatización con Python

```

clase MyCalcDivideTestSuite(unittest.TestCase):
    def configurar(auto):
        self.calc = MiCalc()

    def test_divide(self):
        """ caso de prueba para validar dos números positivos"""
        self.assertEqual(2, self.calc.divide(10 "debe ser 2") , 5), \

```

Tenemos el código de la aplicación de muestra y el código de los cuatro conjuntos de pruebas. El siguiente aspecto es cómo ejecutar todas las suites de prueba de una sola vez. Una forma sencilla de hacerlo es mediante el uso de la **interfaz de línea de comandos (CLI)** con la palabra clave de descubrimiento . En nuestro caso de ejemplo, ejecutaremos el siguiente comando desde la parte superior del proyecto para descubrir y ejecutar todos los casos de prueba en los cuatro conjuntos de pruebas que están disponibles en el directorio tests_mycalc :

```
python -m unittest descubre myunittest/tests/tests_mycalc
```

Este comando se ejecutará de forma recursiva, lo que significa que también puede descubrir los casos de prueba en subdirectorios. Los otros parámetros (opcionales) se pueden usar para seleccionar un conjunto de casos de prueba para su ejecución, y se describen a continuación:

- -v: para que la salida sea detallada.
- -s: Directorio de inicio para el descubrimiento de casos de prueba.
- -p: Patrón a utilizar para buscar los archivos de prueba. El valor predeterminado es test*.py, pero se puede cambiar con este parámetro.
- -t: Este es un directorio de nivel superior del proyecto. Si no se especifica, el directorio de inicio es el directorio de nivel superior

Aunque la opción de la línea de comandos para ejecutar múltiples conjuntos de pruebas es simple y poderosa, a veces necesitamos controlar la forma en que ejecutamos las pruebas seleccionadas de diferentes conjuntos de pruebas que pueden estar en diferentes ubicaciones. Aquí es donde es útil cargar y ejecutar los casos de prueba a través del código de Python. El siguiente fragmento de código es un ejemplo de cómo cargar los conjuntos de pruebas desde un nombre de clase, buscar los casos de prueba en cada uno de los conjuntos y luego ejecutarlos con el ejecutor de pruebas unittest :

```

prueba unitaria de importación
de test_mycalc_add importar MyCalcAddTestSuite
de test_mycalc_subtract importar MyCalcSubtractTestSuite
desde test_mycalc_multiply importar MyCalcMultiplyTestSuite

```

```
de test_mycalc_divide importar MyCalcDivideTestSuite

def ejecutar_mispruebas():
    test_classes = [MyCalcAddTestSuite, \
    MiCalcSubtractTestSuite, \
    MyCalcMultiplyTestSuite, MyCalcDivideTestSuite ]

    cargador = unittest.TestLoader()

    conjuntos_de_prueba = []
    para t_class en test_classes:
        suite = cargador.loadTestsFromTestCase(t_class)
        test_suites.append(suite)

    final_suite = unittest.TestSuite(test_suites)

    corredor = unittest.TextTestRunner()
    resultados = runner.run(final_suite)

    si __nombre__ == '__principal__':
        ejecutar_mispruebas()
```

En esta sección, hemos cubierto la creación de casos de prueba utilizando la biblioteca unittest . En la siguiente sección, trabajaremos con la biblioteca pytest .

Trabajando con el framework pytest

Los casos de prueba escritos con la biblioteca unittest son más fáciles de leer y administrar, especialmente si proviene de un entorno de uso de JUnit u otros marcos similares. Pero para las aplicaciones Python a gran escala, la biblioteca pytest se destaca como uno de los marcos más populares, principalmente debido a su facilidad de uso en la implementación y su capacidad de ampliación para requisitos de prueba complejos. En el caso de la biblioteca pytest , no hay ningún requisito para extender la clase de prueba unitaria desde cualquier clase base; de hecho, podemos escribir los casos de prueba sin siquiera implementar ninguna clase.

170 Pruebas y Automatización con Python

pytest es un marco de código abierto. El marco de prueba de pytest puede descubrir pruebas automáticamente, al igual que con el marco de prueba de unidad , si el nombre de archivo tiene un prefijo de prueba y este formato de descubrimiento es configurable. El marco pytest incluye el mismo nivel de funcionalidad que proporciona el marco unittest para escribir pruebas unitarias. En esta sección, nos centraremos en discutir las características que son diferentes o adicionales en el marco de trabajo de pytest .

Creación de casos de prueba sin una clase base

Para demostrar cómo escribir casos de prueba de unidad usando la biblioteca pytest , revisaremos nuestro módulo myadd2.py implementando la función de agregar sin una clase. Esta nueva función de suma agregará dos números y generará una excepción si los números no se pasan como argumentos. El código del caso de prueba que usa el marco pytest se muestra en el siguiente fragmento:

```
# myadd3.py es una clase con el método de sumar dos números

def suma(self, x, y):
    """Esta función suma dos números"""
    if (no es una instancia (x, (int, float))) | \
        (no es una instancia (y, (int, float))):
        aumentar TypeError("solo se permiten números")
    volver x + y
```

Y el módulo de casos de prueba se muestra a continuación, así:

```
#test_myadd3.py conjunto de pruebas para la función myadd

importar pytest
desde mypytest.src.myadd3 importar agregar

def test_add1():
    """ caso de prueba para validar dos números positivos"""
    afirmar sumar (10, 5) == 15

def test_add2():
    """ caso de prueba para validar dos números positivos"""
    afirmar agregar (10, -5) == 5, "debería ser 5"
```

Solo mostramos dos casos de prueba para el módulo test_myadd3.py ya que los otros casos de prueba serán similares a los primeros dos casos de prueba. Estos casos de prueba adicionales están disponibles con el código fuente de este capítulo en el directorio de GitHub. Aquí se describen un par de diferencias clave en la implementación del caso de prueba:

- No hay ningún requisito para implementar casos de prueba bajo una clase, y podemos implementar casos de prueba como métodos de clase sin heredárselos de ninguna clase base. Esta es una diferencia clave en comparación con la biblioteca unittest .
- Las declaraciones de afirmación están disponibles como una palabra clave para la validación de cualquier condición para declarar si una prueba pasó o falló. Separar las palabras clave de afirmación de la declaración condicional hace que las afirmaciones en los casos de prueba sean muy flexibles y personalizables.

También es importante mencionar que la salida de la consola y los informes son más potentes con el marco pytest . Como ejemplo, aquí se muestra la salida de la consola de ejecutar casos de prueba usando el módulo test_myadd3.py :

```
test_myadd3.py::test_add1 APROBADO [25%]
test_myadd3.py::test_add2 APROBADO [50%]
test_myadd3.py::test_add3 APROBADO [75%]
test_myadd3.py::test_add4 APROBADO [100%]
===== 4 pasaron en 0.03s =====
```

A continuación, investigaremos cómo validar los errores esperados utilizando la biblioteca pytest .

Construir casos de prueba con manejo de errores

Escribir casos de prueba para validar el lanzamiento de una excepción o error esperado es diferente en el marco pytest en comparación con escribir tales casos de prueba en el unittest marco de referencia. El marco pytest utiliza el administrador de contexto para la validación de excepciones. En nuestro módulo de prueba test_myadd3.py , ya agregamos dos casos de prueba para la validación de excepciones. A continuación se muestra un extracto del código en el módulo test_myadd3.py con los dos casos de prueba, como sigue:

```
def prueba_tipoerror1():
    """ caso de prueba para verificar si podemos manejar no número \
    aporte"""
```

172 Pruebas y Automatización con Python

```
con pytest.raises(TypeError):
    sumar('a', 5)

def prueba_tipoerror2():
    """ caso de prueba para verificar si podemos manejar no número \
    aporte"""
    con pytest.raises(TypeError, match="solo los números son \
    permitió"):
        agregar('a', 'b')
```

Para validar la excepción, usamos la función `raises` de la biblioteca `pytest` para indicar qué tipo de excepción se espera al ejecutar una cierta unidad de código (`add('a', 5)` en nuestro primer caso de prueba). En el segundo caso de prueba, usamos un argumento de coincidencia para validar el mensaje que se establece cuando se lanza una excepción.

A continuación, discutiremos cómo usar marcadores con el marco `pytest`.

Creación de casos de prueba con marcadores `pytest`

El framework `pytest` está equipado con marcadores que nos permiten adjuntar metadatos o definir diferentes categorías para nuestros casos de prueba. Estos metadatos se pueden usar para muchos propósitos, como incluir o excluir ciertos casos de prueba. Los marcadores se implementan mediante el decorador `@pytest.mark`.

El marco `pytest` proporciona algunos marcadores integrados, y los más populares se describen a continuación:

- `omitir`: el ejecutor de la prueba omitirá un caso de prueba incondicionalmente cuando se utilice este marcador.
- `skipif`: este marcador se usa para omitir una prueba basada en una expresión condicional que se pasa como argumento a este marcador.
- `xfail`: este marcador se usa para ignorar una falla esperada en un caso de prueba. Se utiliza con una determinada condición.
- `parametrizar`: este marcador se utiliza para realizar múltiples llamadas al caso de prueba con diferentes valores como argumentos.

Para demostrar el uso de los tres primeros marcadores, reescribimos nuestro `test_add3.py` módulo agregando marcadores con las funciones de caso de prueba. El módulo de caso de prueba revisado (`test_add4.py`) se muestra aquí:

```
@pytest.mark.skip

def test_add1():
    """ caso de prueba para validar dos números positivos"""
    afirmar agregar (10, 5) == 15


@pytest.mark.skipif(sys.version_info > (3,6),\ Reason =" omitido para
lanzamiento > que Python 3.6")

def test_add2():
    """ caso de prueba para validar dos números positivos"""
    afirmar agregar (10, -5) == 5, "debería ser 5"


@pytest.mark.xfail(sys.plataforma == "win32", \
razón = "ignorar excepción para Windows")
def test_add3():
    """ caso de prueba para validar dos números positivos"""
    afirmar agregar (-10, 5) == -5
    generar excepción ()
```

Usamos el marcador de salto incondicionalmente para el primer caso de prueba. Esto ignorará el caso de prueba. Para el segundo caso de prueba, usamos el marcador skipif con una condición de una versión de Python superior a la 3.6. Para el último caso de prueba, lanzamos deliberadamente una excepción y usamos el marcador xfail para ignorar este tipo de excepción si la plataforma del sistema es Windows. Este tipo de marcador es útil para ignorar errores en casos de prueba si se esperan para una determinada condición, como el sistema operativo en este caso.

El resultado de la consola de la ejecución de los casos de prueba se muestra aquí:

```
test_myadd4.py::test_add1 SKIPPED (salto incondicional) [33%]
Omitido: salto incondicional

test_myadd4.py::test_add2 OMITIDO (omitido para la versión > que Pytho...) [66%]

Omitido: omitido para el lanzamiento> que Python 3.6

test_myadd4.py::test_add3 XFAIL (ignorar excepción para mac)
[100%]

@ pytest.mark.xfail(sys.plataforma == "win32",
```

```
razón="ignorar excepción para mac")
===== 2 saltados, 1 fallado en 0.06s =====
```

A continuación, discutiremos el uso del marcador parametrizar con la biblioteca pytest .

Construcción de casos de prueba con parametrización

En todos los ejemplos de código anteriores, creamos funciones o métodos de casos de prueba sin pasárselos ningún parámetro. Pero para muchos escenarios de prueba, necesitamos ejecutar el mismo caso de prueba variando los datos de entrada. En un enfoque clásico, ejecutamos múltiples casos de prueba que son diferentes solo en términos de los datos de entrada que usamos para ellos. Nuestro ejemplo anterior de test_myadd3.py muestra cómo implementar casos de prueba utilizando este enfoque clásico. Un enfoque recomendado para este tipo de prueba es utilizar **pruebas basadas en datos (DDT)**.

DDT es una forma de prueba en la que los datos de prueba se proporcionan a través de una tabla, un diccionario o una hoja de cálculo para un solo caso de prueba. Este tipo de prueba también se denomina **prueba basada en tablas** o prueba **parametrizada**. Los datos proporcionados a través de una tabla o un diccionario se utilizan para ejecutar las pruebas utilizando una implementación común del código fuente de la prueba. DDT es beneficioso en escenarios en los que tenemos que probar la funcionalidad mediante una permutación de parámetros de entrada. En lugar de escribir casos de prueba para cada permutación de parámetros de entrada, podemos proporcionar las permutaciones en formato de tabla o de diccionario y usarlo como entrada para nuestro único caso de prueba. Los marcos como pytest ejecutarán nuestro caso de prueba tantas veces como el número de permutaciones esté en la tabla o el diccionario. Un ejemplo del mundo real de DDT es validar el comportamiento de una función de inicio de sesión de una aplicación utilizando una variedad de usuarios con credenciales válidas y no válidas.

En el marco de pytest , DDT se puede implementar mediante la parametrización con el marcador pytest . Al usar el marcador de parametrización , podemos definir qué argumento de entrada necesitamos pasar y también el conjunto de datos de prueba que necesitamos usar. El pytest framework ejecutará automáticamente la función de caso de prueba varias veces según la cantidad de entradas en los datos de prueba proporcionados con el marcador de parametrización .

Para ilustrar cómo usar el marcador de parametrización para DDT, revisaremos nuestro myadd4. módulo py para los casos de prueba de la función add . En el código revisado, solo tendremos una función de caso de prueba, pero se usarán diferentes datos de prueba para los parámetros de entrada, como se ilustra en el siguiente fragmento:

```
# test_myadd5.py conjunto de pruebas usando el marcador de parametrización
sistema de importación
importar pytest
desde mypytest.src.myadd3 importar añadir
```

```
@pytest.mark.parametrize("x,y,ans",
[(10,5,15),(10,-5,5),
(-10,5,-5),(-10,-5,-15)],
ids=["pos-pos","pos-neg",
"neg-pos", "neg-neg"])
def test_add(x, y, respuesta):
""" caso de prueba para validar dos números positivos"""
afirmar agregar (x, y) == ans
```

Para el marcador de parametrización , utilizamos tres parámetros, que se describen a continuación:

- **Argumentos del caso de prueba:** proporcionamos una lista de argumentos para pasar a nuestra prueba función en el mismo orden definido con la definición de función de caso de prueba. Además, los datos de prueba que debemos proporcionar en el siguiente argumento seguirán el mismo orden.
- **Datos:** Los datos de prueba que se pasarán serán una lista de diferentes conjuntos de argumentos de entrada. El número de entradas en los datos de prueba determinará cuántas veces se ejecutará el caso de prueba.
- **ids:** este es un parámetro opcional que principalmente adjunta una etiqueta amigable a diferentes conjuntos de datos de prueba que proporcionamos en el argumento anterior. Estas etiquetas de **identificación (ID)** se utilizarán en el informe de salida para identificar diferentes ejecuciones del mismo caso de prueba.

La salida de la consola para la ejecución de este caso de prueba se muestra a continuación:

```
test_myadd5.py::test_add[pos-pos] APROBADO [25%]
test_myadd5.py::test_add[pos-neg] APROBADO [50%]
test_myadd5.py::test_add[neg-pos] APROBADO [75%]
test_myadd5.py::test_add[neg-neg] APROBADO [100%]
===== 4 pasaron en 0.04s =====
```

Esta salida de la consola nos muestra cuántas veces se ejecuta el caso de prueba y con qué datos de prueba. Los casos de prueba creados con los marcadores pytest son concisos y fáciles de implementar. Esto ahorra mucho tiempo y nos permite escribir más casos de prueba (solo variando los datos) en poco tiempo.

A continuación, discutiremos otra característica importante de la biblioteca pytest : accesorios.

176 Pruebas y Automatización con Python

Creación de casos de prueba con accesorios pytest

En el marco de pytest , los accesorios de prueba se implementan mediante decoradores de Python (@pytest.fixture).

La implementación de accesorios de prueba en el marco pytest es muy poderosa en comparación con otros marcos por las siguientes razones clave:

- Los accesorios en el marco pytest proporcionan una alta escalabilidad. Podemos definir una configuración genérica o accesorios (métodos) que se pueden reutilizar en funciones, clases, módulos y paquetes.
- La implementación de dispositivos del marco pytest es de naturaleza modular. Podemos usar uno o más accesorios con un caso de prueba. Un dispositivo también puede usar uno o muchos otros dispositivos, al igual que usamos funciones para llamar a otras funciones.
- Cada caso de prueba en un conjunto de pruebas tendrá la flexibilidad de usar el mismo conjunto o uno diferente de accesorios.
- Podemos crear accesorios en el marco pytest con un alcance establecido para ellos. Él alcance predeterminado es función, lo que significa que el dispositivo se ejecutará antes que cada función (caso de prueba). Otras opciones de alcance son módulo, clase, paquete o sesión. Estos se definen brevemente a continuación:
 - a) Función: El dispositivo se destruye después de ejecutar un caso de prueba.
 - b) Módulo: El dispositivo se destruye después de ejecutar el último caso de prueba en un módulo.
 - c) Clase: El aparato se destruye después de ejecutar el último caso de prueba en una clase.
 - d) Paquete: El dispositivo se destruye después de ejecutar el último caso de prueba en un paquete.
 - e) Sesión: El dispositivo se destruye después de ejecutar el último caso de prueba en una sesión de prueba.

El marco pytest tiene algunos accesorios incorporados útiles que se pueden usar de forma inmediata, como capfd para capturar la salida a los descriptores de archivos, capsys para capturar la salida a stdout y stderr, solicitud para proporcionar información sobre la función de prueba solicitante, y testdir para proporcionar un directorio de prueba temporal para las ejecuciones de prueba.

Los accesorios en el marco de pytest también se pueden usar para restablecer o desmantelar al final de un caso de prueba. Discutiremos esto más adelante en esta sección.

En el siguiente ejemplo de código, crearemos casos de prueba para nuestra clase MyCalc utilizando accesorios personalizados. El código de muestra para MyCalc ya se comparte en Ejecución de varios conjuntos de pruebas sección. La implementación de un dispositivo de prueba y casos de prueba se muestra aquí:

```
# test_mycalc1.py funciones de cálculo de prueba usando dispositivo de prueba
```

```
sistema de importación
```

```
importar pytest
```

```
desde mypytest.src.myadd3 importar agregar
desde mypytest.src.mycalc importar MyCalc

@pytest.fixture(alcance="módulo")
def mi_calc():
    devolver MiCalc()

@pytest.fixture
def prueba_datos():
    devolver {'x':10, 'y':5}

def test_add(my_calc, test_data):
    """ caso de prueba para sumar dos números"""
    afirmar my_calc.add(test_data.get('x'), \
        test_data.get('y')) == 15

def test_subtract(my_calc, test_data):
    """ caso de prueba para restar dos números"""
    afirmar my_calc.subtract(test_data.get('x'), \
        prueba_datos.get('y'))== 5
```

En este ejemplo de conjunto de pruebas, estos son los puntos clave de discusión:

- Creamos dos accesorios: my_calc y test_data. El accesorio my_calc está configurado con un alcance establecido en módulo porque queremos que se ejecute solo una vez para proporcionar una instancia de la clase MyCalc . El accesorio test_data está utilizando el alcance predeterminado (función), lo que significa que se ejecutará antes que cada método.
- Para los casos de prueba (test_add y test_subtract), usamos los accesorios como argumentos de entrada. El nombre del argumento tiene que coincidir con el nombre de la función del dispositivo. El marco pytest busca automáticamente un accesorio con el nombre utilizado como argumento para un caso de prueba.

El ejemplo de código que discutimos es el uso de un accesorio como función de configuración. Una pregunta que podemos hacer es: ¿Cómo podemos lograr la funcionalidad de desmontaje con los accesorios de prueba de pytest? Hay dos enfoques disponibles para implementar la funcionalidad de desmontaje, y estos se analizan a continuación.

178 Pruebas y Automatización con Python

Usando el rendimiento en lugar de una declaración de devolución

Con este enfoque, escribimos un código principalmente con fines de configuración, usamos una declaración de rendimiento en lugar de return, y luego escriba el código para fines de desmontaje después de la declaración de rendimiento .

Si tenemos un conjunto de pruebas o un módulo con muchos accesorios utilizados, el corredor de pruebas de pytest ejecutará cada accesorio (según el orden de ejecución evaluado) hasta que se encuentre la declaración de rendimiento . Tan pronto como se completa la ejecución del caso de prueba, el corredor de prueba de pytest desencadena la ejecución de todos los accesorios que se producen y ejecuta el código que se escribe después de la declaración de rendimiento . El uso de un enfoque basado en el rendimiento es limpio en el sentido de que el código es fácil de seguir y mantener. Por lo tanto, es un enfoque recomendado.

Agregar un método de finalizador usando el dispositivo de solicitud

Con este enfoque, debemos considerar tres pasos para escribir un método de desmontaje, que se describen a continuación:

- Tenemos que usar un objeto de solicitud en nuestros accesorios. El objeto de solicitud puede ser proporcionado utilizando el accesorio incorporado con el mismo nombre.
- Definiremos un método de desmontaje , por separado o como parte de la implementación del accesorio.
- Proporcionaremos el método de desmontaje como un método invocable para el objeto de solicitud mediante el método addfinalizer .

Para ilustrar ambos enfoques con ejemplos de código, modificaremos nuestra implementación anterior de los accesorios. En el código revisado, implementaremos my_calc accesorio usando un enfoque de rendimiento y el accesorio data_set usando un addfinalizer Acerarse. Aquí está el ejemplo de código revisado:

```
# test_mycalc2.py funciones de cálculo de prueba usando dispositivo de prueba
<importar declaraciones>
@pytest.fixture(alcance="módulo")
def mi_calc():
    mi_calc = MiCalc()
    producir mi_calc
    del mi_calc

    @pytest.fixture
    def data_set(solicitud):
        dictado = {'x':10, 'y':5}
        def delete_dict(obj):
            pass
        return dictado, delete_dict
    return data_set

    def calculate(x, y):
        return x + y
    calculate.__name__ = 'calculate'
    return calculate
```

del objeto

solicitud.addfinalizer(lambda: delete_dict(dict))

dictado de regreso

<resto de los casos de prueba>

Tenga en cuenta que no hay una necesidad real de la funcionalidad de desmontaje para estos dispositivos de ejemplo, pero los agregamos con fines ilustrativos.

Consejo

El uso de nose y doctest para la automatización de pruebas es similar al uso de los marcos unittest y pytest .

En la siguiente sección, discutiremos un enfoque TDD para el desarrollo de software.

Ejecutando TDD

TDD es una práctica bien conocida en ingeniería de software. Este es un enfoque de desarrollo de software en el que los casos de prueba se escriben primero antes de escribir cualquier código para una característica requerida en una aplicación. Aquí están las tres reglas simples de TDD:

- No escriba ningún código funcional a menos que escriba una prueba unitaria que esté fallando.
- No escriba ningún código adicional en la misma prueba más de lo que necesita para hacer la Prueba fallida.
- No escriba ningún código funcional más de lo que se necesita para pasar una prueba fallida.

Estas reglas TDD también nos impulsan a seguir un famoso enfoque de tres fases de desarrollo de software llamado **Red, Green, Refactor**. Las fases se repiten continuamente para TDD.

Estas tres fases se muestran en la Figura 5.4 y se describen a continuación.

Rojo

En esta fase, el primer paso es escribir una prueba sin tener ningún código para probar. La prueba obviamente fallará en este caso. No intentaremos escribir un caso de prueba completo, solo escribiremos suficiente código para fallar la prueba.

Verde

En esta fase, el primer paso es escribir el código hasta que pase una prueba ya escrita.

Nuevamente, solo escribiremos suficiente código para pasar la prueba. Ejecutaremos todas las pruebas para asegurarnos de que también pasen las pruebas escritas anteriormente.

refactorizar

En esta fase, debemos considerar mejorar la calidad del código, lo que significa hacer que el código sea fácil de leer y usar optimización; por ejemplo, se deben eliminar los valores codificados. También se recomienda ejecutar las pruebas después de cada ciclo de refactorización. El resultado de la fase de refactorización es un código limpio. Podemos repetir el ciclo agregando más escenarios de prueba y agregando código para que la nueva prueba pase, y este ciclo debe repetirse hasta que se desarrolle una función.

Es importante comprender que TDD no es un enfoque de prueba ni de diseño. Es un enfoque para desarrollar software de acuerdo con especificaciones definidas por escrito.

casos de prueba primero.

El siguiente diagrama muestra las tres fases de TDD:

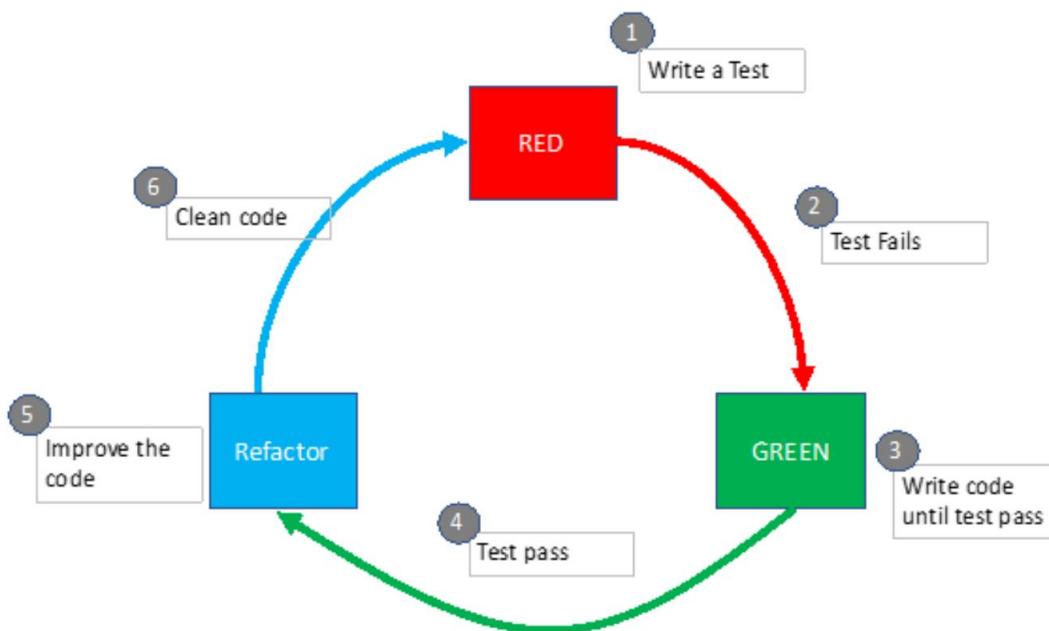


Figura 5.4 – TDD, también conocido como Red, Green, Refactor

En la siguiente sección, presentaremos el papel de la automatización de pruebas en el proceso de IC.

Introducción a la IC automatizada

CI es un proceso que combina los beneficios de las pruebas automatizadas y los sistemas de control de versiones para lograr un entorno de integración completamente automatizado. Con un enfoque de desarrollo de CI, integramos nuestro código en un repositorio compartido con frecuencia. Cada vez que agregamos nuestro código a un repositorio, se espera que se activen los siguientes dos procesos:

- Un proceso de compilación automatizado comienza a validar que el código recién agregado no rompa nada desde el punto de vista de compilación o sintaxis.
- Se inicia una ejecución de prueba automatizada para verificar que tanto los existentes como los nuevos la funcionalidad es según los casos de prueba definidos.

Los diferentes pasos y fases del proceso de CI se representan en el siguiente diagrama.

Aunque mostramos la fase de compilación en este diagrama de flujo, no es una fase obligatoria para los proyectos basados en Python, ya que podemos ejecutar pruebas de integración sin código compilado:

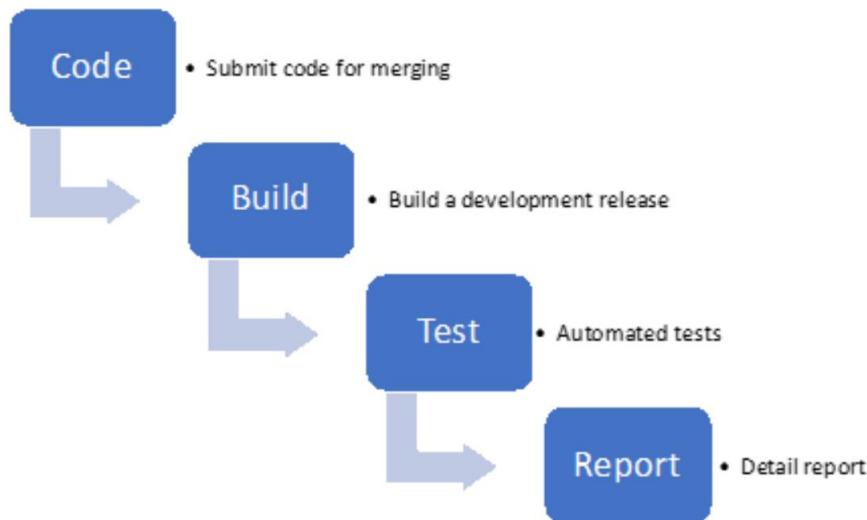


Figura 5.5 – Fases de la prueba de CI

Para construir un sistema CI, necesitamos tener un control de versiones distribuido estable y una herramienta que pueda usarse para implementar un flujo de trabajo para probar toda la aplicación a través de una serie de conjuntos de pruebas. Hay varias herramientas de software comerciales y de código abierto disponibles que proporcionan funcionalidades de CI y **entrega continua (CD)**. Estas herramientas están diseñadas para una fácil integración con un sistema de control de código fuente y con un marco de automatización de pruebas. Algunas herramientas populares disponibles para CI son Jenkins, Bamboo, Buildbot, GitLab CI, CircleCI y Buddy. Los detalles de estas herramientas aparecen en la sección Lecturas adicionales, para aquellos de ustedes que estén interesados en obtener más información.

182 Pruebas y Automatización con Python

Los beneficios obvios de este CI automatizado son detectar errores rápidamente y corregirlos de manera más conveniente desde el principio. Es importante comprender que CI no se trata de corregir errores, pero definitivamente ayuda a identificar errores fácilmente y solucionarlos rápidamente.

Resumen

En este capítulo, presentamos diferentes niveles de prueba para aplicaciones de software. También evaluamos dos marcos de prueba (unittest y pytest) que están disponibles para la automatización de pruebas de Python. Aprendimos cómo construir casos de prueba de nivel básico y avanzado utilizando estos dos marcos. Más adelante en el capítulo, presentamos el enfoque TDD y sus claros beneficios para el desarrollo de software. Finalmente, tocamos el tema de CI, que es un paso clave en la entrega de software utilizando modelos **ágiles** y **de operaciones de desarrollo (devops)**.

Este capítulo es útil para cualquier persona que quiera comenzar a escribir pruebas unitarias para su aplicación Python. Los ejemplos de código proporcionados proporcionan un buen punto de partida para escribir casos de prueba utilizando cualquier marco de prueba.

En el próximo capítulo, exploraremos diferentes trucos y consejos para desarrollar aplicaciones en Python.

Preguntas

1. ¿La prueba unitaria es una forma de prueba de caja blanca o de caja negra?
2. ¿Cuándo debemos usar objetos simulados?
3. ¿Qué métodos se utilizan para implementar accesorios de prueba con el marco unittest ?
4. ¿En qué se diferencia TDD de CI?
5. ¿Cuándo debemos usar DDT?

Otras lecturas

- Aprendizaje de pruebas de Python, por Daniel Arbuckle
- Desarrollo basado en pruebas con Python, por Harry JW Percival
- Programación Experta en Python, por Michaï Jaworski y Tarek Ziadé
- Los detalles del marco unittest están disponibles con la documentación de Python en <https://docs.python.org/3/library/unittest.html>.

respuestas

1. Pruebas de caja blanca
2. Los objetos simulados ayudan a simular el comportamiento de las dependencias internas o externas. Por usando objetos simulados, podemos enfocarnos en escribir pruebas para validar el comportamiento funcional.
3. configuración, desmontaje, configuración de clase, desmontaje de clase
4. TDD es un enfoque para desarrollar software escribiendo primero los casos de prueba. CI es un proceso en el que se ejecutan todas las pruebas cada vez que construimos una nueva versión. No existe una relación directa entre TDD y CI.
5. DDT se usa cuando tenemos que hacer pruebas funcionales con varias permutaciones de parámetros de entrada. Por ejemplo, si debemos probar un punto final de API con una combinación diferente de argumentos, podemos aprovechar DDT.

6

Consejos avanzados y Trucos en Python

En este capítulo, presentaremos algunos consejos y trucos avanzados que se pueden usar como poderosas técnicas de programación al escribir código en Python. Estos incluyen el uso avanzado de funciones de Python, como funciones anidadas, funciones lambda y decoradores de edificios con funciones. Además, cubriremos las transformaciones de datos con las funciones de filtro, mapeador y reductor. A esto le seguirán algunos trucos que se pueden usar con estructuras de datos, como el uso de diccionarios anidados y la comprensión con diferentes tipos de colecciones. Finalmente, investigaremos la funcionalidad avanzada de la biblioteca pandas para objetos DataFrame. Estos consejos y trucos avanzados no solo demostrarán el poder de Python para lograr funciones avanzadas con menos código, sino que también lo ayudarán a codificar de manera más rápida y eficiente.

En este capítulo, cubriremos los siguientes temas:

- Aprender trucos avanzados para usar funciones
- Comprensión de conceptos avanzados con estructuras de datos
- Introducción de trucos avanzados con pandas DataFrame

186 consejos y trucos avanzados en Python

Al final de este capítulo, habrá adquirido una comprensión de cómo usar las funciones de Python para características avanzadas como transformaciones de datos y decoradores de edificios. Además, aprenderá a usar estructuras de datos, incluido pandas DataFrame, para aplicaciones basadas en análisis.

Requerimientos técnicos

Los requisitos técnicos para este capítulo son los siguientes:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- Debe registrar una cuenta con TestPyPI y crear un token de API en su cuenta.

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter06>.

Comenzaremos nuestra discusión con los conceptos avanzados para usar funciones en Python.

Aprender trucos avanzados para usar funciones

El uso de funciones en Python y otros lenguajes de programación es clave para la reutilización y la modularización. Sin embargo, con los nuevos avances en los lenguajes de programación modernos, el papel de las funciones se ha extendido más allá de la reutilización, lo que incluye escribir código simple, corto y conciso sin usar bucles complejos ni declaraciones condicionales.

Comenzaremos con el uso de las funciones counter, zip e itertools , de las que hablaremos a continuación.

Presentamos las funciones counter, itertools y zip para tareas iterativas

Para cualquier tarea de procesamiento de datos, los desarrolladores utilizan mucho los iteradores. Hemos cubierto iteradores, en detalle, en el Capítulo 4, Bibliotecas de Python para programación avanzada. En esta sección, aprenderemos sobre el siguiente nivel de funciones de utilidad para ayudarlo a trabajar convenientemente con iteradores e iterables. Estos incluyen el módulo contador , la función zip y el módulo itertools . Discutiremos cada uno de estos en las siguientes subsecciones.

Mostrador

El **contador** es un tipo de contenedor que realiza un seguimiento del recuento de cada elemento que está presente en un contenedor. El conteo de elementos en un contenedor es útil para encontrar la frecuencia de los datos, lo cual es un requisito previo para muchas aplicaciones de análisis de datos. Para ilustrar el concepto y el uso de la clase Counter , presentaremos un ejemplo de código simple, como sigue:

```
#contador.py
de colecciones importación Contador

#aplicando contador en un objeto de cadena
imprimir(Contador("personas"))

#aplicando contador en un objeto de lista
mi_contador = Contador([1,2,1,2,3,4,1,3])
imprimir (mi_contador.más_común (1))
imprimir(lista(mi_contador.elementos()))

#aplicando contador en un objeto dict
imprimir (Contador ({'A': 2, 'B': 2, 'C': 2, 'C': 3}))
```

En el ejemplo de código anterior, creamos varias instancias de contador usando una cadena objeto, un objeto de lista y un objeto de diccionario. La clase Counter tiene métodos como most_common y elementos. Usamos el método most_common con un valor de 1, que nos da el elemento que más aparece en el contenedor my-counter . Además, usamos el método de elementos para devolver la lista original de la instancia de Counter . La salida de la consola de este programa debe ser la siguiente:

```
Contador ({'p': 2, 'e': 2, 'o': 1, 'l': 1})
[(1, 3)]
[1, 1, 1, 2, 2, 3, 3, 4]
Contador ({'C': 4, 'A': 2, 'B': 2})
```

Es importante tener en cuenta que en el caso del objeto de diccionario, usamos deliberadamente una clave repetida, pero en la instancia de Counter , solo obtenemos un par clave-valor, que es el último en el diccionario. Además, los elementos de la instancia de Counter se ordenan en función de los valores de cada elemento. Tenga en cuenta que la clase Counter convierte el objeto de diccionario en un objeto de tabla hash.

188 consejos y trucos avanzados en Python

Código Postal

La función zip se usa para crear un iterador agregado basado en dos o más iteradores individuales. La función zip es útil cuando se requiere iterar en múltiples iteraciones en paralelo. Por ejemplo, podemos usar la función zip cuando implementamos algoritmos matemáticos que involucran interpolación o reconocimiento de patrones. Esto también es útil en el procesamiento de señales digitales donde combinamos múltiples señales (fuentes de datos) en una sola señal.

Aquí hay un ejemplo de código simple que usa una función zip :

#enérgico

```
lista_núm = [1, 2, 3, 4, 5]
lett_list = ['alfa', 'bravo', 'charlie']

zipped_iter = zip(num_list,lett_list)
imprimir (siguiente (zipped_iter))
imprimir (siguiente (zipped_iter))
imprimir (lista (comprimido_iter))
```

En el ejemplo de código anterior, combinamos las dos listas con fines de iteración mediante la función zip . Tenga en cuenta que una lista es más grande que la otra en términos del número de elementos. La salida de la consola de este programa debe ser la siguiente:

```
(1, 'alfa')
(2, 'bravo')
[(3, 'charlie'), (4, 'delta')]
```

Como era de esperar, obtenemos las dos primeras tuplas usando la siguiente función, que es una combinación de los elementos correspondientes de cada lista. Al final, usamos el constructor de listas para iterar sobre el resto de las tuplas del iterador zip . Esto nos da una lista de las tuplas restantes en formato de lista.

itertools

Python ofrece un módulo, llamado `itertools`, que proporciona funciones útiles para trabajar con iteradores. Cuando se trabaja con un gran conjunto de datos, el uso de iteradores es imprescindible, y ahí es donde las funciones de utilidad proporcionadas por el módulo `itertools` resultan muy útiles. Hay muchas funciones disponibles con el módulo `itertools`. Presentaremos brevemente algunas funciones clave aquí:

- contar: esta función se utiliza para crear un iterador para contar números. Podemos proporcionar un número inicial (predeterminado = 0) y, opcionalmente, establecer un tamaño del paso de conteo para el incremento. El siguiente ejemplo de código devolverá un iterador que proporciona números de conteo, como 10, 12 y 14:

```
#itertools_count.py
importar itertools
iter = itertools.count(10, 2)
imprimir (siguiente (iter))
imprimir (siguiente (iter))
```

- ciclo: esta función le permite recorrer un iterador sin fin.

El siguiente fragmento de código ilustra cómo puede usar esta función para una lista de letras del alfabeto:

```
letras = {'A','B','C'}
para letra en itertools.cycle(letras):
    imprimir (letra)
```

- Repetir: esta función nos proporciona un iterador que devuelve un objeto una y otra vez a menos que haya un argumento de tiempo establecido con él. El siguiente fragmento de código repetirá el objeto de cadena de Python cinco veces:

```
para x en itertools.repeat('Python', times=5):
    imprimir (x)
```

- acumular: esta función devolverá un iterador que nos proporciona un suma acumulada u otros resultados acumulados basados en una función de agregación que se pasó a esta función de acumulación como argumento. Es más fácil entender el uso de esta función con un ejemplo de código, como sigue:

```
#itertools_accumulate.py
importar itertools, operator
lista1 = [1, 3, 5]
```

190 consejos y trucos avanzados en Python

```
res = itertools.acumular(lista1)
imprimir("predeterminado:")
para x en res:
    imprimir (x)
res = itertools.accumulate(lista1, operador.mul)
imprimir("Multiplicar: ")
para x en res:
    imprimir (x)
```

En este ejemplo de código, primero, usamos la función de acumulación sin proporcionar una función de agregador para ningún resultado acumulado. Por defecto, la acumulación La función agregará dos números (1 y 3) de la lista original. Este proceso se repite para todos los números y los resultados se almacenan dentro de un iterable (en nuestro caso, esto es res). En la segunda parte de este ejemplo de código, proporcionamos la función mul (multiplicación) del módulo del operador y, esta vez, los resultados acumulados se basan en la multiplicación de dos números.

- cadena: esta función combina dos o más iterables y devuelve un iterable combinado. Eche un vistazo al siguiente código de ejemplo que muestra dos iterables (listas) junto con la función de cadena :

```
lista1 = ['A', 'B', 'C']
lista2 = ['W','X','Y','Z']

iter_encadenado = itertools.chain(lista1, lista2)
para x en chained_iter:
    imprimir (x)
```

Tenga en cuenta que esta función combinará los iterables de forma serial. Esto significa que se podrá acceder primero a los elementos de la lista1, seguidos de los elementos de la lista2 .

- comprimir: esta función se puede usar para filtrar elementos de un iterable en función de otro iterable.

En el fragmento de código de ejemplo, hemos seleccionado letras del alfabeto de una lista basada en un selector iterable:

```
letras = ['A','B','C']
selector = [Verdadero, 0, 1]
para x en itertools.compress(letras, selector):
    imprimir (x)
```

Para el selector iterable, podemos usar True/False o 1/0. La salida de este programa serán las letras A y C.

- **groupby:** esta función identifica las claves para cada elemento en un objeto iterable y agrupa los elementos en función de las claves identificadas. Esta función requiere otra función (conocida como `key_func`) que identifica una clave en cada elemento de un objeto iterable. El siguiente código de ejemplo explica el uso de esta función junto con cómo implementar una función `key_func`:

```
#itertools_groupby.py
importar itertools

milista = [("A", 100), ("A", 200), ("B", 30), \
           ("B", 10)]
def get_key(grupo):
    grupo de retorno[0]

para clave, grp en itertools.groupby(mylist, get_key):
    imprimir(tecla + "-->", lista(grp))
```

- **tee:** esta es otra función útil que se puede usar para duplicar iteradores de un solo iterador. Aquí hay un código de ejemplo que duplica dos iteradores de una sola lista iterable:

```
letras = ['A', 'B', 'C']
iter1, iter2 = itertools.tee(letras)

para x en iter1:
    imprimir (x)

para x en iter2:
    imprimir (x)
```

A continuación, analizaremos otra categoría de funciones que se usa ampliamente para la transformación de datos.

Uso de filtros, mapeadores y reductores para transformaciones de datos

mapear, filtrar y reducir son tres funciones disponibles en Python que se utilizan para simplificar y escribir código conciso. Estas tres funciones se aplican a los iterables de una sola vez sin usar declaraciones iterativas. Las funciones de mapa y filtro están disponibles como funciones integradas, mientras que la función de reducción requiere que importe las herramientas de función módulo. Estas funciones son ampliamente utilizadas por los científicos de datos para el procesamiento de datos. La función de mapa y la función de filtro se usan para transformar o filtrar datos, mientras que la función de reducción se usa en el análisis de datos para obtener resultados significativos de un gran conjunto de datos.

En las siguientes subsecciones, evaluaremos cada función con su aplicación y ejemplos de código.

mapa

La función de mapa en Python se define usando la siguiente sintaxis:

```
map(func, iter, ...)
```

El argumento func es el nombre de la función que se aplicará a cada elemento del objeto iter . Los tres puntos indican que es posible pasar múltiples objetos iterables.

Sin embargo, es importante comprender que la cantidad de argumentos de la función (func) debe coincidir con la cantidad de objetos iterables. La salida de la función de mapa es un objeto de mapa , que es un objeto generador. El valor devuelto se puede convertir en una lista pasando el objeto de mapa al constructor de listas .

Nota IMPORTANTE

En Python 2, la función de mapa devuelve una lista. Este comportamiento se ha cambiado en Python 3.

Antes de discutir el uso de una función de mapa , primero implementaremos una función de transformación simple que convierte una lista de números en sus valores cuadrados. El ejemplo de código se proporciona a continuación:

```
#map1.py para obtener el cuadrado de cada elemento en una lista
```

```
milista = [1, 2, 3, 4, 5]
nueva_lista = []
```

```
para el artículo en mi lista:
```

```
cuadrado = elemento*elemento  
new_list.append(cuadrado)  
  
imprimir (nueva_lista)
```

Aquí, el ejemplo de código usa una estructura de bucle for para iterar a través de una lista, calcula el cuadrado de cada entrada en la lista y luego lo agrega a una nueva lista. Este estilo de escribir código es común, pero definitivamente no es una forma Pythonic de escribir código. La salida de la consola de este programa es la siguiente:

```
[1, 4, 9, 16, 25]
```

Con el uso de la función de mapa , este código se puede simplificar y acortar, de la siguiente manera:

```
# map2.py para obtener el cuadrado de cada elemento en una lista  
  
def cuadrado(num):  
    devolver número * número  
  
milista = [1, 2, 3, 4, 5]  
nueva_lista = lista(mapa(cuadrado, milista))  
imprimir (nueva_lista)
```

Al usar la función de mapa , proporcionamos el nombre de la función (en este ejemplo, es cuadrado) y la referencia de la lista (en este ejemplo, es milista). El objeto de mapa que devuelve la función de mapa se convierte en un objeto de lista utilizando la lista constructor. La salida de la consola de este ejemplo de código es la misma que la del ejemplo de código anterior.

En el siguiente ejemplo de código, proporcionaremos dos listas como entrada para la función de mapa :

```
# map3.py para obtener el producto de cada artículo en dos listas  
  
def producto(num1, num2):  
    devuelve num1 * num2  
  
milista1 = [1, 2, 3, 4, 5]  
milista2 = [6, 7, 8, 9]  
lista_nueva = lista(mapa(producto, milista1, milista2))  
imprimir (nueva_lista)
```

194 Consejos y trucos avanzados en Python

Esta vez, el objetivo de la función de mapa que se ha implementado es utilizar el producto función. La función de producto toma cada elemento de dos listas y multiplica el elemento correspondiente en cada lista antes de devolverlo a la función de mapa .

La salida de la consola de este ejemplo de código es la siguiente:

```
[6, 14, 24, 36]
```

Un análisis de la salida de esta consola nos dice que la función de mapa solo usa los primeros cuatro elementos de cada lista . La función map se detiene automáticamente cuando se queda sin elementos en cualquiera de los iterables (en nuestro caso, estas son las dos listas). Esto significa que incluso si proporcionamos iterables de diferentes tamaños, la función map no generará ninguna excepción, pero funcionará para la cantidad de elementos que es posible mapear a través de iterables usando la función proporcionada. En nuestro ejemplo de código, tenemos un número menor de elementos en la lista mylist2 , que es cuatro. Es por eso que solo tenemos cuatro elementos en la lista de salida (en nuestro caso, esta es new_list). A continuación, discutiremos la función de filtro con algunos ejemplos de código.

filtrar

La función de filtro también opera en iterables pero solo en un objeto iterable. Como sugiere su nombre, proporciona una funcionalidad de filtrado en el objeto iterable. Los criterios de filtrado se proporcionan a través de la definición de la función. La sintaxis de una función de filtro es la siguiente:

```
filtro (func, iter)
```

La función func proporciona los criterios de filtrado y tiene que devolver True o False. Dado que solo se permite un iterable junto con la función de filtro , solo se permite un argumento para la función func . El siguiente ejemplo de código usa una función de filtro para seleccionar los elementos cuyos valores son números pares. Para implementar los criterios de selección, se implementa la función is_even para evaluar si un número que se le proporciona es un número par o no. El código de ejemplo es el siguiente:

```
# filter1.py para obtener números pares de una lista
```

```
def es_par(num):
    retorno (núm % 2 == 0)

milista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
nueva_lista = lista(filtro(es_par, milista))
imprimir (nueva_lista)
```

La salida de la consola del ejemplo de código anterior es la siguiente:

```
[2, 4, 6, 8]
```

A continuación, discutiremos la función reduce .

reducir

La función de reducción se utiliza para aplicar una función de procesamiento acumulativo en cada elemento de una secuencia, que se le pasa como argumento. Esta función de procesamiento acumulativo no tiene fines de transformación o filtrado. Como sugiere su nombre, la función de procesamiento acumulativo se utiliza para obtener un único resultado al final basado en todos los elementos de una secuencia. La sintaxis del uso de la función reduce es la siguiente:

```
reducir (función, iter[, inicial])
```

La función func es una función que se utiliza para aplicar un procesamiento acumulativo en cada elemento del iterable. Además, initial es un valor opcional que se puede pasar a la función func para que se use como valor inicial para el procesamiento acumulativo. Es importante comprender que siempre habrá dos argumentos para la función func para el caso de la función reduce : el primer argumento será el valor inicial (si se proporciona) o el primer elemento de la secuencia, y el segundo argumento será el siguiente elemento de la secuencia.

En el siguiente ejemplo de código, usaremos una lista simple de los primeros cinco números.

Implementaremos un método personalizado para sumar los dos números y luego usaremos el método de reducción para sumar todos los elementos de la lista. El ejemplo de código se muestra a continuación:

```
# reduce1.py para obtener la suma de números de una lista
de functools importar reducir

def seq_sum(num1, num2):
    devuelve num1+num2

milista = [1, 2, 3, 4, 5]
resultado = reducir(seq_sum, milista)
imprimir (resultado)
```

196 Consejos y trucos avanzados en Python

La salida de este programa es 15, que es una suma numérica de todos los elementos de la lista (en nuestro ejemplo, esto se llama `mylist`). Si proporcionamos el valor inicial a la reducción función, el resultado se agregará según el valor inicial. Por ejemplo, la salida del mismo programa con la siguiente declaración será 25:

```
resultado = reducir(seq_sum, milista, 10)
```

Como se mencionó anteriormente, el resultado o el valor de retorno de la función reduce es un valor único, que es como la función `func`. En este ejemplo, será un número entero.

En esta sección, analizamos las funciones de mapa, filtro y reducción que están disponibles en Python. Los científicos de datos utilizan ampliamente estas funciones para la transformación y el refinamiento de datos. Un problema de usar funciones como mapa y filtro es que devuelven un objeto del tipo mapa o filtro , y tenemos que convertir los resultados explícitamente en un tipo de datos de lista para su posterior procesamiento. Las comprensiones y los generadores no tienen tales limitaciones, pero brindan una funcionalidad similar y son relativamente más fáciles de usar. Es por eso que están obteniendo más tracción que las funciones de mapa, filtro y reducción . Discutiremos la comprensión y los generadores en la sección Comprensión de conceptos avanzados con estructuras de datos. A continuación, investigaremos el uso de funciones lambda.

Aprendiendo a construir funciones lambda

Las funciones lambda son funciones anónimas que se basan en una expresión de una sola línea. Así como la palabra clave `def` se usa para definir funciones regulares, la palabra clave `lambda` se usa para definir funciones anónimas. Las funciones de Lambda están restringidas a una sola línea. Esto significa que no pueden usar varias declaraciones y no pueden usar una declaración de devolución. El valor de retorno se devuelve automáticamente después de la evaluación de la expresión de una sola línea.

Las funciones lambda se pueden usar en cualquier lugar donde se use una función normal. El uso más fácil y conveniente de las funciones lambda es con el mapa, reducir y filtrar funciones Las funciones Lambda son útiles cuando desea que el código sea más conciso.

Para ilustrar una función lambda, reutilizaremos los ejemplos de código de mapa y filtro que discutimos anteriormente. En estos ejemplos de código, reemplazaremos `func` con una función lambda, como se destaca en el siguiente fragmento de código:

```
# lambda1.py para obtener el cuadrado de cada elemento en una lista  
  
milista = [1, 2, 3, 4, 5]  
lista_nueva = lista(mapa(lambda x: x*x, milista))  
imprimir (nueva_lista)
```

```
# lambda2.py para obtener números pares de una lista

milista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
nueva_lista = lista(filtro(lambda x: x % 2 == 0, milista))
imprimir (nueva_lista)

# lambda3.py para obtener el producto del artículo correspondiente en el \
dos listas

milista1 = [1, 2, 3, 4, 5]
milista2 = [6, 7, 8, 9]
lista_nueva = lista(mapa(lambda x,y: x*y, milista1, milista2))
imprimir (nueva_lista)
```

Aunque el código se ha vuelto más conciso, debemos tener cuidado al usar funciones lambda. Estas funciones no son reutilizables y no son fáciles de mantener. Necesitamos repensar esto antes de introducir una función lambda en nuestro programa. Cualquier cambio o funcionalidad adicional no será fácil de agregar. Una regla general es usar solo funciones lambda para expresiones simples cuando escribir una función separada sería una sobrecarga.

Incrustar una función dentro de otra función

Cuando agregamos una función dentro de una función existente, se llama **función interna** o **función anidada**. La ventaja de tener funciones internas es que tienen acceso directo a las variables que están definidas o disponibles en el ámbito de una función externa. Crear una función interna es lo mismo que definir una función regular con la palabra clave def y con la sangría adecuada. Las funciones internas no pueden ser ejecutadas o llamadas por el programa externo. Sin embargo, si la función externa devuelve una referencia de la función interna, la persona que llama puede usarla para ejecutar la función interna. Echaremos un vistazo a ejemplos de referencias de funciones internas devueltas para muchos casos de uso en las siguientes subsecciones.

Las funciones internas tienen muchas ventajas y aplicaciones. A continuación describiremos algunos de ellos.

Encapsulación

Un caso de uso común de una función interna es poder ocultar su funcionalidad del mundo exterior. La función interna solo está disponible dentro del alcance de la función externa y no es visible para el alcance global. El siguiente ejemplo de código muestra una función externa que oculta una función interna:

```
#interior1.py
```

```
def exterior_hola():
    print ("Hola desde la función externa")
def interior_hola():
    print("Hola desde la función interna")
interior_hola()

exterior_hola()
```

Desde el exterior de la función externa, solo podemos llamar a la función externa. La función interna solo se puede llamar desde el cuerpo de la función externa.

Funciones auxiliares

En algunos casos, podemos encontrarnos en una situación en la que el código dentro de un código de función es reutilizable. Podemos convertir dicho código reutilizable en una función separada; de lo contrario, si el código es reutilizable solo dentro del alcance de una función, entonces se trata de construir una función interna. Este tipo de función interna también se denomina función auxiliar. El siguiente fragmento de código ilustra este concepto:

```
def exterior_fn(x, y):
    def obtener_prefijo(s):
        devolver s[:2]
    x2 = obtener_prefijo(x)
    y2 = obtener_prefijo(y)
    #procesar x2 y y2 más
```

En el código de muestra anterior, definimos una función interna, llamada `get_prefix` (una función auxiliar), dentro de una función externa para filtrar las dos primeras letras de un valor de argumento. Dado que tenemos que repetir este proceso de filtrado para todos los argumentos, agregamos una función auxiliar para la reutilización dentro del alcance de esta función, ya que es específica de esta función.

El cierre y las funciones de fábrica.

Este es un tipo de caso de uso en el que brillan las funciones internas. Un **cierre** es una función interna junto con su entorno envolvente. Un cierre es una función creada dinámicamente que puede ser devuelta por otra función. La verdadera magia de un cierre es que la función devuelta tiene acceso completo a las variables y espacios de nombres donde se creó. Esto es cierto incluso cuando la función envolvente (en este contexto, es la función externa) ha terminado de ejecutarse.

El concepto de cierre se puede ilustrar con un ejemplo de código. El siguiente ejemplo de código muestra un caso de uso en el que hemos implementado una fábrica de cierre para crear una función para calcular la potencia del valor base, y el cierre retiene el valor base:

```
# interior2.py
def power_calc_factory(base):
    def power_calc(exponente):
        return base**exponente
    return power_calc

power_calc_2 = power_gen_factory(2)
power_calc_3 = power_gen_factory(3)
imprimir (poder_calc_2 (2))
imprimir (poder_calc_2 (3))
imprimir (poder_calc_3 (2))
imprimir (poder_calc_3 (4))
```

En el ejemplo de código anterior, la función externa (es decir, `power_calc_factory`) actúa como una función de fábrica de cierre porque crea un nuevo cierre cada vez que se llama y luego devuelve el cierre a la persona que llama. Además, `power_calc` es una función interna que toma una variable (es decir, la base) del espacio de nombres de cierre y luego toma la segunda variable (es decir, el exponente), que se le pasa como argumento. Tenga en cuenta que la declaración más importante es `return power_calc`. Esta declaración devuelve la función interna como un objeto con su recinto.

200 consejos y trucos avanzados en Python

Cuando llamamos a la función `power_calc_factory` por primera vez junto con el argumento `base` , se crea un cierre con su espacio de nombres, incluido el argumento que se le pasó, y el cierre se devuelve a la persona que llama. Cuando volvemos a llamar a la misma función, obtenemos un nuevo cierre con el objeto de función interno. En este ejemplo de código, creamos 2 cierres: uno con un valor base de 2 y el otro con un valor base de 3. Cuando llamamos a la función interna pasando diferentes valores para la variable `exponente` , el código dentro de la función interna (en este caso, la función `power_calc`) también tendrá acceso al valor base que ya se pasó a la función externa.

Estos ejemplos de código ilustraron el uso de funciones externas e internas para crear funciones dinámicamente. Tradicionalmente, las funciones internas se utilizan para ocultar o encapsular la funcionalidad dentro de una función. Pero cuando se usan junto con las funciones externas actuando como una fábrica para crear funciones dinámicas, se convierte en la aplicación más poderosa de las funciones internas. Las funciones internas también se utilizan para implementar decoradores.

Hablaremos de esto con más detalle en la siguiente sección.

Modificar el comportamiento de la función usando decoradores

El concepto de decoradores en Python se basa en el patrón de diseño **Decorator** , que es un tipo de patrón de diseño estructural. Este patrón le permite agregar un nuevo comportamiento a los objetos sin cambiar nada en la implementación del objeto. Este nuevo comportamiento se agrega dentro de los objetos contenedores especiales.

En Python, los **decoradores** son funciones especiales de orden superior que permiten a los desarrolladores agregar nuevas funciones a una función existente (o un método) sin agregar ni cambiar nada dentro de la función. Normalmente, estos decoradores se agregan antes de la definición de una función. Los decoradores se utilizan para implementar muchas características de una aplicación, pero son particularmente populares en la validación de datos, registro, almacenamiento en caché, depuración, cifrado y gestión de transacciones.

Para crear un decorador, tenemos que definir una entidad invocable (es decir, una función, un método o una clase) que acepte una función como argumento. La entidad invocable devolverá otro objeto de función con un comportamiento definido por el decorador. La función que está decorada (la llamaremos función decorada en el resto de esta sección) se pasa como argumento a la función que está implementando un decorador (que se llamará función decoradora).

para el resto de esta sección). La función decoradora ejecuta la función que se le pasó además del comportamiento adicional que se agregó como parte de la función decoradora.

Un ejemplo simple de un decorador se muestra en el siguiente ejemplo de código en el que definimos un decorador para agregar una marca de tiempo antes y después de la ejecución de una función:

```
# decorador1.py
desde fechahora fechahora de importación

def add_timestamps(myfunc):
    def _add_timestamps():
        imprimir(fechahora.ahora())
        mifunc()
        imprimir(fechahora.ahora())
    devolver _add_timestamps

@add_timestamps
def hola_mundo():
    imprimir("hola mundo")

Hola Mundo()
```

En este ejemplo de código, definimos una función decoradora `add_timestamps` que toma cualquier función como argumento. En la función interna (`_add_timestamps`), tomamos la hora actual antes y después de la ejecución de la función, que luego se pasa como argumento. La función decoradora devuelve el objeto de función interna con un cierre. Los decoradores no están haciendo nada más que usar las funciones internas de manera inteligente, como discutimos en la sección anterior. El uso del símbolo `@` para decorar una función equivale a las siguientes líneas de códigos:

```
hola = add_timestamps(hola_mundo)
Hola()
```

En este caso, estamos llamando a la función decoradora explícitamente al pasar el nombre de la función como parámetro. En otras palabras, la función decorada es igual a la función interna, que se define dentro de la función decoradora. Así es exactamente como Python interpreta y llama a la función de decorador cuando ve un decorador con el símbolo `@` antes de la definición de una función.

202 Consejos y trucos avanzados en Python

Sin embargo, surge un problema cuando tenemos que obtener detalles adicionales sobre la invocación de funciones, lo cual es importante para la depuración. Cuando usamos la función de ayuda integrada con la función `hello_world`, solo recibimos ayuda para la función interna. Lo mismo sucede si usamos el `docstring`, que también funcionará para la función interna pero no para la función decorada. Además, serializar el código será un desafío para las funciones decoradas. Hay una solución simple disponible en Python para todos estos problemas; esa solución es usar el decorador de envolturas de la biblioteca `functools`. Revisaremos nuestro ejemplo de código anterior para incluir el decorador de envolturas. El ejemplo de código completo es el siguiente:

```
# decorador2.py
desde fechahora fechahora de importación
desde functools importar envolturas

def add_timestamps(myfunc):
    @wraps(mifunc)
    def _add_timestamps():
        imprimir(fechahora.ahora())
        mifunc()
        imprimir(fechahora.ahora())
        devolver _add_timestamps

    @add_timestamps
    def hola_mundo():
        imprimir("hola mundo")

    Hola Mundo()
    ayuda(hola_mundo)
    imprimir (hola_mundo)
```

El uso de los decoradores `wraps` proporcionará detalles adicionales sobre las ejecuciones de las funciones anidadas, y podemos verlas en la salida de la consola si ejecutamos el código de ejemplo que se ha proporcionado.

Hasta ahora, hemos visto un ejemplo simple de un decorador para explicar este concepto. En el resto de esta sección, aprenderemos cómo pasar argumentos con una función a un decorador, cómo devolver valor de un decorador y cómo encadenar varios decoradores. Para comenzar, aprenderemos cómo pasar atributos y devolver un valor con decoradores.

Usar una función decorada con un valor devuelto y un argumento

Cuando nuestra función decorada toma argumentos, entonces decorar dicha función requiere algunos trucos adicionales. Un truco es usar `*args` y `**kwargs` en la función contenedora interna. Esto hará que la función interna acepte cualquier número arbitrario de argumentos posicionales y de palabras clave. Aquí hay un ejemplo simple de una función decorada con argumentos junto con el valor de retorno:

```
# decorador3.py
desde functools importar envolturas

potencia definida(función):
    @wraps(función)
    def cálculo_interno(*args, **kwargs):
        print("Función de poder de decoración")
        n = func(*args, **kwargs)
        regreso m
        volver calculo_interno

@energía
def power_base2(n):
    devolver 2**n

imprimir (power_base2 (3))
```

En el ejemplo anterior, la función interna de `inner_calc` toma los parámetros genéricos de `*args` y `**kwargs`. Para devolver un valor de una función interna (en nuestro ejemplo de código, `inner_calc`), podemos mantener el valor devuelto por la función (en nuestro ejemplo de código, esto es `func` o `power_base2(n)`) que se ejecuta dentro de nuestra función interna y devuelve el valor de retorno final de la función interna de `inner_calc`.

Construyendo un decorador con argumentos propios

En los ejemplos anteriores, usamos lo que llamamos **decoradores estándar**. Un decorador estándar es una función que obtiene el nombre de la función decorada como argumento y devuelve una función interna que funciona como una función decoradora. Sin embargo, es un poco diferente cuando tenemos un decorador con sus propios argumentos. Dichos decoradores se construyen sobre decoradores estándar. En pocas palabras, un decorador con argumentos es otra función que en realidad devuelve un decorador estándar (no la función interna dentro de un decorador). Este concepto de una función de decorador estándar envuelta dentro de otra función de decorador se puede entender mejor con una versión revisada del ejemplo `decorator3.py`. En la versión revisada, calculamos la potencia de un valor base que se pasa como argumento al decorador. Puede ver un ejemplo de código completo usando funciones de decorador anidadas de la siguiente manera:

```
# decorador4.py
desde functools importar envolturas

def power_calc(base):
    def decorador_interno(función):
        @wraps(función)
        def cálculo_interno(*args, **kwargs):
            exponente = func(*args, **kwargs)
            retorno base**exponente
            volver calculo_interno
            volver interior_decorador

    @power_calc(base=3)
    def potencia_n(n):
        regreso m

    imprimir (poder_n (2))
    imprimir (poder_n (4))
```

El funcionamiento de este ejemplo de código es el siguiente:

- La función de decorador `power_calc` toma una base de argumento y devuelve la función `inner_decorator`, que es una implementación de decorador estándar.
- La función `inner_decorator` toma una función como argumento y devuelve la función `inner_calc` para realizar el cálculo real.

- La función inner_calc llama a la función decorada para obtener el exponente atributo (en este caso) y luego usa el atributo base , que se pasa a la función de decorador externo como argumento. Como era de esperar, el cierre alrededor de la función interna hace que el valor del atributo base esté disponible para inner_calc función.

A continuación, discutiremos cómo usar más de un decorador con una función o un método.

Usar múltiples decoradores

Hemos aprendido en numerosas ocasiones que existe la posibilidad de utilizar más de un decorador con una función. Esto es posible encadenando a los decoradores. Los decoradores encadenados pueden ser iguales o diferentes. Esto se puede lograr colocando los decoradores uno tras otro antes de la definición de la función. Cuando se usa más de un decorador con una función, la función decorada solo se ejecuta una vez. Para ilustrar su implementación y uso práctico, hemos seleccionado un ejemplo en el que registramos un mensaje en un sistema de destino utilizando una marca de tiempo. La marca de tiempo se agrega a través de un decorador independiente y el sistema de destino también se selecciona en función de otro decorador. El siguiente ejemplo de código muestra las definiciones de tres decoradores, es decir, add_time_stamp, file y consola:

```
# decorador5.py (parte 1)
desde fechahora fechahora de importación
desde functools importar envolturas

def add_timestamp(función):
    @wraps(función)
    def función_interna(*argumentos, **kwargos):
        res = "{};{}\n".format(datetime.now(),func(*args,
**kwargos))
        volver res
    volver función_interna

archivo de definición (función):
@wraps(función)
def función_interna(*argumentos, **kwargos):
    res = func(*argumentos, **kwargos)
    con open("log.txt", 'a') como archivo:
        archivo.escribir(res)
```

206 Consejos y trucos avanzados en Python

```
volver res  
volver función_interna  
  
def consola(función):  
    @wraps(función)  
    def función_interna(*argumentos, **kwargs):  
        res = func(*argumentos, **kwargs)  
        imprimir (res)  
        volver res  
  
    volver función_interna
```

En el ejemplo de código anterior, implementamos tres funciones de decorador.

Son los siguientes:

- archivo: este decorador usa un archivo de texto predefinido y agrega el mensaje provisto por la función decorada al archivo.
- consola: este decorador emite el mensaje proporcionado por la función decorada a la consola
- add_timestamp: este decorador agrega una marca de tiempo antes del mensaje proporcionado por la función decorada. La ejecución de esta función de decorador debe ocurrir antes que los decoradores de archivos o consolas, lo que significa que este decorador debe colocarse en último lugar en la cadena de decoradores.

En el siguiente fragmento de código, podemos usar estos decoradores para diferentes funciones dentro de nuestro programa principal:

```
#decorador5.py (parte 2)  
@expediente  
@añadir_marca_de_tiempo  
def log(mensaje):  
    devolver mensaje  
  
@expediente  
@consola  
@añadir_marca_de_tiempo  
def log1(mensaje):  
    devolver mensaje
```

```
@consola
@añadir_marca_de_tiempo
def log2(mensaje):
    devolver mensaje

log("Este es un mensaje de prueba solo para archivo")
log1("Este es un mensaje de prueba tanto para el archivo como para la consola")
log2("Este mensaje es solo para consola")
```

En el ejemplo de código anterior, usamos las tres funciones de decorador definidas anteriormente en diferentes combinaciones para exhibir los diferentes comportamientos de la misma función de registro. En la primera combinación, enviamos el mensaje al archivo solo después de agregar la marca de tiempo. En la segunda combinación, enviamos el mensaje tanto al archivo como a la consola. En la combinación final, enviamos el mensaje solo a la consola. Esto demuestra la flexibilidad que brindan los decoradores sin necesidad de cambiar las funciones.

Vale la pena mencionar que los decoradores son muy útiles para simplificar el código y agregar comportamiento de manera concisa, pero tienen el costo de gastos generales adicionales durante la ejecución. El uso de decoradores debe limitarse a aquellos escenarios donde el beneficio es suficiente para compensar los costos generales.

Esto concluye nuestra discusión sobre conceptos y trucos de funciones avanzadas. En la siguiente sección, cambiaremos de tema a algunos conceptos avanzados relacionados con las estructuras de datos.

Comprender conceptos avanzados con datos estructuras

Python ofrece soporte completo para estructuras de datos, incluidas herramientas clave para almacenar datos y acceder a datos para su procesamiento y recuperación. En el Capítulo 4, Bibliotecas de Python para programación avanzada, discutimos los objetos de estructura de datos que están disponibles en Python. En esta sección, cubriremos una serie de conceptos avanzados, como un diccionario dentro de un diccionario y cómo usar la comprensión con una estructura de datos. Comenzaremos incrustando un diccionario dentro de un diccionario.

Incrustar un diccionario dentro de un diccionario

Un diccionario en un diccionario o un diccionario anidado es el proceso de colocar un diccionario dentro de otro diccionario. Un diccionario anidado es útil en muchos ejemplos del mundo real, especialmente cuando procesa y transforma datos de un formato a otro.

208 Consejos y trucos avanzados en Python

La figura 6.1 muestra un diccionario anidado. El diccionario raíz tiene dos diccionarios contra la tecla 1 y la tecla 3. El diccionario contra la tecla 1 tiene más diccionarios dentro. El diccionario contra la clave 3 es un diccionario regular con pares clave-valor como sus entradas:

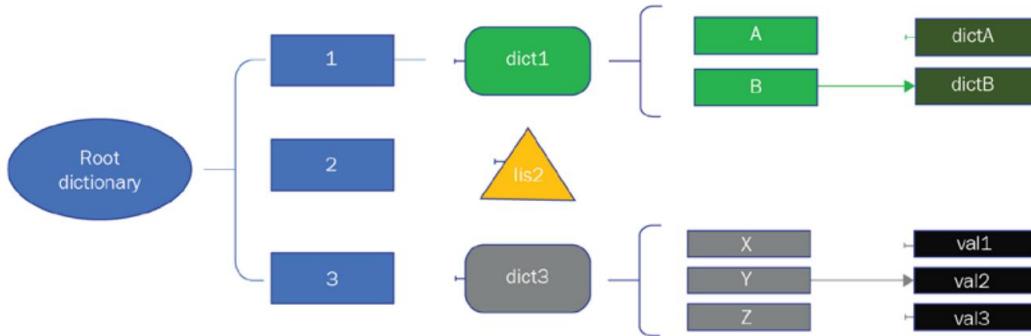


Figura 6.1: Un ejemplo de un diccionario dentro de un diccionario

El diccionario raíz que se muestra en la Figura 6.1 se puede escribir de la siguiente manera:

```

root_dict = {'1': {'A': {dictA}, 'B': {dictB}},
             '2': [lista2],
             '3': {'X': valor1, 'Y': valor2, 'Z': valor3}
         }
  
```

Aquí, creamos un diccionario raíz con una combinación de objetos de diccionario y objetos de lista dentro de él.

Creación o definición de un diccionario anidado Se puede definir o

crear un diccionario anidado colocando diccionarios separados por comas entre corchetes. Para demostrar cómo crear un diccionario anidado, crearemos un diccionario para estudiantes. Cada entrada de estudiante tendrá otro diccionario con el nombre y la edad como elementos, que se asignan a su número de estudiante:

```

# diccionario1.py

dict1 = {100:{'nombre':'Juan', 'edad':24},
          101:{'nombre':'Miguel', 'edad':22},
          102:{'nombre':'Jim', 'edad':21} }

imprimir (dict1)
imprimir(dict1.get(100))
  
```

A continuación, aprenderemos cómo crear un diccionario dinámicamente y cómo agregar o actualizar elementos de diccionario anidados.

Agregar a un diccionario anidado

Para crear un diccionario en un diccionario de forma dinámica o para agregar elementos a un diccionario anidado existente, podemos usar varios enfoques. En el siguiente ejemplo de código, utilizaremos tres enfoques diferentes para crear un diccionario anidado. Son los mismos que definimos en el módulo `dictionary1.py`:

- En el primer caso, construiremos un diccionario interno (es decir, `estudiante101`) a través de la asignación directa de elementos de pares clave-valor y luego asignándolos a una clave en el diccionario raíz. Este es el enfoque preferido siempre que sea posible porque el código es más fácil de leer y administrar.
- En el segundo caso, creamos un diccionario interno vacío (es decir, `estudiante102`) y asignamos los valores a las claves a través de instrucciones de asignación. Este también es un enfoque preferido cuando los valores están disponibles para nosotros a través de otras estructuras de datos.
- En el tercer caso, iniciamos directamente un directorio vacío para la tercera clave del diccionario raíz. Después del proceso de inicialización, asignamos los valores usando doble indexación (es decir, dos claves): la primera clave es para el diccionario raíz y la segunda clave es para el diccionario interno. Este enfoque hace que el código sea conciso, pero no es el enfoque preferido si la legibilidad del código es importante por motivos de mantenimiento.

El ejemplo de código completo para estos tres casos diferentes es el siguiente:

```
# diccionario2.py
#definiendo el diccionario interno 1
estudiante100 = {'nombre': 'Juan', 'edad': 24}

#definiendo el diccionario interno 2
estudiante101 = {}
estudiante101['nombre'] = 'Miguel'
estudiante101['edad'] = '22'

#asignación de diccionarios internos 1 y 2 a un diccionario raíz
dict1 = {}
dict1[100] = estudiante100
dict1[101] = estudiante101
```

210 consejos y trucos avanzados en Python

```
#creando diccionario interno directamente dentro de una raíz \
```

```
diccionario
```

```
dict1[102] = {}
```

```
dict1[102]['nombre'] = 'Jim'
```

```
dict1[102]['edad'] = '21'
```

```
imprimir (dict1)
```

```
imprimir(dict1.get(102))
```

A continuación, discutiremos cómo acceder a diferentes elementos de un diccionario anidado.

Acceder a elementos de un diccionario anidado

Como comentamos anteriormente, para agregar valores y diccionarios dentro de un diccionario, podemos usar la doble indexación. Alternativamente, podemos usar el método `get` del objeto del diccionario. El mismo enfoque es aplicable para acceder a diferentes elementos de un diccionario interno. El siguiente es un código de ejemplo que ilustra cómo acceder a diferentes elementos de los diccionarios internos utilizando el método `get` y los índices dobles:

```
# diccionario3.py
```

```
dict1 = {100:{'nombre':'Juan', 'edad':24},
```

```
101:{'nombre':'Miguel', 'edad':22},
```

```
102:{'nombre':'Jim', 'edad':21} }
```

```
imprimir(dict1.get(100))
```

```
imprimir(dict1.get(100).get('nombre'))
```

```
imprimir (dict1 [101])
```

```
imprimir(dict1[101]['edad'])
```

A continuación, examinaremos cómo eliminar un diccionario interno o un elemento de par clave-valor de un diccionario interno.

Eliminar de un diccionario anidado

Para eliminar un diccionario o un elemento de un diccionario, podemos usar el genérico del función, o podemos usar el método pop del objeto del diccionario . En el siguiente código de ejemplo, presentaremos tanto la función del como el método pop para demostrar su uso:

```
# diccionario4.py
```

```
dict1 = {100:{'nombre':'Juan', 'edad':24},  
        101:{'nombre':'Miguel', 'edad':22},  
        102:{'nombre':'Jim', 'edad':21} }  
  
del (dict1[101]['edad'])  
imprimir (dict1)  
dict1[102].pop('edad')  
imprimir (dict1)
```

En la siguiente sección, discutiremos cómo la comprensión ayuda a procesar datos de diferentes tipos de estructuras de datos.

Usando la comprensión

La comprensión es una forma rápida de crear nuevas secuencias, como listas, conjuntos y diccionarios a partir de secuencias existentes. Python admite cuatro tipos diferentes de comprensión, de la siguiente manera:

- Lista de comprensión
- Comprensión de diccionario
- Comprensión de conjuntos
- Comprensión del generador

Discutiremos una breve descripción, con ejemplos de código, para cada uno de estos tipos de comprensión en las siguientes subsecciones.

Lista de comprensión

La **comprensión** de listas implica la creación de una lista dinámica utilizando un bucle y una declaración condicional si es necesario.

Algunos ejemplos de cómo usar la comprensión de listas nos ayudarán a comprender mejor el concepto. En el primer ejemplo (es decir, list1.py), crearemos una nueva lista a partir de una lista original agregando 1 a cada elemento de la lista original. Aquí está el fragmento de código:

```
#lista1.py

lista1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
lista2 = [x+1 para x en lista1]
imprimir (lista2)
```

En este caso, la nueva lista se creará utilizando la expresión `x+1`, donde `x` es un elemento de la lista original. Esto es equivalente al siguiente código tradicional:

```
lista2 = []
para x en lista1:
    lista2.append(x+1)
```

Usando la comprensión de listas, podemos lograr estas tres líneas de código con solo una línea de código.

En el segundo ejemplo (es decir, list2.py), crearemos una nueva lista a partir de la lista original de números del 1 al 10, pero solo incluiremos números pares. Podemos hacer esto simplemente agregando una condición al ejemplo de código anterior, de la siguiente manera:

```
#lista2.py

lista1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lista2 = [x para x en lista1 si x % 2 == 0]
imprimir (lista2)
```

Como puede ver, la condición se agrega al final de la expresión de comprensión.

A continuación, discutiremos cómo construir diccionarios usando comprensión.

Comprensión de diccionario

También se pueden crear **diccionarios utilizando la comprensión de diccionario**. La comprensión de diccionario, que es similar a la comprensión de lista, es un enfoque para crear un diccionario a partir de otro diccionario de tal manera que los elementos del diccionario de origen se seleccionen o transformen condicionalmente. El siguiente fragmento de código muestra un ejemplo de cómo crear un diccionario a partir de elementos de diccionario existentes que son menores o iguales a 200 y dividir cada valor seleccionado por 2. Tenga en cuenta que los valores también se vuelven a convertir en números enteros como parte de la expresión de comprensión:

```
#dictcomp1.py
```

```
dict1 = {'a': 100, 'b': 200, 'c': 300}  
dict2 = {x:int(y/2) para (x, y) en dict1.items() si y <=200}  
imprimir (dict2)
```

Este código de comprensión de diccionario es equivalente al siguiente fragmento de código si se realiza mediante programación tradicional:

```
dictado2 = {}  
para x,y en dict1.items():  
    si y <= 200:  
        dict2[x] = int(y/2)
```

Tenga en cuenta que la comprensión reduce significativamente el código. A continuación, hablaremos de la comprensión de conjuntos.

Establecer comprensión

Los conjuntos también se pueden crear utilizando **la comprensión de conjuntos**, al igual que la comprensión de listas. La sintaxis del código para crear conjuntos mediante la comprensión de conjuntos es similar a la comprensión de listas. La excepción es que usaremos corchetes en lugar de corchetes. En el siguiente fragmento de código, puede ver un ejemplo de cómo crear un conjunto a partir de una lista mediante la comprensión de conjuntos:

```
#setcomp1.py
```

```
lista1 = [1, 2, 6, 4, 5, 6, 7, 8, 9, 10, 8]  
conjunto1 = {x para x en lista1 si x % 2 == 0}  
imprimir (conjunto1)
```

214 Consejos y trucos avanzados en Python

Este código de comprensión establecido es equivalente al siguiente fragmento de código con la programación tradicional:

```
Conjunto1 = conjunto()  
para x en lista1:  
    si x % 2 == 0:  
        conjunto1.añadir(x)
```

Como era de esperar, las entradas duplicadas se descartarán en el conjunto.

Esto concluye nuestra discusión sobre los tipos de comprensión que están disponibles en Python para diferentes estructuras de datos. A continuación, discutiremos las opciones de filtrado que están disponibles con las estructuras de datos.

Presentamos trucos avanzados con pandas

Marco de datos

pandas es una biblioteca Python de código abierto que proporciona herramientas para la manipulación de datos de alto rendimiento para hacer que el análisis de datos sea rápido y fácil. Los usos típicos de la biblioteca pandas son remodelar, clasificar, segmentar, agregar y fusionar datos.

La biblioteca pandas se basa en la biblioteca **NumPy**, que es otra biblioteca de Python que se usa para trabajar con matrices. La biblioteca NumPy es significativamente más rápida que las listas tradicionales de Python porque los datos se almacenan en una ubicación continua en la memoria, lo que no ocurre con las listas tradicionales.

La biblioteca pandas trata con tres estructuras de datos clave, de la siguiente manera:

- Serie: este es un objeto similar a una matriz unidimensional que contiene una matriz de datos y una matriz de etiquetas de datos. La matriz de etiquetas de datos se denomina índice. el índice se puede especificar automáticamente usando números enteros de 0 a n-1 si un usuario no lo especifica explícitamente.
- DataFrame: esta es una representación de datos tabulares como una hoja de cálculo que contiene una lista de columnas. El objeto DataFrame ayuda a almacenar y manipular datos tabulares en filas y columnas. Curiosamente, el objeto DataFrame tiene un índice para columnas y filas.
- Panel: Es un contenedor tridimensional de datos.

El DataFrame es la estructura de datos clave que se utiliza en el análisis de datos. En el resto de esta sección, utilizaremos ampliamente el objeto DataFrame en nuestros ejemplos de código.

Antes de discutir cualquier truco avanzado con respecto a estos objetos Pandas DataFrame, haremos una revisión rápida de las operaciones fundamentales disponibles para los objetos DataFrame.

Aprendizaje de operaciones de DataFrame

Comenzaremos creando objetos DataFrame. Hay varias formas de crear un DataFrame, como desde un diccionario, un archivo CSV, una hoja de Excel o desde una matriz NumPy. Una de las formas más sencillas es utilizar los datos de un diccionario como entrada. El siguiente fragmento de código muestra cómo puede crear un objeto DataFrame basado en los datos meteorológicos semanales almacenados en un diccionario:

```
# pandas1.py
importar pandas como pd

datos_semanales = {'dia':['lunes','martes', 'miercoles', \
'Jueves, Viernes, Sábado, Domingo'],
'temperatura':[40, 33, 42, 31, 41, 40, 30],
'condición':['Soleado', 'Nublado', 'Soleado', 'Lluvia' \
, 'Soleado', 'Nublado', 'Lluvia']
}

df = pd.DataFrame(weekly_data)
imprimir (df)
```

La salida de la consola mostrará el contenido del DataFrame de la siguiente manera:

	day	temp	condition
0	Monday	40	Sunny
1	Tuesday	33	Cloudy
2	Wednesday	42	Sunny
3	Thursday	31	Rain
4	Friday	41	Sunny
5	Saturday	40	Cloudy
6	Sunday	30	Rain

Figura 6.2 – El contenido del DataFrame

216 Consejos y trucos avanzados en Python

La biblioteca de pandas es muy rica en términos de métodos y atributos. Sin embargo, está más allá del alcance de esta sección cubrirlos todos. En su lugar, presentaremos un breve resumen de los atributos y métodos comúnmente utilizados de los objetos DataFrame a continuación para actualizar nuestro conocimiento antes de usarlos en los próximos ejemplos de código:

- índice: este atributo proporciona una lista de índices (o etiquetas) del objeto DataFrame.
- columnas: este atributo proporciona una lista de columnas en el objeto DataFrame.
- tamaño: Devuelve el tamaño del objeto DataFrame en términos del número de filas multiplicado por el número de columnas.
- forma: Esto nos proporciona una tupla que representa la dimensión del DataFrame objeto.
- ejes: este atributo devuelve una lista que representa los ejes del objeto DataFrame.
En pocas palabras, incluye filas y columnas.
- describe: este poderoso método genera datos estadísticos como el conteo, la media, desviación estándar y valores mínimos y máximos.
- head: este método devuelve n (predeterminado = 5) filas de un objeto DataFrame similar a el comando principal en los archivos.
- tail: este método devuelve las últimas n (predeterminado = 5) filas de un objeto DataFrame.
- drop_duplicates: este método descarta filas duplicadas en función de todos los columnas en un DataFrame.
- dropna: este método elimina los valores faltantes (como filas o columnas) de un Marco de datos. Al pasar los argumentos apropiados a este método, podemos eliminar filas o columnas. Además, podemos establecer si las filas o columnas se eliminarán en función de una sola aparición de un valor faltante o solo cuando falten todos los valores de una fila o columna.
- sort_values: este método se puede utilizar para ordenar las filas en función de una o varias columnas.

En las siguientes secciones, revisaremos algunas operaciones fundamentales para los objetos DataFrame.

Configuración de un índice

personalizado Las etiquetas de columna (índice) normalmente se agregan según los datos proporcionados con un diccionario o según cualquier otro flujo de datos de entrada que se haya utilizado. Podemos cambiar el índice del DataFrame usando una de las siguientes opciones:

- Establecer una de las columnas de datos como un índice, como el día en el mencionado anteriormente ejemplo, usando una declaración simple como esta:

```
df_nuevo = df.set_index('día')
```

El DataFrame comenzará a usar la columna del día como columna de índice, y su contenido será el siguiente:

		temp	condition
day			
Monday	40	Sunny	
Tuesday	33	Cloudy	
Wednesday	42	Sunny	
Thursday	31	Rain	
Friday	41	Sunny	
Saturday	40	Cloudy	
Sunday	30	Rain	

Figura 6.3 – El contenido del DataFrame después de usar la columna del día como índice

- Configure el índice manualmente proporcionándolo a través de una lista, como en la siguiente fragmento de código:

```
# pandas2.py
week_data = <igual que el ejemplo anterior>
df = pd.DataFrame(weekly_data)
df.index = ['LUN','MAR','MIE','JUE','VIE','SABADO','DOM']
imprimir (df)
```

218 Consejos y trucos avanzados en Python

Con este fragmento de código, DataFrame comenzará a usar el índice proporcionado por nosotros a través de un objeto de lista. El contenido del DataFrame mostrará este cambio de la siguiente manera:

		day	temp	condition
MON	Monday	40	Sunny	
TUE	Tuesday	33	Cloudy	
WED	Wednesday	42	Sunny	
THU	Thursday	31	Rain	
FRI	Friday	41	Sunny	
SAT	Saturday	40	Cloudy	
SUN	Sunday	30	Rain	

Figura 6.4: el contenido del DataFrame después de configurar entradas personalizadas para una columna de índice

A continuación, discutiremos cómo navegar dentro de un DataFrame utilizando un índice y una columna determinados.

Navegando dentro de un DataFrame

Hay algunas docenas de formas de obtener una fila de datos o una ubicación particular de un objeto DataFrame. Los métodos típicos que se utilizan para navegar dentro de un DataFrame son los métodos loc e iloc . Exploraremos algunas opciones de cómo navegar a través de un objeto DataFrame usando los mismos datos de muestra que usamos en el ejemplo anterior:

```
# pandas3.py
importar pandas como pd
datos_semanales = <igual que en el ejemplo pandas1.py>
df = pd.DataFrame(weekly_data)
df.index = ['LUN', 'MAR', 'MIÉ', 'JUE', 'VIE', 'SÁBADO', 'DOM']
```

A continuación, analizaremos algunas técnicas, con ejemplos de código, sobre cómo seleccionar una fila o una ubicación en este objeto DataFrame:

- Podemos seleccionar una o más filas usando etiquetas de índice con el método loc . La etiqueta de índice se proporciona como un elemento único o como una lista. En el siguiente fragmento de código, hemos ilustrado dos ejemplos de cómo seleccionar una o más filas:

```
imprimir(df.loc['MAR'])
imprimir(df.loc[['MAR','MIÉ']])
```

- Podemos seleccionar un valor de una ubicación en un objeto DataFrame usando la etiqueta de índice de fila y la etiqueta de columna, de la siguiente manera:

```
imprimir(df.loc['VIE','temp'])
```

- También podemos seleccionar una fila usando un valor de índice sin proporcionar ninguna etiqueta:

```
#Proporcionar una fila con índice 2  
imprimir(df.iloc[2])
```

- Podemos seleccionar un valor de una ubicación utilizando el valor de índice de fila y el valor de índice de columna al tratar el objeto DataFrame como una matriz bidimensional. En el siguiente fragmento de código, obtendremos un valor de una ubicación en la que el índice de fila = 2 y el índice de columna = 2:

```
imprimir(df.iloc[2,2])
```

A continuación, analizaremos cómo agregar una fila o una columna a un objeto DataFrame.

Agregar una fila o columna a un DataFrame

La forma más fácil de agregar una fila a un objeto DataFrame es asignar una lista de valores a una ubicación de índice o una etiqueta de índice. Por ejemplo, podemos agregar una nueva fila con la etiqueta TST para el ejemplo anterior (es decir, pandas3.py) usando la siguiente declaración:

```
df.loc['TST'] = ['Día de prueba 1', 50, 'NA']
```

Es importante tener en cuenta que si la etiqueta de la fila ya existe en el objeto DataFrame, la misma línea de código puede actualizar la fila con nuevos valores.

Si no usamos la etiqueta de índice sino el índice predeterminado, podemos usar el número de índice para actualizar una fila existente o agregar una nueva fila usando la siguiente línea de código:

```
df.loc[8] = ['Día de prueba 2', 40, 'NA']
```

Se muestra un ejemplo de código completo como referencia:

```
# pandas4.py  
importar pandas como pd  
datos_semanales = <igual que en el ejemplo pandas1.py>  
df = pd.DataFrame(weekly_data)  
df.index = ['LUN', 'MAR', 'MIÉ', 'JUE', 'VIE', 'SÁBADO', 'DOM']  
  
df.loc['TST1'] = ['Día de prueba 1', 50, 'NA']  
df.loc[7] = ['Día de prueba 2', 40, 'NA']  
imprimir (df)
```

220 consejos y trucos avanzados en Python

Para agregar una nueva columna a un objeto DataFrame, hay varias opciones disponibles en la biblioteca de pandas. Solo ilustraremos tres opciones, de la siguiente manera:

- **Agregando una lista de valores junto a la etiqueta de la columna:** este enfoque agregará una columna después de las columnas existentes. Si usamos una etiqueta de columna existente, este enfoque también se puede usar para actualizar o reemplazar una columna existente.
- **Usando el método de inserción:** Este método tomará una etiqueta y una lista de valores como argumentos. Esto es particularmente útil cuando desea insertar una columna en cualquier ubicación. Tenga en cuenta que este método no le permite insertar una columna si ya existe una columna dentro del objeto DataFrame con la misma etiqueta. Esto significa que este método no se puede utilizar para actualizar una columna existente.
- **Mediante el método de asignación:** este método es útil cuando desea agregar varias columnas de una sola vez. Si usamos una etiqueta de columna existente, este método se puede usar para actualizar o reemplazar una columna existente.

En el siguiente ejemplo de código, usaremos los tres enfoques para insertar una nueva columna en un objeto DataFrame:

```
# pandas5.py
import pandas como pd

datos_semanales = <igual que en el ejemplo pandas1.py>
df = pd.DataFrame(weekly_data)

#Aregar una nueva columna y luego actualizarla
df['Humedad1'] = [60, 70, 65,62,56,25,""]
df['Humedad1'] = [60, 70, 65,62,56,251,""]

#Insertar una columna en el índice de columna de 2 usando el método de inserción

df.insert(2, "Humedad2" [60, 70, 65,62,56,25,""])

#Aregar dos columnas usando el método de asignación
df1 = df.assign(Humedad3 = [60, 70, 65,62,56,25,""],
                Humedad4 = [60, 70, 65,62,56,25,""])
imprimir (df1)
```

A continuación, evaluaremos cómo eliminar filas y columnas de un objeto DataFrame.

Eliminación de un índice, una fila o una columna de un marco de datos La eliminación de un índice

es relativamente sencilla y puede hacerlo mediante el método `reset_index`. Sin embargo, el método `reset_index` agrega índices predeterminados y mantiene la columna de índice personalizada como una columna de datos. Para eliminar completamente la columna de índice personalizado, debemos usar el argumento de eliminación con el método `reset_index`. El siguiente fragmento de código utiliza el método `reset_index`:

```
# pandas6.py
importar pandas como pd

datos_semanales = <igual que en el ejemplo de pandas1.py> df =
pd.DataFrame(datos_semanales)

df.index = ['LUN', 'MAR', 'MIÉ', 'JUE', 'VIE', 'SÁBADO', 'SÁBADO']
imprimir(df)
imprimir(df.reset_index(soltar=True))
```

Para eliminar una fila duplicada de un objeto DataFrame, podemos usar el método `drop_duplicate`. Para eliminar una fila o columna en particular, podemos usar el método de soltar. En el siguiente ejemplo de código, eliminaremos las filas con las etiquetas SAT y SUN y las columnas con la etiqueta de condición:

```
#pandas7.py
importar pandas como pd

datos_semanales = <igual que en el ejemplo de pandas1.py> df =
pd.DataFrame(datos_semanales)

df.index = ['LUN', 'MAR', 'MIÉ', 'JUE', 'VIE', 'SÁBADO', 'DOM']
imprimir (df)

df1= df.drop(index=['DOM','SAB'])
df2= df1.drop(columnas=['condición'])

imprimir (df2)
```

A continuación, examinaremos cómo cambiar el nombre de un índice o una columna.

222 Consejos y trucos avanzados en Python

Cambiar el nombre de índices y columnas en un DataFrame

Para cambiar el nombre de un índice o una etiqueta de columna, usaremos el método de cambio de nombre . Un ejemplo de código de cómo cambiar el nombre de un índice y una columna es el siguiente:

```
#pandas8.py
importar pandas como pd

datos_semanales = <igual que en el ejemplo pandas1.py>
df = pd.DataFrame(weekly_data)
df.index = ['LUN', 'MAR', 'MIÉ', 'JUE', 'VIE', 'SÁBADO', 'DOM']

df1=df.renombrar(índice={'DOM': 'SU', 'SÁBADO': 'SA'})
df2=df1.rename(columnas={'condición':'condición'})

imprimir (df2)
```

Es importante tener en cuenta que la etiqueta actual y la nueva etiqueta para el índice y la columna se proporcionan como diccionario. A continuación, discutiremos algunos trucos avanzados para usar objetos DataFrame.

Aprendiendo trucos avanzados para un objeto DataFrame

En la sección anterior, evaluamos las operaciones fundamentales que se pueden realizar en un objeto DataFrame. En esta sección, investigaremos el siguiente nivel de operaciones en un objeto DataFrame para la evaluación y transformación de datos. Estas operaciones se analizan en las siguientes subsecciones.

Reemplazo de datos Un

requisito común es reemplazar datos numéricos o datos de cadena con otro conjunto de valores. La biblioteca de pandas está llena de opciones para llevar a cabo dichos reemplazos de datos.

El método más popular para estas operaciones es usar el método at . El método at proporciona una manera fácil de acceder o actualizar datos en cualquier celda de un DataFrame. Para las operaciones de reemplazo masivo, también puede usar un método de reemplazo , y podemos usar este método de muchas maneras. Por ejemplo, podemos usar este método para reemplazar un número con otro número o una cadena con otra cadena, o podemos reemplazar cualquier cosa que coincida con una expresión regular. Además, podemos usar este método para reemplazar cualquier entrada proporcionada a través de una lista o un diccionario. En el siguiente ejemplo de código (es decir, `pandastrick1.py`), cubriremos la mayoría de estas opciones de reemplazo. Para este ejemplo de código, usaremos el mismo objeto DataFrame que usamos en los ejemplos de código anteriores. Aquí está el código de ejemplo:

```
# pandastrick1.py
import pandas como pd
datos_semanales = <igual que en el ejemplo pandas1.py>
df = pd.DataFrame(weekly_data)
```

A continuación, exploraremos varias operaciones de reemplazo en este objeto DataFrame, una por una:

- Reemplace cualquier ocurrencia del valor numérico de 40 con 39 en el objeto DataFrame usando la siguiente instrucción:

```
df.reemplazar(40,39, en el lugar=True)
```

- Reemplace cualquier aparición de una cadena Sunny con Sun en el objeto DataFrame usando la siguiente declaración:

```
df.replace("Sunny", "Sun", inplace=True)
```

- Reemplazar cualquier ocurrencia de una cadena basada en una expresión regular (el objetivo es reemplazar Cloudy with Cloud) usando la siguiente declaración:

```
df.replace(to_replace="^Cl.*", value="Cloud", inplace=True, regex=True)
```

```
#o también podemos aplicar en una columna específica.
```

```
df["condición"].replace(to_replace="^Cl.*", value="Cloud", inplace=True, regex=True)
```

Tenga en cuenta que el uso de las etiquetas de argumento to_replace y value es opcional. •

Reemplace cualquier aparición de múltiples cadenas representadas por una lista con otra lista de cadenas usando la siguiente declaración:

```
df.replace(["Lunes", "Martes"], ["Lun", "Mar"], en el lugar=True)
```

En este código, reemplazamos lunes y martes con lunes y martes. • Reemplazar

cualquier aparición de múltiples cadenas en un objeto DataFrame usando el pares clave-valor en un diccionario. Puede hacer esto usando la siguiente declaración:

```
df.replace({"Miércoles": "Miércoles", "Jueves": "Jueves"}, en el
lugar=True)
```

En este caso, las claves del diccionario (es decir, miércoles y jueves) serán reemplazadas por sus valores correspondientes (es decir, miércoles y jueves).

224 Consejos y trucos avanzados en Python

- Reemplace cualquier ocurrencia de una cadena para una determinada columna utilizando varios diccionarios.

Puede hacerlo utilizando el nombre de la columna como clave en el diccionario y una declaración de ejemplo como la siguiente:

```
df.replace({"day":"Friday"}, {"day":"Fri"}, inplace=True)
```

En este escenario, el primer diccionario se usa para indicar el nombre de la columna y el valor que se va a reemplazar. El segundo diccionario se usa para indicar el mismo nombre de columna pero con un valor que reemplazará el valor original. En nuestro caso, reemplazaremos todas las instancias de viernes en la columna de día con el valor de viernes .

- Reemplace cualquier aparición de múltiples cadenas utilizando un diccionario anidado. Tu puedes hacer esto mediante el uso de un ejemplo de código como el siguiente:

```
df.replace({"día":{"Sábado":"Sábado", "Domingo":"Dom"},  
"condición":{"Lluvia":"Lluvia"}}, en el lugar=True)
```

En este escenario, el diccionario externo (con las claves de día y condición en nuestro ejemplo de código) se usa para identificar las columnas para esta operación y el diccionario interno se usa para contener los datos que se reemplazarán junto con el valor de reemplazo. Al usar este enfoque, reemplazamos el sábado y el domingo con Sat y Sun dentro de la columna de día y la cadena Rainy con Rain dentro de la columna de condición .

El código completo con todas estas operaciones de muestra está disponible en el código fuente de este capítulo como pandastick1.py. Tenga en cuenta que podemos activar la operación de reemplazo en el objeto DataFrame o podemos limitarla a una determinada columna o fila.

Nota IMPORTANTE

El argumento `inplace=True` se usa con todas las llamadas al método `replace` . Este argumento se usa para establecer la salida del método de reemplazo dentro del mismo objeto DataFrame. La opción predeterminada es devolver un nuevo objeto DataFrame sin cambiar el objeto original. Este argumento está disponible con muchos métodos de DataFrame por conveniencia.

Aplicar una función a la columna o fila de un objeto DataFrame A veces, queremos limpiar los datos,

ajustar los datos o transformar los datos antes de comenzar el análisis de datos. Hay una manera fácil de aplicar algún tipo de función en un DataFrame utilizando los métodos `apply`, `applymap` o `map` . El método `apply` es aplicable a columnas o filas, mientras que el método `applymap` funciona elemento por elemento para todo el DataFrame. En comparación, el método del mapa funciona elemento por elemento para una sola serie. Ahora, discutiremos un par de ejemplos de código para ilustrar el uso de los métodos de aplicación y asignación .

Es común tener datos importados en un objeto DataFrame que podría necesitar un poco de limpieza. Por ejemplo, podría tener espacios en blanco al final o al principio, caracteres de nueva línea o cualquier carácter no deseado. Estos se pueden eliminar fácilmente de los datos utilizando el método de mapa y la función lambda en una serie de columnas. La función lambda se utiliza en cada elemento de la columna. En nuestro ejemplo de código, primero eliminaremos el espacio en blanco, el punto y la coma finales. Luego, eliminaremos el espacio en blanco inicial, el guión bajo y el guión de la columna de condición .

Después de limpiar los datos dentro de la columna de condición , el siguiente paso es crear una nueva columna temp_F a partir de los valores de la columna temporal y convertirlos de unidades Celsius a unidades Fahrenheit. Tenga en cuenta que usaremos otra función lambda para esta conversión y usaremos el método de aplicación . Cuando obtengamos el resultado del método de aplicación , lo almacenaremos dentro de una nueva etiqueta de columna, temp_F, para crear una nueva columna. Aquí está el ejemplo de código completo:

```
# pandastrick2.py
importar pandas como pd

datos_semanales = {'día':['lunes', 'martes', 'miércoles',
                          'Jueves, Viernes, Sábado, Domingo'],
                     'temp':[40, 33, 42, 31, 41, 40, 30],
                     'condición':['Soleado','_Nublado ',
                                  'Soleado', 'Lluvia', '--Soleado.', 'Nublado.', 'Lluvia']
                    }
df = pd.DataFrame(weekly_data)
imprimir (df)
df["condición"] = df["condición"].map(
    lambda x: x.lstrip(' - ').rstrip('. '))
df["temp_F"] = df["temp"].aplicar(lambda x: 9/5*x+32 )
imprimir (df)
```

Tenga en cuenta que para el ejemplo de código anterior, proporcionamos los mismos datos de entrada que en los ejemplos anteriores, excepto que agregamos caracteres finales e iniciales a la condición. datos de la columna

Consultando filas en un objeto DataFrame

Para consultar filas en función de los valores de una determinada columna, un enfoque común es aplicar un filtro mediante operaciones lógicas AND u OR. Sin embargo, esto se convierte rápidamente en un enfoque desordenado para requisitos simples, como buscar una fila con un valor entre un rango de valores. La biblioteca pandas ofrece una herramienta más limpia: el método `between` , que es algo similar a la palabra clave `between` en SQL.

El siguiente ejemplo de código usa el mismo objeto `Weekly_data` DataFrame que usamos en el ejemplo anterior. Primero, mostraremos el uso de un filtro tradicional, y luego mostraremos el uso del método `between` para consultar las filas que tienen valores de temperatura entre 30 y 40 inclusive:

```
# pandastrick3.py
import pandas como pd

datos_semanales = <igual que en el ejemplo pandas1.py>
df = pd.DataFrame(weekly_data)

print(df[(df.temp >= 30) & (df.temp<=40)])
imprimir(df[df.temp.entre(30,40)])
```

Obtenemos la misma salida de consola para ambos enfoques que usamos. Sin embargo, usar el método `between` es mucho más conveniente que escribir filtros condicionales.

La consulta de filas basadas en datos de texto también es muy compatible con la biblioteca de pandas. Esto se puede lograr usando el descriptor de acceso `str` en las columnas de tipo cadena del objeto DataFrame. Por ejemplo, si queremos buscar filas en nuestro objeto de DataFrame de datos semanales en función de la condición de un día, como lluvioso o soleado, podemos escribir un filtro tradicional o podemos usar el descriptor de acceso `str` en la columna con el contenido método. El siguiente ejemplo de código ilustra el uso de ambas opciones para obtener las filas con Rainy o Sunny como valores de datos en la columna de condición :

```
# pandastrick4.py
import pandas como pd

datos_semanales = <igual que en el ejemplo pandas1.py>
df = pd.DataFrame(weekly_data)
```

```
print(df[(df.condición=='Lluvia') | (df.condición=='Soleado')])  
print(df[df['condición'].str.contains('Lluvia|Soleado')])
```

Si ejecuta el código anterior, encontrará que la salida de la consola es la misma para los dos enfoques que usamos para buscar los datos.

Obtener estadísticas sobre los datos del objeto DataFrame

Para obtener datos estadísticos como la tendencia central, la desviación estándar y la forma, podemos usar el método de descripción . El resultado del método describe para columnas numéricas incluye lo siguiente:

- contar
- media
- desviación estándar
- min
- máx.
- percentiles 25 , percentil 50 , percentil 75

El desglose predeterminado de percentiles se puede cambiar utilizando los percentiles argumento con el desglose deseado.

Si el método de descripción se usa para datos no numéricos, como cadenas, obtendremos recuento, único, superior y frecuencia. El valor superior es el valor más común, mientras que freq es la frecuencia de valor más común. De forma predeterminada, el método de descripción solo evalúa las columnas numéricas a menos que proporcionemos el argumento de inclusión con un valor apropiado .

En el siguiente ejemplo de código, evaluaremos lo siguiente para la misma fecha_semanal

Objeto de trama de datos:

- El uso del método describe con o sin el argumento include
- El uso del argumento percentiles con el método describe
- El uso del método groupby para agrupar datos en una columna y luego usar el método describe encima de ellos

228 Consejos y trucos avanzados en Python

El ejemplo de código completo es el siguiente:

```
# pandastrick5.py
importar pandas como pd
importar numpy como np
pd.set_option('display.max_columns', Ninguno)

datos_semanales = <igual que en el ejemplo pandas1.py>
df = pd.DataFrame(weekly_data)

imprimir(df.describir())
imprimir(df.describe(incluir="todos"))
print(df.describe(percentiles=np.arange(0, 1, 0.1)))
print(df.groupby('condición').describe(percentiles=np.arange(0,
1, 0.1)))
```

Tenga en cuenta que cambiamos las opciones de max_columns para la biblioteca pandas al principio para mostrar todas las columnas que esperábamos en la salida de la consola. Sin esto, algunas de las columnas se truncarán para la salida de la consola del método groupby .

Esto concluye nuestra discusión sobre los trucos avanzados para trabajar con un objeto DataFrame. Este conjunto de trucos y consejos permitirá a cualquier persona comenzar a utilizar la biblioteca de pandas para el análisis de datos. Para conceptos avanzados adicionales, le recomendamos que consulte la documentación oficial de la biblioteca de pandas.

Resumen

En este capítulo, presentamos algunos trucos avanzados que son importantes cuando desea escribir programas eficientes y concisos en Python. Comenzamos con funciones avanzadas como el mapeador, el reductor y las funciones de filtro. También discutimos varios conceptos avanzados de funciones, como funciones internas, funciones lambda y decoradores. Esto fue seguido por una discusión sobre cómo usar estructuras de datos, incluidos diccionarios anidados y comprensiones.

Finalmente, revisamos las operaciones fundamentales de un objeto DataFrame y luego evaluamos algunos casos de uso utilizando algunas operaciones avanzadas del objeto DataFrame.

Este capítulo se centró principalmente en el conocimiento práctico y la experiencia de cómo usar conceptos avanzados en Python. Esto es importante para cualquiera que quiera desarrollar aplicaciones en Python, especialmente para el análisis de datos. Los ejemplos de código proporcionados en este capítulo son muy útiles para comenzar a aprender los trucos avanzados que están disponibles para funciones, estructuras de datos y la biblioteca pandas.

En el próximo capítulo, exploraremos el multiprocesamiento y los subprocessos múltiples en Python.

Preguntas

1. ¿Cuáles de las funciones map, filter y reduce son funciones integradas de Python?
2. ¿Qué son los decoradores estándar?
3. ¿Preferiría una comprensión de generador o una comprensión de lista para un gran conjunto de datos?
4. ¿Qué es un DataFrame en el contexto de la biblioteca pandas?
5. ¿Cuál es el propósito del argumento inplace en los métodos de biblioteca de pandas?

Otras lecturas

- Dominar los patrones de diseño de Python, por Sakis Kasampalis
- Python para análisis de datos, por Wes McKinney
- Análisis práctico de datos con pandas, segunda edición, por Stefanie Molin
- La documentación oficial de Pandas, que está disponible en <https://pandas.pydata.org/docs/>

respuestas

1. Las funciones de mapa y filtro están integradas.
2. Los decoradores estándar son los que no tienen argumentos.
3. En este caso se prefiere la comprensión del generador. Es eficiente en memoria ya que los valores se generan uno por uno.
4. El DataFrame es una representación de datos tabulares, como una hoja de cálculo, y es un objeto de uso común para el análisis de datos utilizando la biblioteca pandas.
5. Cuando el argumento inplace en los métodos de la biblioteca de pandas se establece en True, el resultado de la operación se guarda en el mismo objeto DataFrame en el que se aplica la operación.

Sección 3:

Escalando más allá de un solo subprocesso

En esta parte del libro, nuestro viaje dará un giro hacia la programación de aplicaciones escalables. Un intérprete típico de Python se ejecuta en un solo hilo que se ejecuta en un solo proceso. Para esta parte de nuestro viaje, analizamos cómo escalar Python más allá de este único subprocesso que se ejecuta en un único proceso. Para hacer eso, primero analizamos la programación multiproceso, el multiprocesamiento y la programación asíncrona en una sola máquina. Luego, exploramos cómo podemos ir más allá de la máquina única y ejecutar nuestras aplicaciones en clústeres usando Apache Spark. Después de eso, investigamos el uso de entornos de computación en la nube para centrarnos en la aplicación y dejar la gestión de la infraestructura a los proveedores de la nube.

Esta sección contiene los siguientes capítulos:

- Capítulo 7, Multiprocesamiento, multiproceso y programación asíncrona
- Capítulo 8, Escalado horizontal de Python mediante clústeres
- Capítulo 9, Programación Python para la nube

7

multiprocesamiento, Multiproceso y asíncrono Programación

Podemos escribir código eficiente y optimizado para un tiempo de ejecución más rápido, pero siempre hay un límite en la cantidad de recursos disponibles para los procesos que ejecutan nuestros programas. Sin embargo, aún podemos mejorar el tiempo de ejecución de la aplicación al ejecutar ciertas tareas en paralelo en la misma máquina o en diferentes máquinas. Este capítulo cubrirá el procesamiento paralelo o la concurrencia en Python para las aplicaciones que se ejecutan en una sola máquina.

Cubriremos el procesamiento paralelo utilizando múltiples máquinas en el próximo capítulo. En este capítulo, nos enfocamos en el soporte integrado disponible en Python para la implementación del procesamiento paralelo. Comenzaremos con los subprocessos múltiples en Python y luego hablaremos sobre el multiprocesamiento. Después de eso, discutiremos cómo podemos diseñar sistemas receptivos usando programación asíncrona. Para cada uno de los enfoques, diseñaremos y discutiremos un caso de estudio de implementación de una aplicación simultánea para descargar archivos desde un directorio de Google Drive.

234 Multiprocesamiento, multiproceso y programación asíncrona

Cubriremos los siguientes temas en este capítulo:

- Comprender los subprocessos múltiples en Python y sus limitaciones
- Ir más allá de una sola CPU: implementar multiprocesamiento
- Uso de programación asíncrona para sistemas receptivos

Después de completar este capítulo, conocerá las diferentes opciones para crear aplicaciones multiproceso o multiprocesamiento utilizando bibliotecas integradas de Python. Estas habilidades lo ayudarán a crear no solo aplicaciones más eficientes, sino también aplicaciones para usuarios a gran escala.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Python 3 (3.7 o posterior)
- Una cuenta de Google Drive
- Clave API habilitada para su cuenta de Google Drive

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter07>.

Comenzaremos nuestra discusión con conceptos de subprocessos múltiples en Python.

Comprender los subprocessos múltiples en Python y sus limitaciones

Un subprocesso es una unidad básica de ejecución dentro de un proceso del sistema operativo y consta de su propio contador de programa, una pila y un conjunto de registros. Un proceso de aplicación se puede construir utilizando varios subprocessos que pueden ejecutarse simultáneamente y compartir la misma memoria.

Para subprocessos múltiples en un programa, todos los subprocessos de un proceso comparten código común y otros recursos, como datos y archivos del sistema. Para cada subprocesso, toda su información relacionada se almacena como una estructura de datos dentro del kernel del sistema operativo, y esta estructura de datos se denomina **Bloque de control de subprocessos (TCB)**. El TCB tiene los siguientes componentes principales:

- **Contador de programa (PC)**: Se utiliza para realizar un seguimiento del flujo de ejecución del programa.
- **Registros del sistema (REG)**: estos registros se utilizan para almacenar datos variables.
- **Pila**: La pila es una matriz de registros que gestiona el historial de ejecución.

La anatomía de un hilo se muestra en la Figura 7.1, con tres hilos. Cada subprocesso tiene su propia PC, una pila y REG, pero comparte código y otros recursos con otros subprocessos:

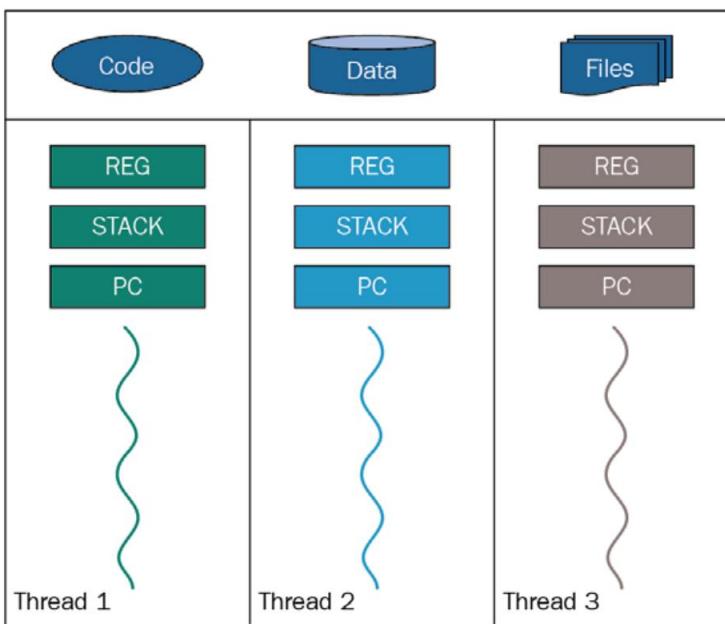


Figura 7.1 – Múltiples subprocessos en un proceso

El TCB también contiene un identificador de subprocesso, el estado del subprocesso (como en ejecución, en espera o detenido) y un puntero al proceso al que pertenece. Multithreading es un concepto de sistema operativo. Es una función que se ofrece a través del kernel del sistema. El sistema operativo facilita la ejecución de múltiples subprocessos simultáneamente en el mismo contexto de proceso, permitiéndoles compartir la memoria del proceso. Esto significa que el sistema operativo tiene el control total de qué subprocesso se activará, en lugar de la aplicación. Necesitamos subrayar este punto para una discusión posterior comparando diferentes opciones de concurrencia.

Cuando los subprocessos se ejecutan en una máquina de una sola CPU, el sistema operativo en realidad cambia la CPU de un subprocesso a otro de modo que los subprocessos parecen estar ejecutándose simultáneamente. ¿Hay alguna ventaja en ejecutar varios subprocessos en una máquina de una sola CPU? La respuesta es sí y no, y depende de la naturaleza de la aplicación. Para las aplicaciones que se ejecutan usando solo la memoria local, puede que no haya ninguna ventaja; de hecho, es probable que muestre un rendimiento más bajo debido a la sobrecarga de cambiar subprocessos en una sola CPU. Pero para las aplicaciones que dependen de otros recursos, la ejecución puede ser más rápida debido a la mejor utilización de la CPU: cuando un subprocesso está esperando otro recurso, otro subprocesso puede utilizar la CPU.

236 Multiprocesamiento, multiproceso y programación asíncrona

Al ejecutar varios subprocesos en multiprocesadores o varios núcleos de CPU, es posible ejecutarlos al mismo tiempo. A continuación, discutiremos las limitaciones de la programación multiproceso en Python.

¿Qué es un punto ciego de Python?

Desde una perspectiva de programación, los subprocesos múltiples son un enfoque para ejecutar diferentes partes de una aplicación al mismo tiempo. Python utiliza varios subprocesos del núcleo que pueden ejecutar los subprocesos de usuario de Python. Pero la implementación de Python (CPython) permite que los subprocesos accedan a los objetos de Python a través de un bloqueo global, que se denomina

Bloqueo de intérprete global (GIL). En palabras simples, GIL es un mutex que permite que solo un subproceso use Python intérprete a la vez y bloquee todos los demás subprocesos. Esto es necesario para proteger el recuento de referencias que se administra para cada objeto en Python de la recolección de elementos no utilizados. Sin dicha protección, el recuento de referencias puede corromperse si varios subprocesos lo actualizan al mismo tiempo. El motivo de esta limitación es proteger las estructuras de datos internas del intérprete y el código C de terceros que no es seguro para subprocesos.

Nota IMPORTANTE

Esta limitación de GIL no existe en Jython y IronPython, que son otras implementaciones de Python.

Esta limitación de Python puede darnos la impresión de que no hay ninguna ventaja en escribir programas de subprocesos múltiples en Python. Esto no es verdad. Todavía podemos escribir código en Python que se ejecute simultáneamente o en paralelo, y lo veremos en nuestro estudio de caso. Multithreading puede ser beneficioso en los siguientes casos:

- **Tareas enlazadas de E/S:** cuando se trabaja con varias operaciones de E/S, siempre hay espacio para mejorar el rendimiento mediante la ejecución de tareas que utilizan más de un subproceso. Cuando un subproceso está esperando una respuesta de un recurso de E/S, liberará el GIL y permitirá que los demás subprocesos funcionen. El subproceso original se activará tan pronto como llegue la respuesta del recurso de E/S.
- **Aplicación de GUI receptiva:** para las aplicaciones de GUI interactivas, es necesario tener un patrón de diseño para mostrar el progreso de las tareas que se ejecutan en segundo plano (por ejemplo, descargar un archivo) y también para permitir que un usuario trabaje en otras funciones de GUI mientras uno o más tareas se ejecutan en segundo plano. Todo esto es posible mediante el uso de subprocesos separados para las acciones iniciadas por un usuario a través de la GUI.

- **Aplicaciones multiusuario:** los subprocessos también son un requisito previo para crear aplicaciones multiusuario. Un servidor web y un servidor de archivos son ejemplos de tales aplicaciones.

Tan pronto como llega una nueva solicitud al subprocesso principal de dicha aplicación, se crea un nuevo subprocesso para atender la solicitud, mientras que el subprocesso principal en la parte posterior escucha una nueva solicitud.

Antes de discutir un estudio de caso de una aplicación de subprocessos múltiples, es importante presentar los componentes clave de la programación de subprocessos múltiples en Python.

Aprender los componentes clave de la programación multiproceso en Python

Multithreading en Python nos permite ejecutar diferentes componentes de un programa al mismo tiempo. Para crear múltiples subprocessos de una aplicación, usaremos el módulo de subprocessos de Python , y los componentes principales de este módulo se describen a continuación.

Comenzaremos discutiendo el módulo de subprocessos en Python primero.

El módulo de roscado

El módulo de subprocessos viene como un módulo estándar y proporciona métodos simples y fáciles de usar para crear múltiples subprocessos de un programa. Debajo del capó, este módulo utiliza el módulo _thread de nivel inferior , que era una opción popular de subprocessos múltiples en la versión anterior de Python.

Para crear un nuevo subprocesso, crearemos un objeto de la clase Subproceso que puede tomar el nombre de una función (a ejecutar) como el atributo de destino y los argumentos que se pasarán a la función como el atributo args . A un subprocesso se le puede dar un nombre que se puede establecer en el momento en que se crea utilizando el argumento de nombre con el constructor.

Después de crear un objeto de la clase Thread , debemos iniciar el hilo utilizando el método de inicio . Para hacer que el programa o subprocesso principal espere hasta que finalicen los objetos de subprocesso recién creados, debemos usar el método de unión . El método de unión se asegura de que el subprocesso principal (un subprocesso de llamada) espere hasta que el subprocesso en el que se llama al método de unión complete su ejecución.

Para explicar el proceso de creación, inicio y espera para finalizar la ejecución de un hilo, crearemos un programa simple con tres hilos. A continuación se muestra un ejemplo de código completo de dicho programa:

```
# thread1.py para crear hilos simples con función
```

```
from threading import current_thread, Thread as Thread
```

```
desde el tiempo de importación del sueño

def imprimir_hola():
    dormir(2)
    imprimir("{}: Hola".format(subproceso_actual().nombre))

def imprimir_mensaje(mensaje):
    dormir(1)
    imprimir("{}: {}".format(subproceso_actual().nombre, mensaje))

# crear hilos
t1 = Subproceso (objetivo = imprimir_hola, nombre = "Th 1")
t2 = Subproceso (objetivo = imprimir_hola, nombre = "Th 2")
t3 = Thread(target=print_message, args=["Buenos días"],
            nombre = "Th 3")

# iniciar los hilos
t1.inicio()
t2.inicio()
t3.inicio()

# esperar a que todo termine
t1.unirse()
t2.unirse()
t3.unirse()
```

En este programa implementamos lo siguiente:

- Creamos dos funciones simples, print_hello e print_message, que serán utilizadas por los subprocesos. Usamos la función de suspensión del módulo de tiempo en ambas funciones para asegurarnos de que las dos funciones terminen su tiempo de ejecución en momentos diferentes.
- Creamos tres objetos Thread . Dos de los tres objetos ejecutarán uno (print_hello) para ilustrar el código compartido por los subprocesos, y el tercer objeto de subproceso utilizará la segunda función (print_message), que también toma un argumento.

- Comenzamos los tres subprocessos uno por uno utilizando el método de inicio .
- Esperamos a que terminara cada subprocesso utilizando el método de unión .

Los objetos Thread se pueden almacenar en una lista para simplificar las operaciones de inicio y unión mediante un bucle for .

La salida de la consola de este programa se verá así:

Jue 3: Buenos días

2: hola

1: hola

El subprocesso 1 y el subprocesso 2 tienen más tiempo de suspensión que el subprocesso 3, por lo que el subprocesso 3 siempre terminará primero. El subprocesso 1 y el subprocesso 2 pueden terminar en cualquier orden dependiendo de quién se apodere primero del procesador.

Nota IMPORTANTE

De forma predeterminada, el método de unión bloquea el subprocesso de la persona que llama indefinidamente. Pero podemos usar un tiempo de espera (en segundos) como argumento para el método de unión . Esto hará que el hilo de la persona que llama se bloquee solo durante el período de tiempo de espera.

Revisaremos algunos conceptos más antes de discutir un estudio de caso más complejo.

Subprocessos de demonios

En una aplicación normal, nuestro programa principal implicitamente espera hasta que todos los demás subprocessos terminen su ejecución. Sin embargo, a veces necesitamos ejecutar algunos subprocessos en segundo plano para que se ejecuten sin bloquear la finalización del programa principal. Estos subprocessos se conocen como **subprocessos daemon**. Estos subprocessos permanecen activos mientras se ejecuta el programa principal (con subprocessos que no son daemon), y está bien terminar los subprocessos daemon una vez que finalizan los subprocessos no daemon. El uso de subprocessos daemon es popular en situaciones en las que no es un problema si un subprocesso muere en medio de su ejecución sin perder ni corromper ningún dato.

Un subprocesso se puede declarar un subprocesso daemon utilizando uno de los dos enfoques siguientes:

- Pase el atributo daemon establecido en True con el constructor (daemon = True).
- Establezca el atributo daemon en True en la instancia del subprocesso (hilo.daemon = Verdadero).

Si un subprocesso se establece como un subprocesso daemon, iniciamos el subprocesso y nos olvidamos de él. El subprocesso se eliminará automáticamente cuando el programa que lo llamó se cierre.

240 Multiprocesamiento, multiproceso y programación asíncrona

El siguiente código muestra el uso de subprocessos tanto daemon como no daemon:

```
#thread2.py para crear subprocessos daemon y no daemon

from threading import current_thread, Thread as Thread from time import sleep

def daeom_func():
    #print(threading.current_thread().isDaemon()) sleep(3) print("{}: Hello from
    daemon".format (current_thread().name))

def nondaeom_func():
    #print(threading.current_thread().isDaemon()) sleep(1) print("{}: Hello from non-
    daemon".format( current_thread().name))

#creando hilos
t1 = Subproceso (objetivo = función_demonio, nombre = "Subproceso de
demonio", demonio = Verdadero)
t2 = Hilo (objetivo = nondaeom_func, nombre = "Hilo no demonio")

# iniciar los hilos
t1.inicio()
t2.inicio()

print("Saliendo del programa principal")
```

En este ejemplo de código, creamos un demonio y un subprocesso que no es demonio. El subprocesso daemon (daeom_func) ejecuta una función que tiene un tiempo de inactividad de 3 segundos, mientras que el subprocesso que no es demonio ejecuta una función (nondaeom_func) que tiene un tiempo de inactividad de 1 segundo. El tiempo de suspensión de las dos funciones se establece para asegurarse de que el subprocesso que no es demonio termine su ejecución primero. La salida de la consola de este programa es la siguiente:

```
Salir del programa principal
Subproceso no daemon: Hola desde no daemon
```

Como no usamos un método de unión en ningún subprocesso, el subprocesso principal sale primero y luego el subprocesso que no es demonio finaliza un poco más tarde con un mensaje de impresión. Pero no hay ningún mensaje de impresión del subprocesso daemon. Esto se debe a que el subprocesso daemon finaliza tan pronto como el subprocesso no daemon finaliza su ejecución. Si cambiamos el tiempo de suspensión en la función `nondaeom_func` a 5, la salida de la consola será la siguiente:

```
Salir del programa principal
Hilo de daemon: Hola de daemon
Subproceso no daemon: Hola desde no daemon
```

Al retrasar la ejecución del subprocesso que no es daemon, nos aseguramos de que el subprocesso daemon finalice su ejecución y no finalice abruptamente.

Nota IMPORTANTE

Si usamos una combinación en el subprocesso del demonio, el subprocesso principal se verá obligado a esperar a que el subprocesso del demonio termine su ejecución.

A continuación, investigaremos cómo sincronizar los hilos en Python.

Sincronizando hilos

La **sincronización de subprocessos** es un mecanismo para garantizar que dos o más subprocessos no ejecuten un bloque de código compartido al mismo tiempo. El bloque de código que normalmente accede a datos compartidos o recursos compartidos también se conoce como **sección crítica**. Este concepto puede quedar más claro a través de la siguiente figura:

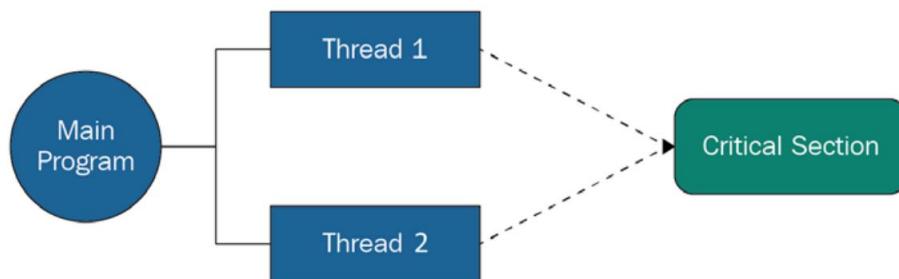


Figura 7.2 – Dos subprocessos accediendo a una sección crítica de un programa

Múltiples subprocessos que acceden a la sección crítica al mismo tiempo pueden intentar acceder o cambiar los datos al mismo tiempo, lo que puede generar resultados impredecibles en los datos. Esta situación se denomina **condición de carrera**.

242 Multiprocesamiento, multiproceso y programación asíncrona

Para ilustrar el concepto de la condición de carrera, implementaremos un programa simple con dos subprocesos, y cada subproceso incrementa una variable compartida 1 millón de veces. Elegimos un número alto para el incremento para asegurarnos de que podemos observar el resultado de la condición de carrera. La condición de carrera también se puede observar usando un valor más bajo para el ciclo de incremento en una CPU más lenta. En este programa, crearemos dos subprocesos que utilizan la misma función (inc en este caso) como destino. El código para acceder a la variable compartida e incrementarla en 1 se encuentra en la sección crítica, y los dos subprocesos acceden a ella sin ninguna protección. El ejemplo de código completo es el siguiente:

```
# thread3a.py cuando no se utiliza sincronización de subprocesos

de subprocessamiento importar subprocesso como subprocesso

def inc():
mundial x
por     __ en el rango (1000000):
x+=1
#variable global
x = 0

# creando hilos
t1 = Subproceso (objetivo = inc, nombre = "Th 1")
t2 = Subproceso (objetivo = inc, nombre = "Th 2")

# iniciar los hilos
t1.inicio()
t2.inicio()

#espera los hilos
t1.unirse()
t2.unirse()
imprimir("valor final de x :", x)
```

El valor esperado de x al final de la ejecución es 2.000.000, que no se observará en la salida de la consola.

Cada vez que ejecutemos este programa, obtendremos un valor diferente de x que es mucho menor que 2,000,000. Esto se debe a la condición de carrera entre los dos subprocessos. Veamos un escenario en el que los subprocessos Th 1 y Th 2 ejecutan la sección crítica ($x+=1$) al mismo tiempo. Ambos hilos preguntarán por el valor actual de x.

Si asumimos que el valor actual de x es 100, ambos subprocessos lo leerán como 100 y lo incrementarán a un nuevo valor de 101. Los dos subprocessos volverán a escribir en la memoria el nuevo valor de 101. Este es un incremento único y, en realidad, los dos hilos deberían incrementar la variable independientemente el uno del otro y el valor final de x debería ser 102. ¿Cómo podemos lograr esto? Aquí es donde la sincronización de subprocessos viene al rescate.

La sincronización de subprocessos se puede lograr mediante el uso de una clase de bloqueo del subprocesamiento módulo. El bloqueo se implementa mediante un objeto **semáforo** proporcionado por el sistema operativo. Un semáforo es un objeto de sincronización a nivel del sistema operativo para controlar el acceso a los recursos y datos para múltiples procesadores e hilos. La clase Lock proporciona dos métodos, adquirir y liberar, que se describen a continuación:

- El método de adquisición se utiliza para adquirir un bloqueo. Un bloqueo puede ser **de bloqueo** (predeterminado) o **no de bloqueo**. En el caso de un bloqueo de bloqueo, la ejecución del subprocesso solicitante se bloquea hasta que el subprocesso de adquisición actual libera el bloqueo. Una vez que el subprocesso de adquisición actual libera el bloqueo (desbloqueado), se proporciona el bloqueo al subprocesso solicitante para continuar. En el caso de una solicitud de adquisición sin bloqueo, la ejecución del subprocesso no se bloquea. Si el bloqueo está disponible (desbloqueado), entonces el bloqueo se proporciona (y bloquea) al subprocesso solicitante para continuar; de lo contrario, el subprocesso solicitante obtiene Falso como respuesta.
- El método de liberación se usa para liberar un bloqueo, lo que significa que restablece el bloqueo a un estado desbloqueado. Si hay algún subprocesso que se bloquea y espera el bloqueo, permitirá que uno de los subprocessos continúe.

El ejemplo de código thread3a.py se revisa con el uso de un candado alrededor de la instrucción de incremento en la variable compartida x. En este ejemplo revisado, creamos un bloqueo en el nivel del subprocesso principal y luego lo pasamos a la función inc para adquirir y liberar un bloqueo alrededor de la variable compartida. El ejemplo de código revisado completo es el siguiente:

```
# thread3b.py cuando se utiliza la sincronización de subprocessos
```

```
desde el bloqueo de importación de subprocessos, subprocesso como subprocesso
```

```
def inc_with_lock (bloqueo):
```

```
mundial x
```

```
por     _   en el rango (1000000):
```

244 Multiprocesamiento, multiproceso y programación asíncrona

```
bloquear.adquirir()
x+=1
bloquear.liberar()

x = 0
mibloqueo = Bloqueo()
# creando hilos
t1 = Subproceso (objetivo = inc_with_lock, args = (mylock,), nombre = "Th 1")

t2 = Thread(target= inc_with_lock, args=(mylock,), name="Th 2")

# iniciar los hilos
t1.inicio()
t2.inicio()

#espera los hilos
t1.unirse()
t2.unirse()

imprimir("valor final de x :", x)
```

Después de usar el objeto Lock , el valor de x siempre es 2000000. El objeto Lock se asegura de que solo un subproceso incremente la variable compartida a la vez. La ventaja de la sincronización de subprocesos es que puede utilizar los recursos del sistema con un rendimiento mejorado y resultados predecibles.

Sin embargo, los bloqueos deben usarse con cuidado porque el uso inadecuado de los bloqueos puede resultar en una situación de interbloqueo. Supongamos que un subproceso adquiere un bloqueo en el recurso A y está esperando adquirir un bloqueo en el recurso B. Pero otro subproceso ya tiene un bloqueo en el recurso B y está buscando adquirir un bloqueo en el recurso A. Los dos subprocesos esperarán el uno al otro para liberarse. las cerraduras, pero nunca sucederá. Para evitar situaciones de interbloqueo, las bibliotecas multihilo y multiprocesamiento vienen con mecanismos como agregar un tiempo de espera para que un recurso mantenga un bloqueo o usar un administrador de contexto para adquirir bloqueos.

Uso de una cola sincronizada

El módulo Queue en Python implementa colas de múltiples productores y múltiples consumidores. Las colas son muy útiles en aplicaciones multihilo cuando la información debe intercambiarse entre diferentes hilos de forma segura. La belleza de la cola sincronizada es que vienen con todos los mecanismos de bloqueo requeridos y no hay necesidad de usar semántica de bloqueo adicional.

Hay tres tipos de colas en el módulo Queue :

- **FIFO:** en la cola FIFO, la tarea agregada primero se recupera primero.
- **LIFO:** en la cola LIFO, la última tarea añadida se recupera primero.
- Cola de **prioridad:** en esta cola, las entradas se ordenan y la entrada con la menor el valor se recupera primero.

Estas colas utilizan bloqueos para proteger el acceso a las entradas de la cola de los subprocessos de la competencia. El uso de una cola con un programa de subprocessos múltiples se ilustra mejor con un ejemplo de código. En el siguiente ejemplo, crearemos una cola FIFO con tareas ficticias en ella. Para procesar las tareas de la cola, implementaremos una clase de subprocesso personalizada al heredar el subprocesso clase. Esta es otra forma de implementar un hilo.

Para implementar una clase de subprocesso personalizada, debemos anular los métodos init y run . En el método init , se requiere llamar al método init de la superclase (la clase Thread). El método de ejecución es la parte de ejecución de la clase de hilo . El ejemplo de código completo es el siguiente:

```
# thread5.py con cola y clase Thread personalizada

de la cola cola de importación
de subprocessamiento importar subprocesso como subprocesso
desde el tiempo de importación del sueño

clase MiTrabajador (Subproceso):
    def __init__(yo, nombre, q):
        subprocessamiento.Subproceso.__init__(auto)
        self.nombre = nombre
        self.cola = q
    def ejecutar (auto):
        mientras que es cierto:
            elemento = self.queue.get()
```

246 Multiprocesamiento, multiproceso y programación asíncrona

```
dormir(1)
tratar:
    imprimir ("{}: {}".format(self.name, item))
por fin:
    self.queue.task_done()

#llenando la cola
micola = Cola()
para i en rango (10):
    myqueue.put("Tarea {}".format(i+1))

# creando hilos
para i en el rango (5):
    trabajador = MiTrabajador("Th {}".format(i+1), micola)
    trabajador.daemon = Verdadero
    trabajador.inicio()

micola.join()
```

En este ejemplo de código, creamos cinco subprocesos de trabajo utilizando la clase de subproceso personalizado (MyThread). Estos cinco subprocesos de trabajo acceden a la cola para obtener el elemento de la tarea. Después de obtener el elemento de la tarea, los subprocesos se suspenden durante 1 segundo y luego imprimen el nombre del subproceso y el nombre de la tarea. Para cada llamada get para un elemento de una cola, una llamada posterior de task_done() indica que el procesamiento de la tarea se ha completado.

Es importante tener en cuenta que usamos el método de unión en el objeto myqueue y no en los subprocesos. El método de unión en la cola bloquea el subproceso principal hasta que todos los elementos de la cola se hayan procesado y completado (se llama a task_done para ellos). Esta es una forma recomendada de bloquear el subproceso principal cuando se usa un objeto de cola para contener los datos de las tareas para los subprocesos.

A continuación, implementaremos una aplicación para descargar archivos de Google Drive utilizando la clase Thread , la clase Queue y un par de bibliotecas de terceros.

Estudio de caso: una aplicación multiproceso para descargar archivos de Google Drive

Hemos discutido en la sección anterior que las aplicaciones de subprocessos múltiples en Python se destacan bien cuando diferentes subprocessos están trabajando en tareas de entrada y salida. Es por eso que seleccionamos implementar una aplicación que descargue archivos de un directorio compartido de Google Drive. Para implementar esta aplicación, necesitaremos lo siguiente:

- **Google Drive:** una cuenta de Google Drive (una cuenta básica gratuita está bien) con una carpeta marcada como compartida.
- **Clave de API:** se requiere una clave de API para acceder a las API de Google. La clave API debe estar habilitado para usar las API de Google para Google Drive. La API se puede habilitar siguiendo las pautas en el sitio de Google Developers (<https://developers.google.com/drive/api/v3/enable-drive-api>).
- **getfilelistpy:** esta es una biblioteca de terceros que obtiene una lista de archivos de Google Drive de la carpeta compartida. Esta biblioteca se puede instalar usando la herramienta pip .
- **gdown:** esta es una biblioteca de terceros que descarga un archivo de Google Drive. Esta biblioteca también se puede instalar a través de la herramienta pip . Hay otras bibliotecas disponibles que ofrecen la misma funcionalidad. Seleccionamos la biblioteca gdown por su facilidad de uso.

Para usar el módulo getfilelistpy , necesitamos crear una estructura de datos de recursos. Esta estructura de datos incluirá un identificador de carpeta como id (este será el ID de la carpeta de Google Drive en nuestro caso), la clave de seguridad API (api_key) para acceder a la carpeta de Google Drive y una lista de atributos de archivo (campos) que se obtendrán cuando obtenemos una lista de archivos. Construimos la estructura de datos de recursos de la siguiente manera:

```
recurso = {  
    "api_key": "AlzaSyDYKmm85kebxddKrGns4z0",  
    "id": "0B8TxHW2Ci6dbckVwTRtTI3RUU",  
    "campos": "archivos (nombre, id, webContentLink)",  
}
```

""La clave de API y la identificación utilizadas en los ejemplos no son originales, por lo que deben reemplazarse según su cuenta y la identificación del directorio compartido""

Limitamos los atributos del archivo a la identificación del archivo, el nombre y su enlace web (URL) únicamente. A continuación, debemos agregar cada elemento del archivo a una cola como una tarea para los subprocessos. Varios subprocessos de trabajo utilizarán la cola para descargar los archivos en paralelo.

248 Multiprocesamiento, multiproceso y programación asíncrona

Para hacer que la aplicación sea más flexible en términos de la cantidad de trabajadores que podemos usar, construimos un grupo de subprocessos de trabajo. El tamaño del grupo está controlado por una variable global que se establece al comienzo del programa. Creamos subprocessos de trabajo según el tamaño del grupo de subprocessos. Cada subprocesso de trabajo en el grupo tiene acceso a la cola, que tiene una lista de archivos. Al igual que el ejemplo de código anterior, cada subprocesso de trabajo tomará un elemento de archivo de la cola a la vez, descargará el archivo y marcará el elemento de archivo como completo mediante el método `task_done`. Un código de ejemplo para definir una estructura de datos de recursos y para definir una clase para el subprocesso de trabajo es el siguiente:

```
#threads_casestudy.py
de la cola cola de importación
de subprocessos de importación Subproceso
tiempo de importación

de getfilelistpy importar getfilelist
importar gdown
THREAD_POOL_SIZE = 1
recurso = {
    "api_key": "AlzaSyDYKmm85kea2bxddKrGns4z0",
    "id": "0B8TxHW2Ci6dbckVweTRtTi3RUU",
    "campos": "archivos (nombre, id, webContentLink)",
}

clase TrabajadorDescarga(Subproceso):

    def __init__(self, nombre, cola):
        Subproceso.__init__(auto)
        self.nombre = nombre
        self.cola = cola

    def ejecutar (auto):
        mientras que es cierto:
            # Obtenga la identificación y el nombre del archivo de la cola
            elemento1 = self.queue.get()
            tratar:
                gdown.download( elemento1['webContentLink'], './files/{}'.format(elemento1['nombre']),
```

```
quiet=False) finalmente: self.queue.task_done()
```

Obtenemos los metadatos de los archivos de un directorio de Google Drive utilizando la estructura de datos de recursos de la siguiente manera:

```
def get_files(recurso): #global files_list  
    res = getfilelist.GetFileList(resource) files_list  
    = res['fileList'][0] return files_list
```

En la función principal , creamos un objeto Queue para insertar metadatos de archivos en la cola.

El objeto Queue se entrega a un grupo de subprocessos de trabajo para descargar los archivos.

Los subprocessos de trabajo descargarán los archivos, como se explicó anteriormente. Usamos la clase de tiempo para medir el tiempo que lleva completar la descarga de todos los archivos del directorio de Google Drive. El código de la función principal es el siguiente:

```
def principal():  
    tiempo_inicio = tiempo.monotónico()  
  
    archivos = get_files(recurso)  
  
    #añadir información de archivos a la cola  
    cola = cola()  
    para elemento en archivos['archivos']:  
        cola.poner(elemento)  
  
    para i en rango (THREAD_POOL_SIZE): trabajador =  
        DownlaodWorker("Hilo {}".format(i+1), cola)  
  
    trabajador.daemon = Verdadero  
    trabajador.inicio()  
  
    queue.join() end_time =  
        time.monotonic()
```

250 Multiprocesamiento, multiproceso y programación asíncrona

```
print('Tiempo de descarga: {} segundos'.
      formato (hora_fin - hora_inicio))
principal()
```

Para esta aplicación, tenemos 10 archivos en el directorio de Google Drive, que varían en tamaño de 500 KB a 3 MB. Ejecutamos la aplicación con 1, 5 y 10 subprocesos de trabajo. El tiempo total necesario para descargar los 10 archivos con 1 subproceso fue de aproximadamente 20 segundos. Esto es casi equivalente a escribir un código sin subprocesos. De hecho, hemos escrito un código para descargar los mismos archivos sin subprocesos y lo hemos puesto a disposición con el código fuente de este libro como ejemplo. El tiempo que se tardó en descargar 10 archivos con una aplicación sin subprocesos fue de aproximadamente 19 segundos.

Cuando cambiamos la cantidad de subprocesos de trabajo a 5, el tiempo necesario para descargar los 10 archivos se redujo significativamente a aproximadamente 6 segundos en nuestra máquina MacBook (Intel Core i5 con 16 GB de RAM). Si ejecuta el mismo programa en su computadora, el tiempo puede ser diferente, pero definitivamente habrá una mejora si aumentamos la cantidad de subprocesos de trabajo. Con 10 subprocesos, observamos que el tiempo de ejecución ronda los 4 segundos.

Esta observación muestra que hay una mejora en el tiempo de ejecución de las tareas vinculadas a E/S mediante el uso de subprocesos múltiples, independientemente de la limitación de GIL que tenga.

Esto concluye nuestra discusión sobre cómo implementar subprocesos en Python y cómo beneficiarse de diferentes mecanismos de bloqueo utilizando la clase Lock y la clase Queue . A continuación, discutiremos la programación de multiprocesamiento en Python.

Más allá de una sola CPU: implementación del multiprocesamiento

Hemos visto la complejidad de la programación multiproceso y sus limitaciones. La pregunta es si la complejidad de los subprocesos múltiples vale la pena. Puede valer la pena para tareas relacionadas con E/S, pero no para casos de uso de aplicaciones generales, especialmente cuando existe un enfoque alternativo. El enfoque alternativo es usar el multiprocesamiento porque los procesos separados de Python no están restringidos por la GIL y la ejecución puede ocurrir en paralelo. Esto es especialmente beneficioso cuando las aplicaciones se ejecutan en procesadores multinúcleo e involucran tareas intensivas que demandan CPU. En realidad, el uso de multiprocesamiento es la única opción en las bibliotecas integradas de Python para utilizar varios núcleos de procesador.

Las unidades de procesamiento de gráficos (GPU) proporcionan una mayor cantidad de núcleos que las CPU normales y se consideran más adecuadas para tareas de procesamiento de datos, especialmente cuando se ejecutan en paralelo. La única salvedad es que para ejecutar un programa de procesamiento de datos en una GPU, tenemos que transferir los datos de la memoria principal a la memoria de la GPU.

Este paso adicional de transferencia de datos se compensará cuando estemos procesando un gran conjunto de datos. Pero habrá poco o ningún beneficio si nuestro conjunto de datos es pequeño. El uso de GPU para el procesamiento de big data, especialmente para entrenar modelos de aprendizaje automático, se está convirtiendo en una opción popular. NVIDIA ha introducido una GPU para el procesamiento en paralelo llamada CUDA, que es compatible con bibliotecas externas en Python.

Cada proceso tiene una estructura de datos denominada **Bloque de control de procesos (PCB)** a nivel del sistema operativo. Al igual que la TCB, la PCB tiene un **ID de proceso (PID)** para la identificación del proceso, almacena el estado del proceso (como en ejecución o en espera) y tiene un contador de programa, registros de CPU, información de programación de CPU y muchos más atributos.

En el caso de múltiples procesos para CPU, no se comparte la memoria de forma nativa. Esto significa que hay una menor probabilidad de corrupción de datos. Si los dos procesos tienen que compartir los datos, necesitan usar algún mecanismo de comunicación entre procesos. Python admite la comunicación entre procesos a través de sus primitivas. En las siguientes subsecciones, primero discutiremos los fundamentos de la creación de procesos en Python y luego discutiremos cómo lograr la comunicación entre procesos.

Creación de múltiples procesos Para la programación

de multiprocesamiento, Python proporciona un paquete de multiprocesamiento que es muy similar al paquete de subprocessos múltiples. El paquete de multiprocesamiento incluye dos enfoques para implementar el multiprocesamiento, que utilizan el objeto Process y el objeto Pool . Discutiremos cada uno de estos enfoques uno por uno.

Uso del objeto Proceso

Los procesos se pueden generar creando un objeto de proceso y luego usando su inicio método similar al método de inicio para iniciar un objeto Thread . De hecho, el Proceso El objeto ofrece la misma API que el objeto Thread . Un ejemplo de código simple para crear múltiples procesos secundarios es el siguiente:

```
# process1.py para crear procesos simples con función  
importar sistema operativo  
desde el proceso de importación de multiprocesamiento, current_process como cp  
desde el tiempo de importación del sueño  
  
def imprimir_hola():
```

252 Multiprocesamiento, multiproceso y programación asíncrona

```
sleep(2) print("{}-{}:  
Hola".format(os.getpid(), cp().nombre))  
  
def print_message(msg): sleep(1)  
    print("{}-{}: {}".format(os.getpid(),  
                           cp().name, msg))  
  
def principal():  
    procesos = []  
  
    # creando proceso  
    procesos.append(Proceso(objetivo=imprimir_hola, nombre="Proceso  
1"))  
    procesos.append(Proceso(objetivo=imprimir_hola, nombre="Proceso  
2"))  
    procesos.append(Proceso(objetivo=imprimir_mensaje, args=["Buenos  
días"], nombre="Proceso 3"))  
  
    # iniciar el proceso  
    para p en procesos:  
        p.inicio()  
  
    # esperar a que todo termine  
    para p en procesos:  
        p.unirse()  
  
    print("Saliendo del proceso principal")  
  
si __nombre__ == '__principal__':  
    principal()
```

Como ya se mencionó, los métodos utilizados para el objeto Process son prácticamente los mismos que los utilizados para el objeto Thread . La explicación de este ejemplo es la misma que para el código de ejemplo en los ejemplos de código de subprocessos múltiples.

Usando el objeto Pool

El objeto Pool ofrece una manera conveniente (utilizando su método de mapa) de crear procesos, asignar funciones a cada nuevo proceso y distribuir parámetros de entrada a través de los procesos. Seleccionamos el ejemplo de código con un tamaño de grupo de 3 pero proporcionamos parámetros de entrada para cinco procesos. La razón para establecer el tamaño del grupo en 3 es asegurarse de que un máximo de tres procesos secundarios estén activos a la vez, independientemente de cuántos parámetros pasemos con el método map del objeto Pool . Los parámetros adicionales se entregarán a los mismos procesos secundarios tan pronto como finalicen su ejecución actual. Aquí hay un ejemplo de código con un tamaño de grupo de 3:

```
# process2.py para crear procesos usando un grupo

importar sistema operativo

desde proceso de importación de multiprocesamiento, Pool, proceso_actual
como cp

desde el tiempo de importación del sueño

def imprimir_mensaje(mensaje):
    dormir(1)
    imprimir("{}-{}: {}".format(os.getpid(), cp().nombre, mensaje))

def principal():
    # proceso de creación desde un grupo
    con Pool(3) como proceso:
        proc.map(imprimir_mensaje, ["Naranja", "Manzana", "Banana",
                                    "Uvas", "Peras"])

    print("Saliendo del proceso principal")

si __nombre__ == '__principal__':
    principal()
```

254 Multiprocesamiento, multiproceso y programación asíncrona

La magia de distribuir parámetros de entrada a una función que está vinculada a un conjunto de procesos de grupo se realiza mediante el método de mapa . El método de mapa espera hasta que todas las funciones completen su ejecución, y es por eso que no es necesario usar un método de unión si los procesos se crean usando el objeto Pool .

En la siguiente tabla se muestran algunas diferencias entre el uso del objeto Process y el objeto Pool :

Using the Pool object	Using the Process object
Only active processes stay in memory	All created processes stay in memory
Works better for large datasets and for repetitive tasks	Works better for small datasets
Processes block on I/O operation until I/O resource is granted	Processes are not blocked on the I/O operation

Tabla 7.1 – Comparación del uso del objeto Pool y el objeto Process

A continuación, discutiremos cómo intercambiar datos entre procesos.

Compartir datos entre procesos

Hay dos enfoques en el paquete de multiprocesamiento para compartir datos entre procesos.

Estos son **la memoria compartida** y **el proceso del servidor**. Se describen a continuación.

Uso de objetos ctype compartidos (memoria compartida)

En este caso, se crea un bloque de memoria compartida y los procesos tienen acceso a este bloque de memoria compartida. La memoria compartida se crea en cuanto iniciamos uno de los ctype tipos de datos disponibles en el paquete de multiprocesamiento . Los tipos de datos son Array y Value. El tipo de datos Array es una matriz ctype y el tipo de datos Value es un objeto ctype genérico , ambos asignados desde la memoria compartida. Para crear un ctype arreglo, usaremos una declaración como la siguiente:

```
milista = multiprocesamiento.Array('i', 5)
```

Esto creará una matriz del tipo de datos entero con un tamaño de 5. i es uno de los códigos de tipo y significa entero. Podemos usar el código de tipo d para tipos de datos flotantes. También podemos inicializar la matriz proporcionando la secuencia como segundo argumento (en lugar del tamaño) de la siguiente manera:

```
milista = multiprocesamiento.Array('i', [1,2,3,4,5])
```

Para crear un objeto Value ctype , usaremos una declaración similar a la siguiente:

```
obj = multiprocesamiento.Valor('i')
```

Esto creará un objeto del tipo de datos entero porque el código de tipo se establece en i. Se puede acceder al valor de este objeto o establecerlo mediante el atributo de valor .

Ambos objetos ctype tienen Lock como argumento opcional, que se establece en True de forma predeterminada. Este argumento, cuando se establece en True , se usa para crear un nuevo objeto de bloqueo recursivo que proporciona acceso sincronizado al valor de los objetos. Si se establece en False, no habrá protección y no será un proceso seguro. Si su proceso solo accede a la memoria compartida con fines de lectura, está bien establecer el Bloqueo en Falso. Dejamos este candado argumento como predeterminado (Verdadero) en nuestros siguientes ejemplos de código.

Para ilustrar el uso de estos objetos ctype de la memoria compartida, crearemos una lista predeterminada con tres valores numéricos, una matriz ctype de tamaño 3 para contener los valores incrementados de la matriz original y un objeto ctype para contener la suma de los matriz incrementada. Estos objetos serán creados por un proceso padre en la memoria compartida y serán accedidos y actualizados por un proceso hijo desde la memoria compartida. Esta interacción de los procesos padre e hijo con la memoria compartida se muestra en la siguiente figura:

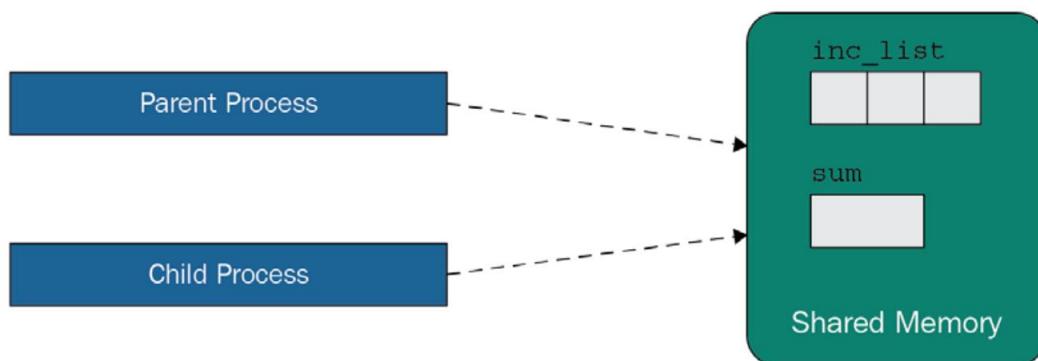


Figura 7.3 – Uso de memoria compartida por un proceso padre e hijo

256 Multiprocesamiento, multiproceso y programación asíncrona

A continuación se muestra un ejemplo de código completo del uso de la memoria compartida:

```
# process3.py para usar objetos ctype de memoria compartida

importar multiprocesamiento
desde proceso de importación de multiprocesamiento, Pool, proceso_actual
como cp

def inc_sum_list(lista, inc_list, suma):
    suma.valor = 0
    para índice, número en enumerar (lista):
        lista_inc[índice] = número + 1
        suma.valor = suma.valor
        + lista_inc[índice]

def principal():

    mylist = [2, 5, 7]

    inc_list = multiprocesamiento.Array('i', 3)

    suma = multiprocesamiento.Valor('i')

    p = Proceso (objetivo = inc_sum_list,
                 args=(mylist, inc_list, sum))

    p.inicio()
    p.unirse()
    print("lista incrementada: ", lista(lista_inc)) print("suma de lista inc: ",
    suma.valor)

    print("Saliendo del proceso principal")

si __nombre__ == '__principal__':
    principal()
```

Tanto el proceso principal como el secundario acceden a los tipos de datos compartidos (inc_list y sum en este caso). Es importante mencionar que el uso de la memoria compartida no es una opción recomendada porque requiere mecanismos de sincronización y bloqueo (similar a lo que hicimos para subprocesos múltiples) cuando varios procesos acceden a los mismos objetos de memoria compartida y el argumento Lock se establece en False .

El siguiente enfoque para compartir datos entre procesos es usar el proceso del servidor.

Usando el proceso del servidor

En este caso, un proceso de servidor se inicia tan pronto como se inicia un programa de Python. Este nuevo proceso se utiliza para crear y administrar los nuevos procesos secundarios solicitados por un proceso principal. Este proceso de servidor puede contener objetos de Python a los que otros procesos pueden acceder mediante servidores proxy.

Para implementar el proceso del servidor y compartir los objetos entre los procesos, el paquete de multiprocesamiento proporciona un objeto Administrador . El objeto Manager admite diferentes tipos de datos, como los siguientes:

- Listas
- Diccionarios
- Cerraduras
- cerraduras
- Colas
- Valores
- Matrices

258 Multiprocesamiento, multiproceso y programación asíncrona

El ejemplo de código que seleccionamos para ilustrar el proceso del servidor crea un diccionario usando el objeto Manager , luego pasa el objeto del diccionario a diferentes procesos secundarios para insertar más datos e imprimir el contenido del diccionario. Crearemos tres procesos secundarios para nuestro ejemplo: dos para insertar datos en el objeto del diccionario y uno para obtener el contenido del diccionario como salida de la consola. La interacción entre el proceso principal, el proceso del servidor y los tres procesos secundarios se muestra en la Figura 7.4.

El proceso principal crea el proceso del servidor tan pronto como se ejecuta una nueva solicitud de proceso utilizando el contexto del Administrador. Los procesos secundarios son creados y administrados por el proceso del servidor. Los datos compartidos están disponibles dentro del proceso del servidor y todos los procesos pueden acceder a ellos, incluido el proceso principal:

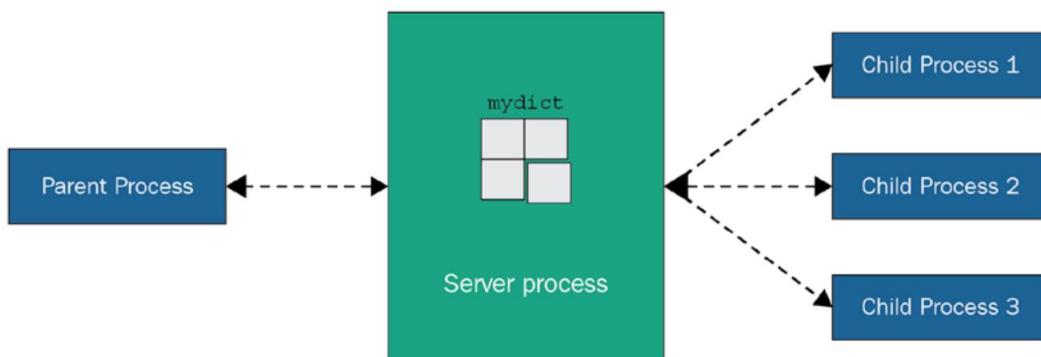


Figura 7.4 – Uso del proceso del servidor para compartir datos entre procesos

El ejemplo de código completo se muestra a continuación:

```
# process4.py para usar la memoria compartida usando el proceso del servidor
```

```
importar multiprocesamiento
```

```
desde el proceso de importación de multiprocesamiento, Administrador
```

```
def insert_data (dict1, código, asunto):
```

```
dict1[código] = sujeto
```

```
salida de definición (dict1):
```

```
print("Datos del diccionario: ", dict1)
```

```
def principal():
```

```
con multiprocessing.Manager() como administrador:
```

```
# crear un diccionario en el proceso del servidor
```

```
mydict = mgr.dict({100: "Matemáticas", 200: "Ciencias"})

p1 = Proceso (objetivo = insertar_datos, args = (mydict, 300,
    "Inglés"))
p2 = Proceso (objetivo = insertar_datos, args = (mydict, 400,
    "Francés"))
p3 = Proceso (objetivo = salida, args = (mydict,))

p1.inicio()
p2.inicio()

p1.unirse()
p2.unirse()

p3.inicio()
p3.unirse()

print("Saliendo del proceso principal")

si __nombre__ == '__principal__':
    principal()
```

El enfoque de proceso de servidor ofrece más flexibilidad que el enfoque de memoria compartida porque admite una gran variedad de tipos de objetos. Sin embargo, esto tiene el costo de un rendimiento más lento en comparación con el enfoque de memoria compartida.

En la siguiente sección, exploraremos las opciones de comunicación directa entre los procesos.

Intercambio de objetos entre procesos

En la sección anterior, estudiamos cómo compartir datos entre los procesos a través de un bloque de memoria externa o un nuevo proceso. En esta sección, investigaremos el intercambio de datos entre procesos usando objetos de Python. El módulo de multiprocesamiento proporciona dos opciones para este propósito. Estos están usando el objeto Queue y el objeto Pipe .

Uso del objeto Queue El objeto

Queue está disponible en el paquete de multiprocesamiento y es casi igual que el objeto de cola sincronizada (queue.Queue) que usamos para subprocessos múltiples.

Este objeto Queue es seguro para el proceso y no requiere ninguna protección adicional. Un ejemplo de código para ilustrar el uso del objeto Queue de multiprocesamiento para el intercambio de datos se muestra a continuación:

```
# process5.py para usar la cola para intercambiar datos

import multiprocessing
from multiprocessing import Queue, Process

def copy_data (lista, myqueue):
    for num in lista:
        myqueue.put(num)

def salida(myqueue): while not
    myqueue.empty(): print(myqueue.get())

def principal():
    milista = [2, 5, 7] micola = Queue()
    p1 = Process(target=copy_data, args=(milista, micola)) p2 = Process(target=salida,
    args=(micola,))

    p1.start()
    p2.start()

    print("La cola está vacía: ",myqueue.empty()) print("Saliendo del proceso
    principal")

if __name__ == '__main__':
    principal()
```

En este ejemplo de código, creamos un objeto de lista estándar y una Cola de multiprocesamiento objeto. Los objetos de lista y cola se pasan a un nuevo proceso, que se adjunta a una función llamada copy_data. Esta función copiará los datos del objeto de lista al objeto de Cola . Se inicia un nuevo proceso para imprimir el contenido del objeto Queue .

Tenga en cuenta que los datos en el objeto Queue están establecidos por el proceso anterior y los datos estarán disponibles para el nuevo proceso. Esta es una forma conveniente de intercambiar datos sin agregar la complejidad de la memoria compartida o el proceso del servidor.

Usar el objeto Tubería

El objeto Pipe es como una tubería entre dos procesos para el intercambio de datos. Es por esto que este objeto es especialmente útil cuando se requiere una comunicación bidireccional. Cuando creamos un objeto de tubería , proporciona dos objetos de conexión, que son los dos extremos de la tubería . objeto. Cada objeto de conexión proporciona un método de envío y uno de recepción para enviar y recibir datos.

Para ilustrar el concepto y el uso del objeto Pipe , crearemos dos funciones que se adjuntarán a dos procesos separados:

- La primera función es para enviar el mensaje a través de una conexión de objeto Pipe . Envaremos algunos mensajes de datos y finalizaremos la comunicación con un mensaje BYE .
- La segunda función es recibir el mensaje usando el otro objeto de conexión del objeto Tubería . Esta función se ejecutará en un ciclo infinito hasta que reciba un mensaje BYE .

Las dos funciones (o procesos) se proporcionan con los dos objetos de conexión de una tubería.

El código completo es el siguiente:

```
# process6.py para usar Pipe para intercambiar datos
```

```
de proceso de importación de multiprocesamiento, Tubería
```

```
def mysender(s_conn):  
    s_conn.send({100, "Matemáticas"})  
    s_conn.send({200, "Ciencia"})  
    s_conn.send("ADIOS")
```

```
def mireceptor(r_conn):  
    mientras que es cierto:  
        mensaje = r_conn.recv()
```

262 Multiprocesamiento, multiproceso y programación asíncrona

```
si mensaje == "ADIOS":  
    romper  
    print("Mensaje recibido: ", msj)  
  
def principal():  
    sender_conn, receiver_conn= Tubería()  
  
    p1 = Proceso (objetivo = mysender, args = (sender_conn,))  
    p2 = Proceso (objetivo = myreceiver, args = (receiver_conn,))  
  
    p1.inicio()  
    p2.inicio()  
  
    p1.unirse()  
    p2.unirse()  
  
    print("Saliendo del proceso principal")  
  
    si __nombre__ == '__principal__':  
        principal()
```

Es importante mencionar que los datos en un objeto Pipe pueden corromperse fácilmente si los dos procesos intentan leer o escribir en él usando el mismo objeto de conexión al mismo tiempo. Es por eso que las colas de multiprocesamiento son la opción preferida: porque brindan una sincronización adecuada entre los procesos.

Sincronización entre procesos

La sincronización entre procesos asegura que dos o más procesos no accedan a los mismos recursos o código de programa al mismo tiempo, lo que también se denomina **sección crítica**. Tal situación puede conducir a una condición de carrera, que puede corromper los datos. La posibilidad de que ocurra una condición de carrera entre diferentes procesos no es muy alta, pero aún es posible si están usando memoria compartida o accediendo a los mismos recursos. Estas situaciones se pueden evitar usando los objetos apropiados con sincronización incorporada o usando el objeto Lock , similar a lo que usamos en el caso de subprocessos múltiples.

Ilustramos el uso de colas y tipos de datos ctype con Lock establecido en True, que es seguro para el proceso. En el siguiente ejemplo de código, ilustraremos el uso del objeto Lock para asegurarnos de que un proceso tenga acceso a la salida de la consola a la vez. Creamos los procesos usando el objeto Pool y para pasar el mismo objeto Lock a todos los procesos, usamos el Lock del objeto Manager y no el del paquete de multiprocesamiento. También usamos la función parcial para vincular el objeto Lock a cada proceso, junto con una lista que se distribuirá a cada función de proceso. Aquí está el ejemplo de código completo:

```
# process7.py para mostrar sincronización y bloqueo
de functools import parcial
desde grupo de importación de multiprocesamiento, administrador

def printme (bloqueo, mensaje):
    bloquear.adquirir()
    tratar:
        imprimir (mensaje)
    por fin:
        bloquear.librear()

def principal():
    con Pool(3) como proceso:
        bloquear = Administrador(). Bloquear()
        func = parcial(printme,lock)
        proc.map(func, ["Naranja", "Manzana", "Banana",
                        "Uvas","Peras"])

    print("Saliendo del proceso principal")

    si __nombre__ == '__principal__':
        principal()
```

Si no usamos el objeto Lock , la salida de los diferentes procesos puede mezclarse.

264 Multiprocesamiento, multiproceso y programación asíncrona

Estudio de caso: una aplicación multiprocesador para descargar archivos de Google Drive

En esta sección, implementaremos el mismo caso de estudio que hicimos en el Caso de estudio: una aplicación de subprocessos múltiples para descargar archivos de la sección Google Drive, pero usando procesadores en su lugar. Los requisitos previos y los objetivos son los mismos que se describen para el estudio de caso de la aplicación multiproceso.

Para esta aplicación, usamos el mismo código que creamos para la aplicación de subprocessos múltiples, excepto que usamos procesos en lugar de subprocessos. Otra diferencia es que usamos el objeto JoinableQueue del módulo de multiprocesamiento para lograr la misma funcionalidad que obteníamos del objeto Queue normal . El código para definir una estructura de datos de recursos y una función para descargar archivos de Google Drive es el siguiente:

```
#procesos_estudiodecaso.py
tiempo de importación
desde el proceso de importación de multiprocesamiento, JoinableQueue
de getfilelistpy importar getfilelist
importar gdown
PROCESOS_POOL_SIZE = 5

recurso = {
    "api_key": "AlzaSyDYKmm85keqnk4bF1Da2bxddKrGns4z0",
    "id": "0B8TxHW2Ci6dbckVwetTIV3RUU",
    "campos": "archivos (nombre, id, webContentLink)",
}

def mydownloader( cola):
    mientras que es cierto:
        # Obtenga la identificación y el nombre del archivo de la cola
        item1 = cola.get()
        tratar:
            gdown.download(item1['webContentLink'],
            './archivos/{}'.format(elemento1['nombre']),
            tranquilo = falso)
        por fin:
            cola.tarea_hecha()
```

Obtenemos los metadatos de los archivos, como el nombre y el enlace HTTP, de un directorio de Google Drive utilizando la estructura de datos de recursos de la siguiente manera:

```
def get_files(recurso): res =  
    getFilelist.GetFileList(resource) files_list = res['fileList'][0] return  
    files_list
```

En nuestra función principal , creamos un objeto JoinableQueue e insertamos los metadatos de los archivos en la cola. La cola se entregará a un grupo de procesos para descargar los archivos.

Los procesos descargarán los archivos. Usamos la clase de tiempo para medir el tiempo que lleva descargar todos los archivos del directorio de Google Drive. El código de la función principal es el siguiente:

```
def principal ():  
  
    archivos = get_files(recurso) #agregar  
    información de archivos a la cola  
    myqueue = JoinableQueue() for item in  
    files['files']:  
        myqueue.put(elemento)  
  
    procesos = []  
    for id in range(PROCESSES_POOL_SIZE): p =  
        Process(target=mydownloader, args=(myqueue,))  
        p.daemon = True p.start()  
  
    start_time = time.monotonic() myqueue.join()  
    total_exec_time = time.monotonic() - start_time  
  
    print(f'Tiempo de descarga: {total_exec_time:.2f} segundos')  
  
    si __nombre__ == '__principal__':  
        principal()
```

266 Multiprocesamiento, multiproceso y programación asíncrona

Ejecutamos esta aplicación variando la cantidad diferente de procesos, como 3, 5, 7 y 10. Descubrimos que el tiempo que se tarda en descargar los mismos archivos (como en el caso de estudio de subprocessos múltiples) es ligeramente mejor que con el aplicación multiproceso. El tiempo de ejecución variará de una máquina a otra, pero en nuestra máquina (MacBook Pro: Intel Core i5 con 16 GB de RAM), tomó alrededor de 5 segundos con 5 procesos y 3 segundos con 10 procesos ejecutándose en paralelo. Esta mejora de 1 segundo con respecto a la aplicación multiproceso está en línea con los resultados esperados, ya que el multiprocesamiento proporciona una verdadera concurrencia.

Uso de programación asíncrona para sistemas receptivos

Con el multiprocesamiento y la programación de subprocessos múltiples, nos ocupamos principalmente de la programación síncrona, en la que solicitamos algo y esperamos a que se reciba la respuesta antes de pasar al siguiente bloque de código. Si se aplica algún cambio de contexto, lo proporciona el sistema operativo. La programación asíncrona en Python es diferente principalmente en los siguientes dos aspectos:

- Las tareas se crearán para ejecución asíncrona. Esto significa que la persona que llama principal no tiene que esperar la respuesta de otro proceso. El proceso responderá a la persona que llama una vez que termine la ejecución.
- El sistema operativo ya no administra el cambio de contexto entre los procesos y los subprocessos. El programa asíncrono recibirá solo un hilo en un proceso, pero podemos hacer varias cosas con él. En este estilo de ejecución, cada proceso o tarea libera voluntariamente el control cuando está inactivo o esperando otro recurso para asegurarse de que las otras tareas obtengan su turno. Este concepto se denomina **multitarea cooperativa**.

La multitarea cooperativa es una herramienta eficaz para lograr la concurrencia a nivel de aplicación. En la multitarea cooperativa, no construimos procesos o subprocessos, sino tareas, que comprenden **tasklets**, corrutinas o **subprocesos verdes**. Estas tareas están coordinadas por una única función conocida como **bucle de eventos**. El bucle de eventos registra las tareas y maneja el flujo de control entre las tareas. La belleza del ciclo de eventos en Python es que se implementa usando generadores, y estos generadores pueden ejecutar una función y pausarla en un punto específico (usando el rendimiento) mientras mantienen la pila de objetos bajo control antes de que se reanude.

Para un sistema basado en la multitarea cooperativa, siempre existe la duda de cuándo devolver el control a un programador o a un bucle de eventos. La lógica más utilizada es usar la operación de E/S como el evento para liberar el control porque siempre hay un tiempo de espera cuando estamos realizando una operación de E/S.

Pero espera, ¿no es la misma lógica que usamos para subprocessos múltiples?

Descubrimos que los subprocessos múltiples mejoran el rendimiento de la aplicación cuando se trata de operaciones de E/S. Pero hay una diferencia aquí. En el caso de subprocessos múltiples, el sistema operativo administra el cambio de contexto entre los subprocessos y puede adelantarse a cualquier subprocesso en ejecución por cualquier motivo y otorgar el control a otro subprocesso. Pero en la programación asíncrona o la multitarea cooperativa, las tareas o corrutinas no son visibles para los sistemas operativos y no se pueden adelantar. De hecho, las corrutinas no pueden ser reemplazadas por el bucle de eventos principal. Pero esto no significa que el sistema operativo no pueda adelantarse a todo el proceso de Python. El proceso principal de Python sigue compitiendo por los recursos con otras aplicaciones y procesos a nivel del sistema operativo.

En la siguiente sección, analizaremos algunos componentes básicos de la programación asíncrona en Python, que proporciona el módulo `asyncio`, y concluiremos con un estudio de caso completo.

Entendiendo el módulo `asyncio`

El módulo `asyncio` está disponible en Python 3.5 o posterior para escribir programas simultáneos usando la sintaxis `async/await`. Pero se recomienda usar Python 3.7 o posterior para crear cualquier aplicación `asyncio` seria. La biblioteca es rica en funciones y admite la creación y ejecución de corrutinas de Python, la realización de operaciones de E/S de red, la distribución de tareas a las colas y la sincronización de código concurrente.

Comenzaremos con cómo escribir y ejecutar rutinas y tareas.

Corrutinas y tareas

Las rutinas son las funciones que se van a ejecutar de forma asíncrona. Un ejemplo simple de enviar una cadena a la salida de la consola usando una rutina es el siguiente:

```
#asyncio1.py para construir una rutina básica
```

```
importar asyncio
```

```
tiempo de importación
```

```
asíncrono def decir (retraso, mensaje):
```

```
    esperar asyncio.sleep (retraso)
```

```
    imprimir (mensaje)
```

268 Multiprocesamiento, multiproceso y programación asíncrona

```
print("Comenzó a las ", hora.strftime("%X"))
asyncio.run(decir(1,"Bien"))
asyncio.run(decir(2, "Mañana"))
print("Detenido en ", time.strftime("%X"))
```

En este ejemplo de código, es importante tener en cuenta lo siguiente:

- La corutina toma argumentos de retraso y mensaje . El argumento de retraso se utiliza para agregar un retraso antes de enviar la cadena de mensaje a la salida de la consola.
- Utilizamos la función asyncio.sleep en lugar de la tradicional time.sleep función. Si se utiliza la función time.sleep , no se devolverá el control al bucle de eventos. Por eso es importante utilizar el asyncio compatible. función de sueño
- La corutina se ejecuta dos veces con dos valores diferentes de la demora argumento utilizando el método de ejecución . El método de ejecución no ejecutará las corrutinas al mismo tiempo.

La salida de la consola de este programa será la siguiente. Esto muestra que las corrutinas se ejecutan una tras otra ya que el retraso total agregado es de 3 segundos:

```
Comenzó a las 15:59:55
Bueno
Mañana
Detenido a las 15:59:58
```

Para ejecutar las corrutinas en paralelo, necesitamos usar la función create_task del módulo asyncio . Esta función crea una tarea que se puede usar para programar rutinas para que se ejecuten simultáneamente.

El siguiente ejemplo de código es una versión revisada de asyncio1.py, en la que envolvimos la rutina (digamos en nuestro caso) en una tarea usando la función create_task . En esta versión revisada, creamos dos tareas que envuelven la corutina de decir . Esperamos a que se completaran las dos tareas usando la palabra clave await :

```
#asyncio2.py para construir y ejecutar rutinas en paralelo
import asyncio
tiempo de importación

asíncrono def decir (retraso, mensaje):
    esperar asyncio.sleep (retraso)
```

```
imprimir (mensaje)

asíncrono definición principal ():
    tarea1 = asyncio.create_task (decir(1, 'Bien'))
    tarea2 = asyncio.create_task(decir(1, 'Mañana'))
    print("Comenzó a las ", hora.strftime("%X"))
    esperar tarea1
    esperar tarea2
    print("Detenido en ", time.strftime("%X"))

asyncio.run(principal())
```

La salida de la consola de este programa es la siguiente:

```
Comenzó a las 16:04:40
Bueno
Mañana
Detenido a las 16:04:41
```

Esta salida de la consola muestra que las dos tareas se completaron en 1 segundo, lo que prueba que las tareas se ejecutan en paralelo.

Uso de objetos disponibles

Un objeto es awaitable si podemos aplicarle la instrucción await . La mayoría de las funciones y módulos de asyncio que contiene están diseñados para trabajar con objetos disponibles.

Pero la mayoría de los objetos de Python y las bibliotecas de terceros no están diseñados para la programación asíncrona. Es importante seleccionar bibliotecas compatibles que proporcionen objetos disponibles para usar al crear aplicaciones asíncronas.

Los objetos disponibles se dividen principalmente en tres tipos: corrutinas, tareas y **futuros**.

Ya discutimos las corrutinas y las tareas. Un futuro es un objeto de bajo nivel que es como un mecanismo de devolución de llamada que se usa para procesar el resultado proveniente de async/await. Los objetos Future normalmente no están expuestos para la programación a nivel de usuario.

270 Multiprocesamiento, multiproceso y programación asíncrona

Ejecutar tareas al mismo tiempo

Si tenemos que ejecutar varias tareas en paralelo, podemos usar la palabra clave await como hicimos en el ejemplo anterior. Pero hay una mejor manera de hacer esto utilizando el método de recopilación función. Esta función ejecutará los objetos disponibles en la secuencia proporcionada. Si alguno de los objetos en espera es una rutina, se programará como una tarea. Veremos el uso de la función de recopilación en la siguiente sección con un ejemplo de código.

Distribuir tareas usando colas

El objeto Queue en el paquete asyncio es similar al módulo Queue pero no es seguro para subprocessos. El módulo asyncio proporciona una variedad de implementaciones de colas, como colas FIFO, colas prioritarias y colas LIFO. Las colas en el módulo asyncio se pueden usar para distribuir las cargas de trabajo a las tareas.

Para ilustrar el uso de una cola con tareas, escribiremos un pequeño programa que simulará el tiempo de ejecución de una función real durmiendo durante un período de tiempo aleatorio. El tiempo de inactividad aleatorio se calcula para 10 de estas ejecuciones y el proceso principal lo agrega a un objeto Queue como elementos de trabajo. El objeto Queue se pasa a un grupo de tres tareas.

Cada tarea en el grupo ejecuta la rutina asignada, que consume el tiempo de ejecución según la entrada de la cola disponible para ella. El código completo se muestra a continuación:

```
#asincio3.py para distribuir el trabajo a través de la cola
```

```
importar asyncio
```

```
importar al azar
```

```
tiempo de importación
```

```
ejecutor asíncrono def (nombre, cola):
```

```
    mientras que es cierto:
```

```
        exec_time = cola de espera.get()
```

```
        esperar asyncio.sleep (exec_time)
```

```
        cola.tarea_hecha()
```

```
        #print(f'{name} ha tomado {exec_time:.2f} segundos')
```

```
asíncrono definición principal ():
```

```
    micola = asyncio.Cola()
```

```
    calc_exuction_time = 0
```

```
    por _ en el rango (10):
```

```
        sleep_for = random.uniform(0.4, 0.8)
```

```
        calc_exuction_time += sleep_for
```

```
myqueue.put_nowait(sleep_for)

tareas = []
para id en el rango (3):
    tarea = asyncio.create_task(ejecutor(f'Tarea-{id+1}', micola))

tareas.append(tarea)

tiempo_inicio = tiempo.monotónico()
esperar myqueue.join()
total_exec_time = time.monotonic() - start_time

para tarea en tareas:
    tarea.cancel()

esperar asyncio.gather(*tareas, return_exceptions=True)

print(f"Tiempo de ejecución calculado {calc_exuection_time:0.2f}")

print(f"Tiempo de ejecución real {total_exec_time:0.2f}")

asyncio.run(principal())
```

Usamos la función put_no_wait del objeto Queue porque es una operación sin bloqueo. La salida de la consola de este programa es la siguiente:

```
Tiempo de ejecución calculado 5.58
Tiempo de ejecución real 2.05
```

Esto muestra claramente que las tareas se ejecutan en paralelo y la ejecución es tres veces mejor que si las tareas se ejecutan secuencialmente.

Hasta ahora, hemos cubierto los conceptos fundamentales del paquete asyncio en Python. Antes de concluir este tema, revisaremos el caso de estudio que hicimos para la sección de subprocessos múltiples al implementarlo usando las tareas asyncio .

Caso práctico: aplicación asyncio para descargar archivos de Google Drive

Implementaremos el mismo caso de estudio que hicimos en el Caso de estudio: una aplicación de subprocessos múltiples para descargar archivos de la sección Google Drive, pero utilizando el módulo asyncio con `async`, `await` y `async queue`. Los requisitos previos para este estudio de caso son los mismos, excepto que usamos la biblioteca `aiohttp` y `aiofiles` en lugar de `gdown` biblioteca. La razón es simple: la biblioteca `gdown` no está construida como un módulo asíncrono. No hay ningún beneficio de usarlo con programación asíncrona. Este es un punto importante a tener en cuenta al seleccionar bibliotecas para usarlas con aplicaciones asincrónicas.

Para esta aplicación, creamos una corriente, `mydownloader`, para descargar un archivo de Google Drive usando los módulos `aiohttp` y `aiofiles`. Esto se muestra en el siguiente código, y se resalta el código que es diferente de los casos de estudio anteriores:

```
#asincio_casestudy.py
import asyncio
tiempo de importación

importar archivos aio, aiohttp
de getfilelistpy importar getfilelist

TAREA_POOL_SIZE = 5

recurso = {
    "api_key": "AlzaSyDYKmm85keqnk4bF1DpYa2dKrGns4z0",
    "id": "0B8TxHW2Ci6dbckVwetTIV3RUU",
    "campos": "archivos (nombre, id, webContentLink)",
}

async def mydownloader (nombre, cola):
    mientras que es cierto:
        # Obtenga la identificación y el nombre del archivo de la cola
        item = cola de espera.get()
        tratar:
            asíncrono con aiohttp.ClientSession() como sess:
                asíncrono con sess.get(item['webContentLink'])
            como resp:
                si resp.estado == 200:
```

```
f = esperar aiofiles.open('./files/{}'.format(  
    elemento['nombre']), modo='wb')  
esperar f.escribir(esperar resp.leer())  
esperar f.cerrar()  
finalmente: print(f"{{name}}:  
Descarga completada para ",item['name'])  
  
cola.tarea_hecha()
```

El proceso para obtener la lista de archivos de una carpeta compartida de Google Drive es el mismo que usamos en el estudio de caso anterior para subprocessos múltiples y procesamiento múltiple. En este caso de estudio, creamos un grupo de tareas (configurable) basado en la corutina mydownloader. Luego, estas tareas se programan para ejecutarse juntas, y nuestro proceso principal espera a que todas las tareas completen su ejecución. Un código para obtener una lista de archivos de Google Drive y luego descargar los archivos usando tareas asyncio es el siguiente:

```
def get_files(recurso): res =  
    getfilelist.GetFileList(resource) files_list = res['fileList'][0] return files_list  
  
  
  
  
  
async def main ():  
    archivos = get_files (recurso) #agregar  
    información de archivos a la cola myqueue =  
    asyncio.Queue()  
    para elemento en archivos['archivos']:  
        myqueue.put_nowait(elemento)  
  
  
    tareas = []  
    for id in range(TASK_POOL_SIZE): tarea =  
        asyncio.create_task( mydownloader(f'Task-{id+1}',  
            myqueue)) tareas.append(tarea)  
  
  
  
  
    start_time = time.monotonic() await myqueue.join()  
    total_exec_time = time.monotonic() - start_time
```

274 Multiprocesamiento, multiproceso y programación asíncrona

```
para tarea en tareas:  
    tarea.cancel()  
  
esperar asyncio.gather(*tareas, return_exceptions=True)  
  
print(f'Tiempo de descarga: {total_exec_time:.2f} segundos')  
  
asyncio.run(principal())
```

Ejecutamos esta aplicación variando la cantidad de tareas, como 3, 5, 7 y 10. Descubrimos que el tiempo que tomó descargar los archivos con las tareas asyncio es menor que cuando descargamos los mismos archivos usando el enfoque de subprocesamiento múltiple o el enfoque de multiprocesamiento. Los detalles exactos del tiempo empleado con el enfoque de subprocesos múltiples y el enfoque de procesamiento múltiple están disponibles en las secciones Estudio de caso: una aplicación de subprocesos múltiples para descargar archivos de Google Drive y Estudio de caso: una aplicación de multiprocesador para descargar archivos de Google Drive.

El tiempo de ejecución puede variar de una máquina a otra, pero en nuestra máquina (MacBook Pro: Intel Core i5 con 16 GB de RAM), tomó alrededor de 4 segundos con 5 tareas y 2 segundos con 10 tareas ejecutándose en paralelo. Esta es una mejora significativa en comparación con los números que observamos para los estudios de casos de subprocesos múltiples y procesamiento múltiple. Esto está en línea con los resultados esperados, ya que asyncio proporciona un mejor marco de concurrencia cuando se trata de tareas relacionadas con E/S, pero debe implementarse utilizando el conjunto correcto de objetos de programación.

Esto concluye nuestra discusión sobre la programación asíncrona. Esta sección proporcionó todos los ingredientes básicos para construir una aplicación asíncrona usando el paquete `asyncio`.

Resumen

En este capítulo, discutimos diferentes opciones de programación concurrente en Python utilizando las bibliotecas estándar. Comenzamos con subprocesos múltiples con una introducción a los conceptos básicos de la programación concurrente. Presentamos los desafíos con subprocesos múltiples, como GIL, que permite que solo un subproceso a la vez acceda a los objetos de Python. Los conceptos de bloqueo y sincronización se exploraron con ejemplos prácticos de código Python. También discutimos los tipos de tareas para las que la programación de subprocesos múltiples es más efectiva usando un estudio de caso.

Estudiamos cómo lograr la concurrencia usando múltiples procesos en Python. Con la programación de multiprocesamiento, aprendimos cómo compartir datos entre procesos que usan memoria compartida y el proceso del servidor, y también cómo intercambiar objetos de forma segura entre procesos que usan el objeto Queue y el objeto Pipe . Al final, construimos el mismo caso de estudio que hicimos para el ejemplo de subprocesos múltiples, pero usando procesos en su lugar. Luego, introdujimos un enfoque completamente diferente para lograr la concurrencia mediante el uso de programación asíncrona. Este fue un cambio completo en el concepto, y lo comenzamos observando los conceptos de alto nivel de las palabras clave `async` y `await` y cómo crear tareas, o corrutinas, usando el paquete `asyncio` . Concluimos el capítulo con el mismo estudio de caso que examinamos para multiprocesamiento y subprocesos múltiples pero usando programación asíncrona.

Este capítulo ha proporcionado muchos ejemplos prácticos de cómo implementar aplicaciones concurrentes en Python. Este conocimiento es importante para cualquier persona que desee crear aplicaciones asíncronas o de subprocesos múltiples utilizando las bibliotecas estándar disponibles en Python.

En el próximo capítulo, exploraremos el uso de bibliotecas de terceros para crear aplicaciones simultáneas en Python.

Preguntas

1. ¿Qué coordina los hilos de Python? ¿Es un intérprete de Python?
2. ¿Qué es el GIL en Python?
3. ¿Cuándo debería usar subprocesos daemon?
4. Para un sistema con memoria limitada, ¿debemos usar un objeto de proceso o un objeto de grupo para crear procesos?
5. ¿Qué son los futuros en el paquete asyncio ?
6. ¿Qué es un bucle de eventos en la programación asíncrona?
7. ¿Cómo se escribe una rutina o función asíncrona en Python?

276 Multiprocesamiento, multiproceso y programación asíncrona

Otras lecturas

- Aprendizaje de concurrencia en Python por Elliot Forbes
- Programación experta en Python por Michal Jaworski y Tarek Ziade
- Programación orientada a objetos de Python 3, segunda edición por Dusty Phillips
- Dominar la concurrencia en Python por Quan Nguyen
- Concurrencia de Python con asyncio por Matheu Fowler

respuestas

1. Los subprocessos y los procesos están coordinados por el kernel del sistema operativo.
2. El GIL de Python es un mecanismo de bloqueo utilizado por Python para permitir que solo se ejecute un subprocesso a la vez.
3. Los subprocessos Daemon se utilizan cuando no es un problema que un subprocesso se termine una vez su hilo principal termina.
4. El objeto Pool mantiene solo los procesos activos en la memoria, por lo que es una mejor opción.
5. Los futuros son como un mecanismo de devolución de llamada que se utiliza para procesar el resultado proveniente de llamadas asíncronas/en espera.
6. Un objeto de bucle de eventos realiza un seguimiento de las tareas y maneja el flujo de control entre ellas.
7. Podemos escribir una rutina asíncrona comenzando con `async def`.

8

Escalando Python Uso de clústeres

En el capítulo anterior, discutimos el procesamiento paralelo para una sola máquina usando hilos y procesos. En este capítulo, ampliaremos nuestra discusión sobre el procesamiento en paralelo de una sola máquina a varias máquinas en un clúster. Un clúster es un grupo de dispositivos informáticos que trabajan juntos para realizar tareas de computación intensiva, como el procesamiento de datos.

En particular, estudiaremos las capacidades de Python en el área de computación intensiva en datos. La computación intensiva en datos generalmente usa clústeres para procesar grandes volúmenes de datos en paralelo. Aunque hay bastantes marcos y herramientas disponibles para la computación intensiva en datos, nos centraremos en **Apache Spark** como motor de procesamiento de datos y en PySpark como biblioteca de Python para crear dichas aplicaciones.

Si Apache Spark con Python está correctamente configurado e implementado, el rendimiento de su aplicación puede multiplicarse y superar las plataformas de la competencia, como **Hadoop MapReduce**. También veremos cómo se utilizan los conjuntos de datos distribuidos en un entorno agrupado. Este capítulo lo ayudará a comprender el uso de plataformas informáticas de clúster para el procesamiento de datos a gran escala y cómo implementar aplicaciones de procesamiento de datos utilizando Python. Para ilustrar el uso práctico de Python para aplicaciones con requisitos de computación en clúster, incluiremos dos casos de estudio; el primero es calcular el valor de Pi (π) y el segundo es generar una nube de palabras a partir de un archivo de datos.

Cubriremos los siguientes temas en este capítulo:

- Aprender sobre las opciones de clúster para el procesamiento en paralelo
- Introducir conjuntos de **datos distribuidos resilientes (RDD)** • Usar PySpark para el procesamiento de datos en paralelo
- Estudios de casos de uso de Apache Spark y PySpark

Al final de este capítulo, sabrá cómo trabajar con Apache Spark y cómo puede escribir aplicaciones de Python para el procesamiento de datos que se pueden ejecutar en los nodos trabajadores de un clúster de Apache Spark.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Python 3.7 o posterior instalado en su computadora
- Un clúster de un solo nodo de Apache Spark
- PySpark instalado sobre Python 3.7 o posterior para el desarrollo de programas de controladores

Nota

La versión de Python que se usa con Apache Spark tiene que coincidir con la versión de Python que se usa para ejecutar el programa del controlador.

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter08>.

Comenzaremos nuestra discusión observando las opciones de clúster disponibles para el procesamiento paralelo en general.

Información sobre las opciones de clúster para el procesamiento en paralelo

Cuando tenemos un gran volumen de datos para procesar, no es eficiente y, a veces, incluso no es factible usar una sola máquina con múltiples núcleos para procesar los datos de manera eficiente. Esto es especialmente un desafío cuando se trabaja con transmisión de datos en tiempo real. Para tales escenarios, necesitamos varios sistemas que puedan procesar datos de manera distribuida y realizar estas tareas en varias máquinas en paralelo. El uso de múltiples máquinas para procesar tareas intensivas de cómputo en paralelo y de manera distribuida se denomina **computación en clúster**. Hay varios marcos distribuidos de big data disponibles para coordinar la ejecución de trabajos en un clúster, pero Hadoop MapReduce y Apache Spark son los principales contendientes en esta carrera. Ambos marcos son proyectos de código abierto de Apache. Hay muchas variantes (por ejemplo, Databricks) de estas dos plataformas disponibles con características complementarias y soporte de mantenimiento, pero los fundamentos siguen siendo los mismos.

Si observamos el mercado, la cantidad de implementaciones de Hadoop MapReduce puede ser mayor en comparación con Apache Spark, pero con su creciente popularidad, Apache Spark eventualmente cambiará las tornas. Dado que Hadoop MapReduce sigue siendo muy relevante debido a su gran base instalada, es importante discutir qué es exactamente Hadoop MapReduce y cómo Apache Spark se está convirtiendo en una mejor opción. Veamos una descripción general rápida de los dos en las siguientes subsecciones.

Mapa de HadoopReducir

Hadoop es un marco de procesamiento distribuido de propósito general que ofrece la ejecución de trabajos de procesamiento de datos a gran escala en cientos o miles de nodos informáticos en un clúster de Hadoop. Los tres componentes principales de Hadoop se incluyen en la siguiente figura:

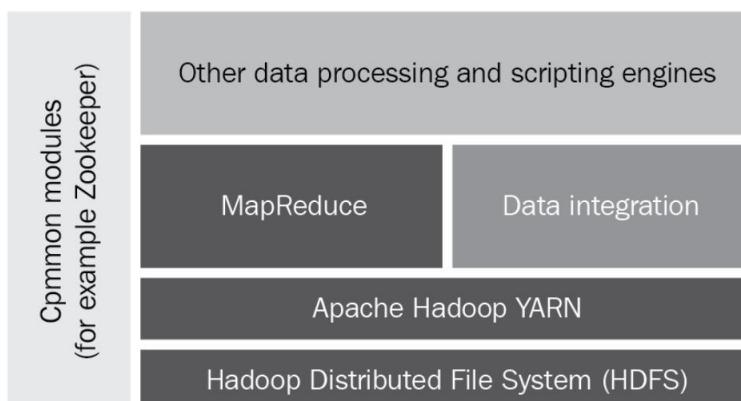


Figura 8.1 – Ecosistema Apache Hadoop MapReduce

Los tres componentes principales se describen a continuación:

- **Sistema de archivos distribuidos de Hadoop (HDFS):** este es un sistema de archivos nativo de Hadoop que se utiliza para almacenar archivos de modo que esos archivos se puedan parallelizar en un clúster.
- **Otro negociador de recursos (YARN):** este es un sistema que procesa datos almacenados en HDFS y programa los trabajos enviados (para el procesamiento de datos) para que los ejecute un sistema de procesamiento. Los sistemas de procesamiento se pueden utilizar para procesamiento de gráficos, procesamiento de flujo o procesamiento por lotes.
- **MapReduce:** este es un marco de programación que nos permite procesar grandes conjuntos de datos al distribuir los datos en varios conjuntos de datos pequeños. El marco MapReduce procesa los datos utilizando dos tipos de funciones: mapeador y reductor.
Los roles individuales de las funciones mapeador (mapa) y reductor (reducir) son los mismos que discutimos en el Capítulo 6, Consejos y trucos avanzados en Python. La diferencia clave es que usamos muchas funciones de mapa y reducción en paralelo para procesar varios conjuntos de datos al mismo tiempo.

Después de dividir el conjunto de datos grande en conjuntos de datos pequeños, podemos proporcionar los conjuntos de datos pequeños como entrada para muchas funciones de mapeo para su procesamiento en diferentes nodos de un clúster de Hadoop. Cada función de mapeador toma un conjunto de datos como entrada, procesa los datos en función del objetivo establecido por el programador y produce la salida como pares clave-valor. Una vez que la salida de todos los conjuntos de datos pequeños esté disponible, una o varias funciones de reducción tomarán la salida de las funciones del mapeador y agregarán los resultados según los objetivos de las funciones de reducción.

Para explicarlo con un poco más de detalle, podemos tomar un ejemplo de contar palabras particulares como ataque y arma en una gran fuente de datos de texto. Los datos de texto se pueden dividir en pequeños conjuntos de datos, por ejemplo, ocho conjuntos de datos. Podemos tener ocho funciones de mapeo que cuentan las dos palabras dentro del conjunto de datos que se les proporciona. Cada función del mapeador nos proporciona el conteo de las palabras ataque y arma como salida para el conjunto de datos que se le proporcionó. En la siguiente fase, las salidas de todas las funciones del mapeador se proporcionan a dos funciones reductoras, una para cada palabra. Cada función reductora agrega el recuento de cada palabra y proporciona los resultados agregados como salida. El funcionamiento del marco MapReduce para este ejemplo de conteo de palabras se muestra a continuación. Tenga en cuenta que la función de mapeador generalmente se implementa como map y la función de reducción como reduce en la programación de Python:

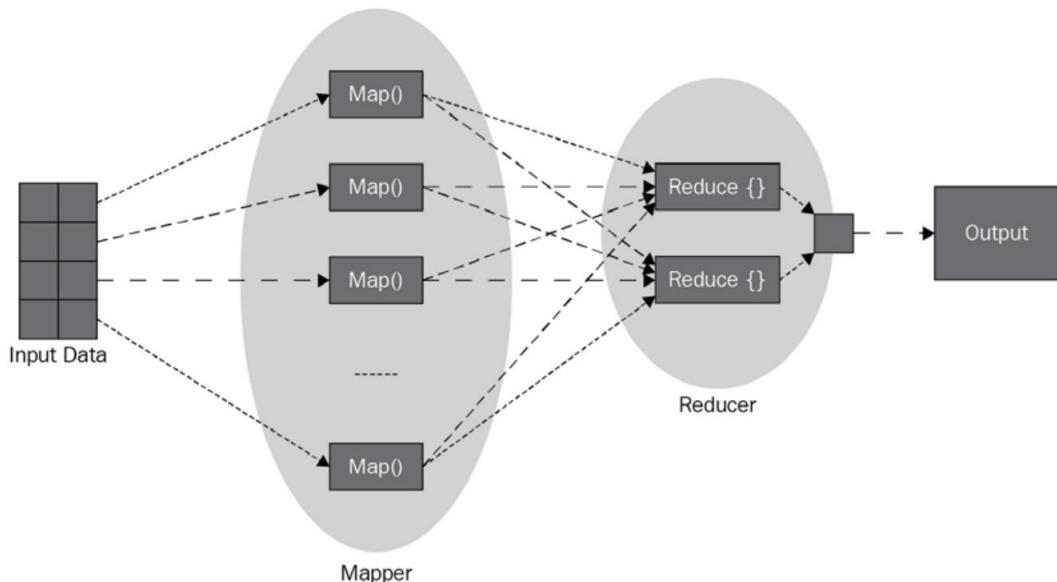


Figura 8.2 – Funcionamiento del marco MapReduce

Omitiremos los siguientes niveles de los componentes de Hadoop, ya que no son relevantes para nuestra discusión en este capítulo. Hadoop está construido principalmente en Java, pero se puede usar cualquier lenguaje de programación, como Python, para escribir componentes de mapeador y reductor personalizados para el módulo MapReduce.

Hadoop MapReduce es bueno para procesar una gran cantidad de datos dividiéndolos en pequeños bloques. Los nodos del clúster procesan estos bloques por separado y luego los resultados se agregan antes de enviarlos al solicitante. Hadoop MapReduce procesa los datos de un sistema de archivos y, por lo tanto, no es muy eficiente en términos de rendimiento. Sin embargo, funciona muy bien si la velocidad de procesamiento no es un requisito crítico, por ejemplo, si el procesamiento de datos se puede programar para que se realice durante la noche.

chispa apache

Apache Spark es un marco informático de clúster de código abierto para el procesamiento de datos en tiempo real y por lotes. La característica principal de Apache Spark es que es un marco de procesamiento de datos en memoria, lo que lo hace eficiente en términos de lograr una baja latencia y lo hace adecuado para muchos escenarios del mundo real debido a los siguientes factores adicionales:

- Obtiene resultados rápidamente para aplicaciones de misión crítica y sensibles al tiempo como como escenarios en tiempo real o casi en tiempo real.
- Es bueno para realizar tareas de manera repetida o iterativa de manera eficiente debido al procesamiento en memoria.
- Puede utilizar algoritmos de aprendizaje automático listos para usar. • Puede aprovechar el soporte de lenguajes de programación adicionales como Java, Python, Scala y R.

De hecho, Apache Spark cubre una amplia gama de cargas de trabajo, incluidos datos por lotes, procesamiento iterativo y transmisión de datos. La belleza de Apache Spark es que también puede usar Hadoop (a través de YARN) como un clúster de implementación, pero también tiene su propio administrador de clústeres.

A un alto nivel, los componentes principales de Apache Spark están segregados en tres capas, como se muestra en la siguiente figura:

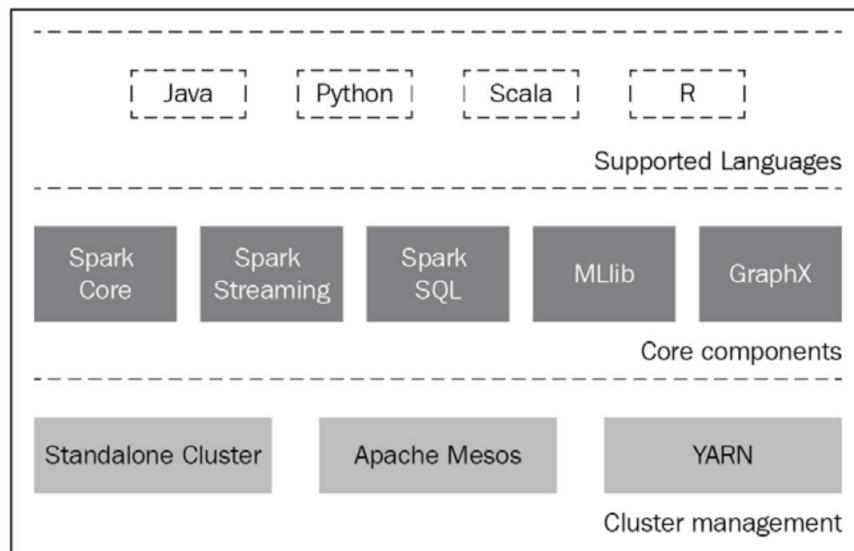


Figura 8.3 – Ecosistema Apache Spark

Estas capas se analizan a continuación.

Idiomas de soporte

Scala es un lenguaje nativo de Apache Spark, por lo que es bastante popular para el desarrollo.

Apache Spark también proporciona API de alto nivel para Java, Python y R. En Apache Spark, la compatibilidad con varios idiomas se proporciona mediante la interfaz de llamada a **procedimiento remoto (RPC)**.

Hay un adaptador RPC escrito para cada idioma en Scala que transforma las solicitudes del cliente escritas en un idioma diferente a las solicitudes nativas de Scala. Esto facilita su adopción en toda la comunidad de desarrollo.

Componentes principales

Una breve descripción de cada uno de los componentes principales se analiza a continuación:

- **Spark Core y RDD:** Spark Core es un motor central de Spark y es responsable de proporcionar abstracción a los RDD, programar y distribuir trabajos a un clúster, interactuar con sistemas de almacenamiento como HDFS, Amazon S3 o un RDBMS, y administrar la memoria y recuperaciones de fallas. Un RDD es un conjunto de datos distribuido resistente que es una colección de datos inmutable y distribuible. Los RDD están particionados para ejecutarse en los diferentes nodos de un clúster. Discutiremos los RDD con más detalle en la siguiente sección.

- **Spark SQL:** este módulo es para consultar datos almacenados tanto en RDD como en fuentes de datos externas mediante interfaces abstractas. El uso de estas interfaces comunes permite a los desarrolladores combinar los comandos SQL con las herramientas de análisis para una aplicación determinada.

- **Spark Streaming:** este módulo se utiliza para procesar datos en tiempo real, lo cual es fundamental para analizar flujos de datos en vivo con baja latencia.

- **MLlib:** la **biblioteca de aprendizaje automático (MLlib)** se utiliza para aplicar el aprendizaje automático algoritmos en Apache Spark.

- **GraphX:** este módulo proporciona una API para computación paralela basada en gráficos. Este módulo viene con una variedad de algoritmos gráficos. Tenga en cuenta que un gráfico es un concepto matemático basado en vértices y aristas que representa cómo un conjunto de objetos se relaciona o depende entre sí. Los objetos están representados por vértices y sus relaciones por los bordes.

Gestión de clústeres

Apache Spark admite algunos administradores de clústeres, como Standalone, Mesos, YARN y Kubernetes. La función clave de un administrador de clústeres es programar y ejecutar los trabajos en los nodos del clúster y administrar los recursos en los nodos del clúster. Pero para interactuar con uno o más administradores de clústeres, se usa un objeto especial en el programa principal o controlador llamado **SparkSession**. Antes de la versión 2.0, el objeto **SparkContext** se consideraba un punto de entrada, pero su API ahora se incluye como parte del objeto **SparkSession**.

Conceptualmente, la siguiente figura muestra la interacción de un administrador de clústeres, **SparkSession (SparkContext)** y los **nodos trabajadores** en un clúster:

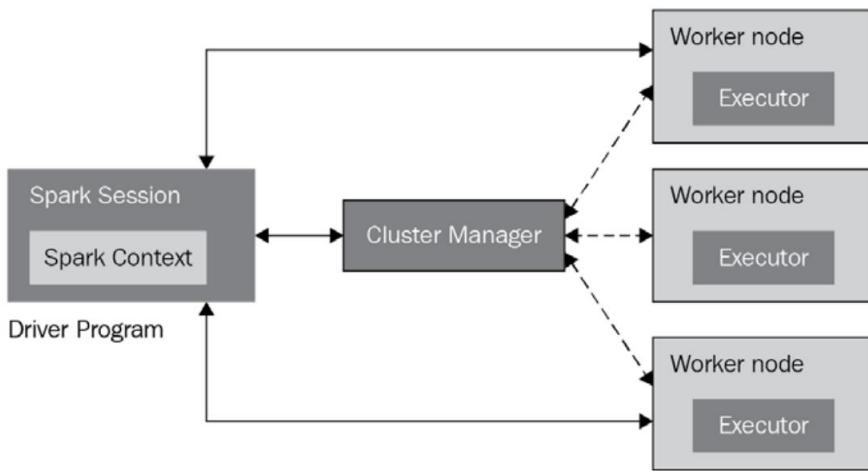


Figura 8.4 – Ecosistema Apache Spark

El objeto **SparkSession** puede conectarse a diferentes tipos de administradores de clústeres. Una vez conectados, los ejecutores se adquieren en los nodos del clúster a través de los administradores del clúster. Los ejecutores son los procesos de Spark que ejecutan los trabajos y almacenan los resultados del trabajo computacional. El administrador de clústeres en un nodo principal es responsable de enviar el código de la aplicación a los procesos ejecutores en los nodos trabajadores. Una vez que el código y los datos de la aplicación (si corresponde) se mueven a los nodos trabajadores, el objeto **SparkSession** en un programa controlador interactúa directamente con los procesos ejecutores para la ejecución de tareas.

Según la versión 3.1 de Apache Spark, se admiten los siguientes administradores de clústeres:

- **Independiente:** este es un administrador de clúster simple que se incluye como parte del motor Spark Core. El clúster independiente se basa en procesos maestros y trabajadores (o esclavos). Un proceso maestro es básicamente un administrador de clústeres y los procesos de trabajo alojan a los ejecutores. Aunque los maestros y los trabajadores se pueden alojar en una sola máquina, este no es el escenario de implementación real de un clúster autónomo de Spark. Se recomienda distribuir trabajadores a diferentes máquinas para obtener el mejor resultado. El clúster independiente es fácil de configurar y proporciona la mayoría de las funciones requeridas de un clúster.

- **Apache Mesos:** este es otro administrador de clústeres de propósito general que también puede ejecutar Hadoop MapReduce. Para entornos de clústeres a gran escala, Apache Mesos es la opción preferida. La idea de este administrador de clústeres es que agrega los recursos físicos en un solo recurso virtual que actúa como un clúster y proporciona una abstracción a nivel de nodo. Es un administrador de clúster distribuido por diseño.
- **Hadoop YARN:** este administrador de clústeres es específico de Hadoop. Este también es un marco distribuido por naturaleza.
- **Kubernetes:** Esto está más en la fase experimental. El propósito de esto cluster manager es automatizar la implementación y el escalado de las aplicaciones en contenedores. La última versión de Apache Spark incluye el programador de Kubernetes.

Antes de concluir esta sección, vale la pena mencionar otro marco, **Dask**, que es una biblioteca de código abierto escrita en Python para computación paralela. El marco Dask funciona directamente con plataformas de hardware distribuidas como Hadoop. El marco Dask utiliza bibliotecas probadas en la industria y proyectos de Python como NumPy, pandas y scikit learn. Dask es un marco pequeño y liviano en comparación con Apache Spark y puede manejar clústeres de tamaño pequeño a mediano. En comparación, Apache Spark admite varios idiomas y es la opción más adecuada para clústeres a gran escala.

Después de presentar las opciones de clúster para computación paralela, analizaremos en la siguiente sección la estructura de datos central de Apache Spark, que es el RDD.

Introducción a los RDD

El RDD es la estructura de datos central en Apache Spark. Esta estructura de datos no es solo una colección distribuida de objetos, sino que también se divide de tal manera que cada conjunto de datos se puede procesar y calcular en diferentes nodos de un clúster. Esto convierte al RDD en un elemento central del procesamiento de datos distribuidos. Además, un objeto RDD es resistente en el sentido de que es tolerante a fallas y el marco puede reconstruir los datos en caso de falla. Cuando creamos un objeto RDD, el nodo maestro replica el objeto RDD en múltiples ejecutores o nodos trabajadores. Si algún proceso ejecutor o nodo trabajador falla, el nodo maestro detecta el error y permite que un proceso ejecutor en otro nodo se haga cargo de la ejecución. El nuevo nodo ejecutor ya tendrá una copia del objeto RDD y podrá iniciar la ejecución inmediatamente. Todos los datos procesados por el nodo ejecutor original antes de fallar se perderán y serán calculados nuevamente por el nuevo nodo ejecutor.

En las siguientes subsecciones, aprenderemos sobre dos operaciones clave de RDD y cómo crear objetos de RDD a partir de diferentes fuentes de datos.

Aprendiendo operaciones RDD

Un RDD es un objeto inmutable, lo que significa que una vez que se crea, no se puede modificar. Pero se pueden realizar dos tipos de operaciones sobre los datos de un RDD. Estas son **transformaciones** y **acciones**. Estas operaciones se describen a continuación.

Transformaciones

Estas operaciones se aplican en un objeto RDD y dan como resultado la creación de un nuevo objeto RDD. Este tipo de operación toma un RDD como entrada y produce uno o más RDD como salida. También debemos recordar que estas transformaciones son de naturaleza perezosa. Esto significa que solo se ejecutarán cuando se dispare una acción sobre ellos, que es otro tipo de operación. Para explicar el concepto de evaluación perezosa, podemos suponer que estamos transformando datos numéricos en un RDD restando 1 de cada elemento y luego sumando aritméticamente (la acción) todos los elementos al RDD de salida del paso de transformación. Debido a la evaluación perezosa, la operación de transformación no ocurrirá hasta que llamemos a la operación de acción (la suma, en este caso).

Hay varias funciones de transformación integradas disponibles con Apache Spark. Las funciones de transformación comúnmente utilizadas son las siguientes:

- mapa: la función de mapa itera cada elemento o línea de un objeto RDD y aplica la función de mapa definida para cada elemento.
- filtro: esta función filtrará los datos del RDD original y proporcionará una nueva RDD con los resultados filtrados.
- unión: esta función se aplica a dos RDD si son del mismo tipo y da como resultado la producción de otro RDD que es una unión de los RDD de entrada.

Comportamiento

Las acciones son operaciones computacionales que se aplican en un RDD y los resultados de tales operaciones deben devolverse al programa controlador (por ejemplo, SparkSession). Hay varias funciones de acción integradas disponibles con Apache Spark. Las funciones de acción comúnmente utilizadas son las siguientes:

- contar: La acción de contar devuelve el número de elementos en un RDD.
- recopilar: esta acción devuelve todo el RDD al programa del controlador.
- reducir: Esta acción reducirá los elementos de un RDD. Un ejemplo simple es una operación de suma en un conjunto de datos RDD.

Para obtener una lista completa de las funciones de transformación y acción, le sugerimos que consulte la documentación oficial de Apache Spark. A continuación, estudiaremos cómo crear RDD.

Creación de objetos RDD

Existen tres enfoques principales para crear objetos RDD, que se describen a continuación.

Paralelización de una colección

Este es uno de los enfoques más simples utilizados en Apache Spark para crear RDD.

En este enfoque, se crea o se carga una colección en un programa y luego se pasa al método de paralelización del objeto `SparkContext`. Este enfoque no se utiliza más allá del desarrollo y las pruebas. Esto se debe a que requiere que un conjunto de datos completo esté disponible en una máquina, lo cual no es conveniente para una gran cantidad de datos.

Conjuntos de datos externos

Apache Spark admite conjuntos de datos distribuidos desde un sistema de archivos local, HDFS, HBase o incluso Amazon S3. En este enfoque de creación de RDD, los datos se cargan directamente desde una fuente de datos externa. Hay métodos convenientes disponibles con `SparkContext` objeto que se puede utilizar para cargar todo tipo de datos en RDD. Por ejemplo, el archivo de texto El método se puede usar para cargar datos de texto desde recursos locales o remotos mediante una URL adecuada (por ejemplo, `file://`, `hdfs://` o `s3n://`).

De los RDD existentes

Como se discutió anteriormente, los RDD se pueden crear mediante operaciones de transformación. Este es uno de los diferenciadores de Apache Spark de Hadoop MapReduce. El RDD de entrada no se cambia porque es un objeto inmutable, pero se pueden crear nuevos RDD a partir de RDD existentes.

Ya hemos visto algunos ejemplos de cómo crear RDD a partir de RDD existentes utilizando las funciones de mapa y filtro .

Esto concluye nuestra introducción de RDD. En la siguiente sección, proporcionaremos más detalles con ejemplos de código Python utilizando la biblioteca PySpark.

Uso de PySpark para el procesamiento de datos en paralelo

Como se discutió anteriormente, Apache Spark está escrito en lenguaje Scala, lo que significa que no hay soporte nativo para Python. Hay una gran comunidad de científicos de datos y expertos en análisis que prefieren usar Python para el procesamiento de datos debido al amplio conjunto de bibliotecas disponibles con Python. Por lo tanto, no es conveniente pasar a usar otro lenguaje de programación solo para el procesamiento de datos distribuidos. Por lo tanto, la integración de Python con Apache Spark no solo es beneficiosa para la comunidad de ciencia de datos, sino que también abre las puertas para muchos otros que deseen adoptar Apache Spark sin aprender o cambiar a un nuevo lenguaje de programación.

La comunidad de Apache Spark ha creado una biblioteca de Python, **PySpark**, para facilitar el trabajo con Apache Spark mediante Python. Para que el código de Python funcione con Apache Spark, que se basa en Scala (y Java), se ha desarrollado una biblioteca de Java, **Py4J**. Esta biblioteca Py4J se incluye con PySpark y permite que el código Python interactúe con los objetos JVM.

Esta es la razón por la que cuando instalamos PySpark, primero debemos tener JVM instalado en nuestro sistema.

PySpark ofrece casi las mismas características y ventajas que Apache Spark. Estos incluyen computación en memoria, la capacidad de paralelizar cargas de trabajo, el uso del patrón de diseño de evaluación perezoso y soporte para múltiples administradores de clústeres como Spark, YARN y Mesos.

La instalación de PySpark (y Apache Spark) está fuera del alcance de este capítulo. El enfoque de este capítulo es discutir el uso de PySpark para utilizar el poder de Apache Spark y no cómo instalar Apache Spark y PySpark. Pero vale la pena mencionar algunas opciones de instalación y dependencias.

Hay muchas guías de instalación disponibles en línea para cada versión de Apache Spark/PySpark y las diversas plataformas de destino (por ejemplo, Linux, macOS y Windows).

PySpark está incluido en el lanzamiento oficial de Apache Spark, que ahora se puede descargar desde el sitio web de Apache Spark (<https://spark.apache.org/>). PySpark también está disponible a través de la utilidad pip de PyPI, que se puede usar para una configuración local o para conectarse a un clúster remoto. Otra opción al instalar PySpark es usar **Anaconda**, que es otro sistema popular de administración de paquetes y entornos. Si estamos instalando PySpark junto con Apache Spark, necesitamos que lo siguiente esté disponible o instalado en la máquina de destino:

- JVM
- Escala
- chispa apache

Para los ejemplos de código que se discutirán más adelante, hemos instalado Apache Spark versión 3.1.1 en macOS con PySpark incluido. PySpark viene con el **shell de PySpark**, que es una CLI para la API de PySpark. Cuando se inicia el shell de PySpark, inicializa automáticamente los objetos `SparkSession` y `SparkContext`, que se pueden usar para interactuar con el motor principal de Apache Spark. La siguiente figura muestra la inicialización del shell PySpark:

Figura 8.5 – Shell de PySpark

De los pasos de inicialización del shell PySpark, podemos observar lo siguiente:

- El objeto `SparkContext` ya está creado y su instancia está disponible en el concha como `sc`.
 - También se crea el objeto `SparkSession` y su instancia está disponible como `chispa`. Ahora, `SparkSession` es un punto de entrada al marco PySpark para crear dinámicamente objetos `RDD` y `DataFrame`. El objeto `SparkSession` también se puede crear mediante programación, y hablaremos de esto más adelante con un ejemplo de código.

• Apache Spark viene con una interfaz de usuario web y un servidor web para alojar la interfaz de usuario web, y se inicia en `http://192.168.1.110:4040` para la instalación de nuestra máquina local.

Tenga en cuenta que la dirección IP mencionada en esta URL es una dirección privada que es específica de nuestra máquina. Apache Spark selecciona el puerto 4040 como el puerto predeterminado. Si este puerto está en uso, Apache Spark intentará hospedarse en el próximo puerto disponible, como 4041 o 4042.

En las siguientes subsecciones, aprenderemos cómo crear objetos `SparkSession`, exploraremos PySpark para operaciones RDD y aprenderemos a usar PySpark DataFrames y PySpark SQL. Comenzaremos con la creación de una sesión de Spark usando Python.

Creación de programas SparkSession y SparkContext

Antes de la versión 2.0 de Spark, SparkContext se usaba como punto de entrada a PySpark. Desde la versión 2.0 de Spark, SparkSession se ha introducido como un punto de entrada al marco subyacente de PySpark para trabajar con RDD y DataFrames. SparkSession también incluye todas las API disponibles en SparkContext, SQLContext, StreamingContext y HiveContext. Ahora, SparkSession también se puede crear usando la clase SparkSession usando su método de construcción , que se ilustra en el siguiente ejemplo de código:

```
importar pyspark
desde pyspark.sql importar SparkSession
chispa1 = SparkSession.builder.master("local[2]")
 appName('Nueva aplicación').getOrCreate()
```

Cuando ejecutamos este código en el shell de PySpark, que ya tiene una SparkSession predeterminada objeto creado como chispa, devolverá la misma sesión como salida de este constructor método. La siguiente salida de la consola muestra la ubicación de los dos SparkSession objetos (chispa y chispa1), lo que confirma que apuntan al mismo objeto SparkSession :

```
>>> chispa
<objeto pyspark.sql.session.SparkSession en 0x1091019e8>
>>> chispa1
<objeto pyspark.sql.session.SparkSession en 0x1091019e8>
```

Algunos conceptos clave para entender con respecto al método de construcción son los siguientes:

- **getOrCreate:** este método es la razón por la que obtendremos un session en el caso del shell PySpark. Este método creará una nueva sesión si ya no existe ninguna sesión; de lo contrario, devuelve una sesión ya existente.
- **master:** Si queremos crear una sesión conectada a un clúster, proporcionaremos el nombre maestro, que puede ser el nombre de instancia del administrador de clúster Spark, YARN o Mesos. Si usamos una opción Apache Spark implementada localmente, podemos usar local[n], donde n es un número entero mayor que cero. La n determinará el número de particiones que se crearán para el RDD y DataFrame. Para una configuración local, n puede ser el número de núcleos de CPU en el sistema. Si lo establecemos en local[*], que es una práctica común, se crearán tantos subprocessos de trabajo como núcleos lógicos haya en el sistema.

Si es necesario crear un nuevo objeto `SparkSession` , podemos usar `newSession` método, que está disponible en el nivel de instancia de un objeto `SparkSession` existente .

A continuación se muestra un ejemplo de código de creación de un nuevo objeto `SparkSession` :

```
importar pyspark  
desde pyspark.sql importar SparkSession  
chispa2 = chispa.nuevaSesion()
```

La salida de la consola para el objeto `spark2` confirma que se trata de una sesión diferente a la de los objetos `SparkSession` creados anteriormente :

```
>>> chispa2  
<objeto pyspark.sql.session.SparkSession en 0x10910df98>
```

El objeto `SparkContext` también se puede crear mediante programación. La forma más sencilla de obtener un objeto `SparkContext` de una instancia de `SparkSession` es mediante el atributo `sparkContext` . También hay una clase `SparkConext` en la biblioteca PySpark que también se puede usar para crear un objeto `SparkContext` directamente, que era un enfoque común antes de la versión 2.0 de Spark.

Nota

Podemos tener múltiples objetos `SparkSession` pero solo un objeto `SparkContext` por JVM.

La clase `SparkSession` ofrece algunos métodos y atributos más útiles que se resumen a continuación:

- `getActiveSession`: este método devuelve una `SparkSession` activa bajo el subproceso actual de Spark.
- `createDataFrame`: este método crea un objeto `DataFrame` a partir de un `RDD`, una lista de objetos o un objeto Pandas `DataFrame`.
- `conf`: este atributo devuelve la interfaz de configuración para una sesión de Spark. • catálogo: este atributo proporciona una interfaz para crear, actualizar o consultar bases de datos, funciones y tablas.

Se puede explorar una lista completa de métodos y atributos utilizando la documentación de PySpark para la clase `SparkSession` en <https://spark.apache.org/docs/latest/api/python/reference/api/>.

Explorando PySpark para operaciones RDD

En la sección Introducción a los RDD, cubrimos algunas de las funciones y operaciones clave de los RDD.

En esta sección, ampliaremos la discusión en el contexto de PySpark con ejemplos de código.

Creación de RDD a partir de una colección de Python y de un archivo externo

Discutimos algunas formas de crear RDD en la sección anterior. En los siguientes ejemplos de código, analizaremos cómo crear RDD desde una colección de Python en memoria y desde un recurso de archivo externo. Estos dos enfoques se describen a continuación:

- Para crear un RDD a partir de una recopilación de datos de Python, tenemos un paralelismo disponible en la instancia de `sparkContext`. Este método distribuye la colección para formar un objeto RDD. El método toma una colección como parámetro.
Un segundo parámetro opcional está disponible con el método de paralelización para establecer el número de particiones que se crearán. De forma predeterminada, este método crea las particiones de acuerdo con la cantidad de núcleos disponibles en la máquina local o la cantidad de núcleos establecida al momento de crear el objeto `SparkSession`.
- Para crear un RDD a partir de un archivo externo, utilizaremos el método `textFile` disponible en la instancia de `sparkContext`. El método `textFile` puede cargar un archivo como RDD desde HDFS o desde un sistema de archivos local (para que esté disponible en todos los nodos del clúster). Para la implementación basada en el sistema local, se puede proporcionar una ruta absoluta y/o relativa. Es posible establecer el número mínimo de particiones que se crearán para el RDD utilizando este método.

A continuación, se muestra un código de ejemplo rápido (`rddcreate.py`) para ilustrar la sintaxis exacta de las declaraciones de PySpark que se utilizarán para la creación de un nuevo RDD:

```
datos = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7,  
8, 4, 4, 8, 2, 7, 8, 9, 6, 9]  
rdd1 = chispa.sparkContext.parallelize(datos)  
imprimir(rdd1.getNumPartitions())  
rdd2 = chispa.sparkContext.textFile('muestra.txt')  
imprimir(rdd2.getNumPartitions())
```

Tenga en cuenta que el archivo `sample.txt` tiene datos de texto aleatorios y su contenido no es relevante para este ejemplo de código.

Operaciones de transformación RDD con PySpark

Hay varias operaciones de transformación integradas disponibles con PySpark. Para ilustrar cómo implementar una operación de transformación como un mapa usando PySpark, tomaremos un archivo de texto como entrada y usaremos la función de mapa disponible con RDD para transformarlo en otro RDD. El código de muestra (rddtransform1.py) se muestra a continuación:

```
rdd1 = chispa.sparkContext.textFile('muestra.txt') rdd2 = rdd1.map( líneas
lambda: líneas.inferior())
rdd3 = rdd1.map( líneas lambda: líneas.superior())

imprimir (rdd2. recopilar ())
imprimir (rdd3. recopilar ())
```

En este código de muestra, aplicamos dos funciones lambda con la operación de mapa para convertir el texto en el RDD a minúsculas y mayúsculas. Al final, usamos el recopilar operación para obtener el contenido de los objetos RDD.

Otra operación de transformación popular es el filtro, que se puede usar para filtrar algunas entradas de datos. A continuación se muestra un código de ejemplo (rddtransform2.py) desarrollado para filtrar todos los números pares de un RDD:

```
datos = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7,
8, 4, 4, 8, 2, 7, 8, 9, 6, 9]
rdd1 = chispa.sparkContext.parallelize(datos)
rdd2 = rdd1.filtro(lambda x: x % 2 != 0 )
imprimir (rdd2. recopilar ())
```

Cuando ejecute este código, proporcionará una salida de consola con 3, 7, 7 y 9 como entradas de colección. A continuación, exploraremos algunos ejemplos de acción con PySpark.

Operaciones de acción RDD con PySpark

Para ilustrar la implementación de operaciones de acción, usaremos un RDD creado a partir de la colección de Python y luego aplicaremos algunas operaciones de acción integradas que vienen con la biblioteca PySpark. El código de ejemplo (rddaction1.py) se muestra a continuación:

```
datos = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7,
8, 4, 4, 8, 2, 7, 8, 9, 6, 9]
rdd1 = chispa.sparkContext.parallelize(datos)

print("Contenido RDD con particiones:" + str(rdd1.glom()).
```

```
recolectar())))
print("Contar por valores: " +str(rdd1.countByValue()))
print("función de reducción: " + str(rdd1.glom().collect()))
print("Suma del contenido de RDD:" +str(rdd1.sum()))
imprimir("arriba: " + str(rdd1.superior(5)))
imprimir("contar: " + str(rdd1.contar()))
imprimir("max: " + str(rdd1.max()))
imprimir("min" + str(rdd1.min()))

tiempo.dormir(60)
```

Algunas de las operaciones de acción utilizadas en este ejemplo de código se explican por sí mismas y son triviales (recuento, máximo, mínimo, recuento y suma). El resto de operaciones de acción (no triviales) se explican a continuación:

- **glom**: esto da como resultado un RDD que se crea fusionando todas las entradas de datos con cada partición en una lista.
- **recopilar**: este método devuelve todos los elementos de un RDD en forma de lista.
- **reduce**: Esta es una función genérica para aplicar al RDD para reducir el número de elementos en ella. En nuestro caso, usamos una función lambda para combinar dos elementos en uno, y así sucesivamente. Esto da como resultado la adición de todos los elementos en el RDD.
- **top(x)**: esta acción devuelve los elementos x superiores de la matriz si los elementos de la matriz están ordenados.

Hemos cubierto cómo crear RDD usando PySpark y cómo implementar operaciones de transformación y acción en un RDD. En la siguiente sección, cubriremos PySpark DataFrame, que es otra estructura de datos popular utilizada principalmente para análisis.

Aprender sobre los marcos de datos de PySpark

PySpark **DataFrame** es una estructura de datos tabulares que consta de filas y columnas, como las tablas que tenemos en una base de datos relacional y como Pandas DataFrame, que presentamos en el Capítulo 6, Consejos y trucos avanzados en Python. En comparación con pandas DataFrames, la diferencia clave es que los objetos PySpark DataFrame se distribuyen en el clúster, lo que significa que los datos se almacenan en diferentes nodos en un clúster. El uso de un DataFrame es principalmente para procesar una gran colección de datos estructurados o no estructurados, que pueden alcanzar los petabytes, de manera distribuida. Al igual que los RDD, los marcos de datos de PySpark son inmutables y se basan en una evaluación perezosa, lo que significa que la evaluación se retrasará hasta que sea necesario realizarla.

Podemos almacenar tipos de datos numéricos y de cadena en un DataFrame. Las columnas en un PySpark DataFrame no pueden estar vacías; deben tener el mismo tipo de datos y deben tener la misma longitud. Las filas en un DataFrame pueden tener datos de diferentes tipos de datos. Se requiere que los nombres de fila en un DataFrame sean únicos.

En las siguientes subsecciones, aprenderemos cómo crear un DataFrame y cubrir algunas operaciones clave en DataFrames usando PySpark.

Creando un objeto DataFrame

Se puede crear un PySpark DataFrame utilizando una de las siguientes fuentes de datos:

- Colecciones de Python como listas, tuplas y diccionarios.
- Archivos (CSV, XML, JSON, Parquet, etc.).
- RDD, utilizando el método `toDF` o el método `createDataFrame` de PySpark.
- Los mensajes de streaming de Apache Kafka se pueden convertir en PySpark DataFrames mediante el método `readStream` del objeto `SparkSession`.
- Las tablas de bases de datos (por ejemplo, Hive y HBase) se pueden consultar mediante comandos SQL tradicionales y la salida se transformará en un PySpark DataFrame.

Comenzaremos creando un DataFrame a partir de una colección de Python, que es el enfoque más simple, pero es más útil con fines ilustrativos. El siguiente fragmento de código de muestra nos muestra cómo crear un PySpark DataFrame a partir de una colección de datos de empleados:

```
datos = [('James','Bylsma','HR','M',40000),
          ('Kamal','Rahim','HR','M',41000),
          ('Robert','Zaine','Finanzas','M',35000),
          ('Sophia','Anne','Richer','Finanzas','F',47000),
          ('Juan','Voluntad','Brown','Ingeniería','F',65000)
         ]
columnas = ["nombre","segundo nombre","apellido",
            "departamento","género","salario"]
df = chispa.createDataFrame (datos = datos, esquema = columnas)
imprimir(df.imprimirEsquema())
imprimir (df. mostrar ())
```

En este ejemplo de código, primero creamos los datos de fila como una lista de empleados y luego creamos un esquema con nombres de columna. Cuando el esquema es solo una lista de nombres de columna, el tipo de datos de cada columna está determinado por los datos y cada columna se marca como anulable de forma predeterminada. Se puede usar una API más avanzada (StructType o StructField) para definir el esquema de DataFrame manualmente, lo que incluye establecer el tipo de datos y marcar una columna como anulable o no anulable. La salida de la consola de este código de muestra se muestra a continuación, que muestra primero el esquema y luego el contenido de DataFrame como una tabla:

```
raiz
|-- nombre: cadena (anulable = verdadero)
|-- segundo nombre: cadena (anulable = verdadero)
|-- apellido: cadena (anulable = verdadero)
|-- departamento: cadena (anulable = verdadero)
|-- género: cadena (anulable = verdadero)
|-- salario: largo (anulable = verdadero)

+-----+-----+-----+-----+
|nombre|segundo nombre|apellido|departamento|género|salario|
+-----+-----+-----+-----+-----+
| Jaime| Bylsma| RRHH| M| 40000| |
| Kamal| Rahim| | RRHH| M| 41000|
| Roberto| Zaine| Finanzas| M| 35000|
| Sofía| Ana| Más rico| Finanzas| F| 47000|
| Juan| Voluntad| Marrón| Ingeniería| F| 65000|
+-----+-----+-----+-----+
```

En el siguiente ejemplo de código, crearemos un DataFrame a partir de un archivo CSV. El archivo CSV tendrá las mismas entradas que usamos en el ejemplo de código anterior. En este código de muestra (dfcreate2.py), también definimos el esquema manualmente mediante el uso de los objetos StructType y StructField :

```
esquemas = StructType([ \
StructField("nombre", StringType(), Verdadero), \
StructField("segundo nombre", StringType(), Verdadero), \
StructField("apellido", StringType(), Verdadero), \
```

```
StructField("departamento", StringType(), True), \  
StructField("género", StringType(), Verdadero), \  
StructField("salario", IntegerType(), True) \  
])  
df = chispa.read.csv('df2.csv', encabezado=Verdadero, esquema=esquemas)  
imprimir(df.imprimirEsquema())  
imprimir (df. mostrar ())
```

El resultado de la consola de este código será el mismo que se muestra en el ejemplo de código anterior.

La importación de archivos JSON, de texto o XML en un DataFrame es compatible con el método de lectura que usa una sintaxis similar. El soporte de otras fuentes de datos, como RDD y bases de datos, se deja para que usted evalúe e implemente como ejercicio.

Trabajar en un marco de datos PySpark

Una vez que hemos creado un DataFrame a partir de algunos datos, independientemente de la fuente de los datos, estamos listos para analizarlo, transformarlo y tomar algunas medidas para obtener resultados significativos.

La mayoría de las operaciones admitidas por PySpark DataFrame son similares a RDD y pandas DataFrames.

Con fines ilustrativos, cargaremos los mismos datos que en el ejemplo de código anterior en un objeto DataFrame y luego realizaremos las siguientes operaciones:

1. Seleccione una o más columnas del objeto DataFrame utilizando el método de selección .
2. Reemplace los valores en una columna usando un diccionario y el método de reemplazo .
Hay más opciones para reemplazar datos en una columna disponible en la biblioteca PySpark.
3. Agregue una nueva columna con valores basados en los datos de una columna existente.

El código de ejemplo completo (dfoperations.py) se muestra a continuación:

```
datos = [('James','Bylsma','HR','M',40000),  
('Kamal','Rahim','HR','M',41000),  
('Robert','Zaine','Finanzas','M',35000),  
('Sophia','Anne','Richer','Finanzas','F',47000),  
('Juan','Voluntad','Brown','Ingeniería','F',65000)  
]
```

298 Escalado horizontal de Python mediante clústeres

```
columnas = ["nombre", "segundo nombre", "apellido",
"departamento", "género", "salario"] df =
chispa.createDataFrame(datos=datos, esquema = columnas)

#mostrar dos columnas
imprimir(df.select([df.nombre, df.salario]).mostrar())

#reemplazo de valores de una
columna myDict = {'F':'Female', 'M':'Male'}
df2 = df.replace(myDict, subset=['gender'])

#agregar un nuevo nivel de pago de columna basado en los valores de una columna
existente
df3 = df2.withColumn(" Nivel de pago", when((df2.salary
< 40000), lit("10")) \ .when((df.salary >= 40000) & (df.salary <= 50000),
encendido("11")) \ .de lo contrario(encendido("12")) \

)
imprimir (df3. mostrar ())
```

El siguiente es el resultado del ejemplo de código anterior:

```
+-----+-----+
|firstname|salary|
+-----+-----+
|    James| 40000|
|   Kamal| 41000|
| Robert| 35000|
| Sophia| 47000|
|     John| 65000|
+-----+-----+
+-----+-----+-----+-----+-----+-----+
|firstname|middlename|lastname| department|gender|salary|Pay Level|
+-----+-----+-----+-----+-----+-----+
|    James|          | Bylsma|        HR| Male| 40000|      11|
|   Kamal|    Rahim|       |        HR| Male| 41000|      11|
| Robert|          | Zaine|    Finance| Male| 35000|      10|
| Sophia|    Anne| Richer|    Finance|Female| 47000|      11|
|     John|    Will| Brown|Engineering|Female| 65000|      12|
+-----+-----+-----+-----+-----+-----+
```

Figura 8.6 – Salida de consola del programa dfoperations.py

La primera tabla muestra el resultado de la operación de selección . La siguiente tabla muestra el resultado de la operación de reemplazo en la columna de género y también una nueva columna, Nivel de pago.

Hay muchas operaciones integradas disponibles para trabajar con PySpark DataFrames, y muchas de ellas son las mismas que discutimos para pandas DataFrames. Los detalles de esas operaciones se pueden explorar utilizando la documentación oficial de Apache Spark para la versión de software que tiene.

Hay una pregunta legítima que cualquiera haría en este punto, que es: ¿Por qué deberíamos usar PySpark DataFrame cuando ya tenemos pandas DataFrame que ofrece los mismos tipos de operaciones? La respuesta es muy simple. PySpark ofrece DataFrames distribuidos, y las operaciones en dichos DataFrames están destinadas a ejecutarse en un grupo de nodos en paralelo. Esto hace que el rendimiento de PySpark DataFrame sea significativamente mejor que el de pandas DataFrame.

Hemos visto hasta ahora que, como programadores, en realidad no tenemos que programar nada sobre cómo delegar RDD y DataFrames distribuidos a diferentes ejecutores en un clúster independiente o distribuido. Nuestro enfoque es solo en el aspecto de programación del procesamiento de datos. SparkSession y SparkContext se encargan automáticamente de la coordinación y la comunicación con un grupo de nodos local o remoto . Esta es la belleza de Apache Spark y PySpark: permitir que los programadores se centren en resolver los problemas reales en lugar de preocuparse por cómo se ejecutarán las cargas de trabajo.

Introducción a PySpark SQL

Spark SQL es uno de los módulos clave de Apache Spark; se utiliza para el procesamiento de datos estructurados y actúa como un motor de consulta SQL distribuido. Como puede imaginar, Spark SQL es altamente escalable, ya que es un motor de procesamiento distribuido. Por lo general, la fuente de datos para Spark SQL es una base de datos, pero las consultas SQL se pueden aplicar a vistas temporales, que se pueden crear a partir de RDD y DataFrames.

Para demostrar el uso de la biblioteca PySpark con Spark SQL, usaremos el mismo DataFrame que en el código de muestra anterior, usando los datos de los empleados para crear un TempView instancia para consultas SQL. En nuestro ejemplo de código, haremos lo siguiente:

1. Crearemos un PySpark DataFrame para los datos de los empleados desde un Python colección como lo hicimos para el ejemplo de código anterior.
2. Crearemos una instancia de TempView desde PySpark DataFrame usando el método `createOrReplaceTempView` .
3. Usando el método `sql` del objeto Spark Session, ejecutaremos las consultas SQL convencionales en la instancia de TempView , como consultar todos los registros de empleados, consultar empleados con salarios superiores a 45,000, consultar el recuento de empleados por tipo de género y usar el grupo por comando SQL para la columna de género .

El ejemplo de código completo (sql1.py) es el siguiente:

```
datos = [('James', ", 'Bylsma', 'HR', 'M', 40000),  
         ('Kamal', 'Rahim', ", 'HR', 'M', 41000),  
         ('Robert', ", 'Zaine', 'Finanzas', 'M', 35000),  
         ('Sophia', 'Anne', 'Richer', 'Finanzas', 'F', 47000),  
         ('Juan', 'Voluntad', 'Brown', 'Ingeniería', 'F', 65000)  
]  
  
columnas = ["nombre", "segundo nombre", "apellido",
```

```
"departamento","género","salario"] df =  
chispa.createDataFrame(datos=datos, esquema = columnas)  
  
df.createOrReplaceTempView("EMP_DATA")  
  
df2 = chispa.sql("SELECCIONAR * DESDE EMP_DATA")  
imprimir(df2.mostrar())  
  
df3 = spark.sql("SELECCIONE nombre, segundo nombre, apellido,  
salario DESDE EMP_DATA DONDE SALARIO > 45000") print (df3.show())  
  
  
df4 = spark.sql(("SELECCIONE sexo, cuente (*) del grupo EMP_DATA por sexo"))  
print(df4.show())
```

La salida de la consola mostrará los resultados de las tres consultas SQL:

```
+-----+-----+-----+-----+-----+  
|nombre|segundo nombre|apellido| departamento|género|salario|  
+-----+-----+-----+-----+-----+  
| Jaime| Bylsma| RRHH| M| 40000| |
| Kamal| Rahim| RRHH| M| 41000|  
| Roberto| Zaine| Finanzas| M| 35000|  
| Sofía| Ana| Más rico| Finanzas| F| 47000|  
| Juan| Voluntad| Marrón| Ingeniería| F| 65000|  
+-----+-----+-----+-----+-----+  
+-----+-----+-----+-----+  
|nombre|segundo nombre|apellido|salario|  
+-----+-----+-----+-----+  
| Sofía| Ana| Más rico| 47000|  
| Juan| Voluntad| Marrón| 65000|  
+-----+-----+-----+  
+-----+-----+  
|género|recuento(1)|  
+-----+-----+  
| F| 2|
```

| M| 3|

+-----+

Spark SQL es un gran tema dentro de Apache Spark. Brindamos solo una introducción a Spark SQL para mostrar el poder de usar comandos SQL sobre las estructuras de datos de Spark sin conocer la fuente de los datos. Esto concluye nuestra discusión sobre el uso de PySpark para el procesamiento y análisis de datos. En la siguiente sección, discutiremos un par de casos de estudio para construir algunas aplicaciones del mundo real.

Estudios de casos de uso de Apache Spark y PySpark

En secciones anteriores, cubrimos los conceptos fundamentales y la arquitectura de Apache Spark y PySpark. En esta sección, analizaremos dos casos de estudio para implementar dos aplicaciones populares e interesantes para Apache Spark.

Estudio de caso 1: calculadora Pi (π) en Apache Spark

Calcularemos Pi (π) utilizando el clúster Apache Spark que se ejecuta en nuestra máquina local. Pi es el área de un círculo cuando su radio es 1. Antes de analizar el algoritmo y el programa controlador para esta aplicación, es importante presentar la configuración de Apache Spark utilizada para este estudio de caso.

Configuración del clúster de Apache Spark

En todos los ejemplos de código anteriores, usamos PySpark instalado localmente en nuestra máquina sin un clúster. Para este estudio de caso, configuraremos un clúster de Apache Spark mediante el uso de varias máquinas virtuales. Hay muchas herramientas de software de virtualización disponibles, como **VirtualBox**, y cualquiera de estas herramientas de software funcionará para crear este tipo de configuración.

Usamos Ubuntu **Multipass** (<https://multipass.run/>) para construir las máquinas virtuales sobre macOS. Multipass funciona tanto en Linux como en Windows. Multipass es un administrador de virtualización liviano y está diseñado específicamente para que los desarrolladores creen máquinas virtuales con un solo comando. Multipass tiene muy pocos comandos, lo que facilita su uso. Si decide utilizar Multipass, le recomendamos que utilice la documentación oficial para la instalación y configuración. En nuestra configuración de máquinas virtuales, tenemos las siguientes máquinas virtuales creadas con Multipass:

Name	State	IPv4	Image
vm1	Running	192.168.64.2	Ubuntu 20.04 LTS
vm2	Running	192.168.64.3	Ubuntu 20.04 LTS
vm3	Running	192.168.64.4	Ubuntu 20.04 LTS

Figura 8.7 – Máquinas virtuales creadas para nuestro clúster Apache Spark

Instalamos Apache Spark 3.1.1 en cada máquina virtual usando apt-get utilidad. Iniciamos Apache Spark como maestro en vm1 y luego iniciamos Apache Spark como trabajador en vm2 y vm3 proporcionando el URI de Spark maestro, que es Spark://192.168.64.2.7077 en nuestro caso. La configuración completa del clúster de Spark se verá como se muestra aquí:

Node Name	Role	Web UI
192.168.64.2	Master (Spark://192.168.64.2:7077)	http://192.168.64.2:8080/
192.168.64.3	Worker	http://192.168.64.3:8081/
192.168.64.4	Worker	http://192.168.64.4:8081/

Figura 8.8 – Detalles de los nodos del clúster de Apache Spark

La interfaz de usuario web para el nodo principal de Spark se verá como se muestra aquí:



The screenshot shows the Apache Spark Web UI interface. At the top, it displays "Spark Master at spark://192.168.64.2:7077". Below this, there is a summary of cluster metrics:

- URL: spark://192.168.64.2:7077
- Alive Workers: 2
- Cores in use: 2 Total, 0 Used
- Memory in use: 2.0 GiB Total, 0.0 B Used
- Resources in use:
- Applications: 0 Running, 0 Completed
- Drivers: 0 Running, 0 Completed
- Status: ALIVE

Below the summary, there is a section titled "Workers (2)" which lists two workers:

Worker Id	Address	State	Cores	Memory	Resources
worker-20210529145544-192.168.64.3-43027	192.168.64.3:43027	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20210529150201-192.168.64.4-34453	192.168.64.4:34453	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	

At the bottom, there are two links: "Running Applications (0)" and "Completed Applications (0)".

Figura 8.9: interfaz de usuario web para el nodo principal en el clúster de Apache Spark

304 Escalado horizontal de Python mediante clústeres

Aquí se proporciona un resumen de la interfaz de usuario web para el nodo maestro:

- La interfaz de usuario web proporciona el nombre del nodo con la URL de Spark. En nuestro caso, usamos la dirección IP como nombre de host, por lo que tenemos una dirección IP en la URL.
- Están los detalles de los nodos trabajadores, de los cuales hay dos en nuestro caso.
Cada nodo trabajador utiliza 1 núcleo de CPU y 1 GB de memoria.
- La interfaz de usuario web también proporciona detalles de las aplicaciones en ejecución y completadas.

La interfaz de usuario web para los nodos trabajadores tendrá el siguiente aspecto:

The screenshot shows the Apache Spark Web UI for a worker node. At the top, it says "Spark Worker at 192.168.64.3:43027". Below that, it lists basic details:

- ID:** worker-20210529145544-192.168.64.3-43027
- Master URL:** spark://192.168.64.2:7077
- Cores:** 1 (0 Used)
- Memory:** 1024.0 MiB (0.0 B Used)
- Resources:**

Below these details is a link: "Back to Master".

There are two main sections for executors:

- Running Executors (0)**
- Finished Executors (13)**

Under each section, there is a table header with columns: ExecutorID, State, Cores, Memory, Resources, Job Details, and Logs.

Figura 8.10: interfaz de usuario web para los nodos trabajadores en el clúster de Apache Spark

Aquí se proporciona un resumen de la interfaz de usuario web para los nodos trabajadores:

- La interfaz de usuario web proporciona los ID de los trabajadores, así como los nombres de los nodos y los puertos donde los trabajadores escuchan las solicitudes.
- La URL del nodo maestro también se proporciona en la interfaz de usuario web.
- Los detalles del núcleo de la CPU y la memoria asignada a los nodos de trabajo son también disponible.
- La interfaz de usuario web proporciona detalles de los trabajos en curso (**Ejecutores en ejecución**) y los trabajos que ya están terminados.

Escribir un programa controlador para el cálculo de Pi

Para calcular Pi, estamos usando un algoritmo de uso común (el algoritmo de **Monte Carlo**) que asume un cuadrado que tiene un área igual a 4 que circunscribe un círculo unitario (círculo con un valor de radio igual a 1). La idea es generar una gran cantidad de números aleatorios en el dominio de un cuadrado con lados que miden 2. Podemos suponer que hay un círculo dentro del cuadrado con el mismo valor de diámetro que la longitud del lado del cuadrado. Esto significa que el círculo se inscribirá dentro del cuadrado. El valor de Pi se estima calculando la relación entre el número de puntos que se encuentran dentro del círculo y el número total de puntos generados.

El código de muestra completo para el programa controlador se muestra a continuación. En este programa, decidimos usar dos particiones ya que tenemos dos trabajadores disponibles. Utilizamos 10.000.000 puntos por cada trabajador. Otra cosa importante a tener en cuenta es que usamos la URL del nodo maestro de Spark como un atributo maestro al crear la sesión de Apache Spark:

```
#casestudy1.py: Calculadora Pi
desde la importación del operador agregar
de importación aleatoria aleatoria

desde pyspark.sql importar SparkSession

chispa = SparkSession.builder.master
("chispa://192.168.64.2:7077") \
.appName("Aplicación de calculadora Pi") \
.getOrCreate()

particiones = 2
n = 10000000 * particiones

función def(_):
    x = aleatorio() * 2 - 1
    y = aleatorio() * 2 - 1
    devuelve 1 si x ** 2 + y
                           ** 2 <= 1 más 0

cuenta = chispa.chispaContexto.parallelizar(rango(1, n + 1),
particiones).mapa(función).reducir(agregar)
print("Pi es aproximadamente %f" % (4.0 * cuenta / n))
```

306 Escalado horizontal de Python mediante clústeres

La salida de la consola es la siguiente:

Pi es aproximadamente 3.141479

La interfaz de usuario web de Spark proporcionará el estado de la aplicación cuando se ejecute e incluso después de que complete su ejecución. En la siguiente captura de pantalla, podemos ver que dos trabajadores fueron contratados para completar el trabajo:

The screenshot shows the Apache Spark web interface. At the top, it displays "Spark Master at spark://192.168.64.2:7077". Below this, it provides system statistics: URL: spark://192.168.64.2:7077, Alive Workers: 2, Cores in use: 2 Total, 2 Used, Memory in use: 2.0 GiB Total, 2.0 GiB Used, Resources in use: Applications: 1 Running, 0 Completed, Drivers: 0 Running, 0 Completed, and Status: ALIVE.

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20210529145544-192.168.64.3-43027	192.168.64.3:43027	ALIVE	1 (1 Used)	1024.0 MiB (1024.0 MiB Used)	
worker-20210529150201-192.168.64.4-34453	192.168.64.4:34453	ALIVE	1 (1 Used)	1024.0 MiB (1024.0 MiB Used)	

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20210529191451-0000	Pi calculator (kill app)	2	1024.0 MiB		2021/05/29 19:14:51	muasif	RUNNING	9 s

Figura 8.11: estado de la calculadora Pi en la interfaz de usuario web de Spark

Podemos hacer clic en el nombre de la aplicación para pasar al siguiente nivel de detalle de la aplicación, como se muestra en la Figura 8.12. Esta captura de pantalla muestra qué trabajadores están involucrados en completar las tareas y qué recursos se están utilizando (si las cosas aún se están ejecutando):

 Application: Pi claculator app

ID: app-20210529191451-0000
 Name: Pi claculator app
 User: muasif
 Cores: Unlimited (2 granted)
 Executor Limit: Unlimited (2 granted)
 Executor Memory: 1024.0 MiB
 Executor Resources:
 Submit Date: 2021/05/29 19:14:51
 State: FINISHED

▼ Executor Summary (2)

ExecutorID	Worker	Cores	Memory	Resources	State	Logs
1	worker-20210529150201-192.168.64.4-34453	1	1024		KILLED	stdout stderr
0	worker-20210529145544-192.168.64.3-43027	1	1024		KILLED	stdout stderr

▼ Removed Executors (2)

ExecutorID	Worker	Cores	Memory	Resources	State	Logs
1	worker-20210529150201-192.168.64.4-34453	1	1024		KILLED	stdout stderr
0	worker-20210529145544-192.168.64.3-43027	1	1024		KILLED	stdout stderr

Figura 8.12 – Detalles a nivel de ejecutor de la aplicación Calculadora Pi

En este estudio de caso, cubrimos cómo podemos configurar un clúster de Apache Spark con fines de prueba y experimentación y cómo podemos crear un programa de controlador en Python utilizando la biblioteca PySpark para conectarnos a Apache Spark y enviar nuestros trabajos para que se procesen en dos nodos de clúster.

En el próximo estudio de caso, construiremos una nube de palabras utilizando la biblioteca PySpark.

Estudio de caso 2 – Nube de palabras usando PySpark

Una **nube de palabras** es una representación visual de la frecuencia de las palabras que aparecen en algunos datos de texto. En pocas palabras, si una palabra específica aparece con más frecuencia en un texto, aparece más grande y en negrita en la nube de palabras. También se conocen como nubes de **etiquetas** o nubes de **texto** y son herramientas muy útiles para identificar qué partes de algunos datos textuales son más importantes. Un caso de uso práctico de esta herramienta es el análisis de contenido en redes sociales, que tiene muchas aplicaciones para marketing, análisis de negocios y seguridad.

308 Escalado horizontal de Python mediante clústeres

Con fines ilustrativos, hemos creado una aplicación de nube de palabras simple que lee un archivo de texto del sistema de archivos local. El archivo de texto se importa a un objeto RDD que luego se procesa para contar la cantidad de veces que se produjo cada palabra. Procesamos más los datos para filtrar las palabras que se repiten menos de dos veces y también filtramos las palabras que tienen una longitud inferior a cuatro letras. Los datos de frecuencia de palabras se envían al objeto de biblioteca de WordCloud . Para mostrar la nube de palabras, usamos la biblioteca matplotlib .

El código de ejemplo completo se muestra a continuación:

```
#casestudy2.py: aplicación de conteo de palabras
importar matplotlib.pyplot como plt
desde pyspark.sql importar SparkSession
de wordcloud importar WordCloud

chispa = SparkSession.builder.master("local[*]")
.appName("aplicación de nube de palabras")
.getOrCreate()

wc_umbral = 1
wl_umbral = 3
textRDD = chispa.sparkContext.textFile('palabranube.txt',3)
flatRDD = textRDD.flatMap(lambda x: x.split(' '))
wcRDD = flatRDD.map(lambda palabra : (palabra, 1)).\
reduceByKey(lambda v1, v2: v1 + v2)

# filtrar palabras con menos ocurrencias que el umbral
filtradoRDD = wcRDD.filter( par lambda: par[1] >= wc_threshold)

filtradoRDD2 = filtradoRDD.filter(par lambda : len(par[0]) > wl_threshold)

word_freq = dict(filtradoRDD2.collect())
# Crear el objeto de nube de palabras
wordcloud = WordCloud(ancho=480, alto=480, margen=0).\
generar_de_frecuencias(word_freq)

# Mostrar la imagen de la nube generada
plt.imshow(nube de palabras, interpolación='bilínea')
plt.eje("apagado")
```

plt.márgenes(x=0, y=0)

`plt.mostrar()`

La salida de este programa se traza como una aplicación de ventana y la salida se verá como se muestra aquí, según el texto de muestra (wordcloud.txt) proporcionado a la aplicación:

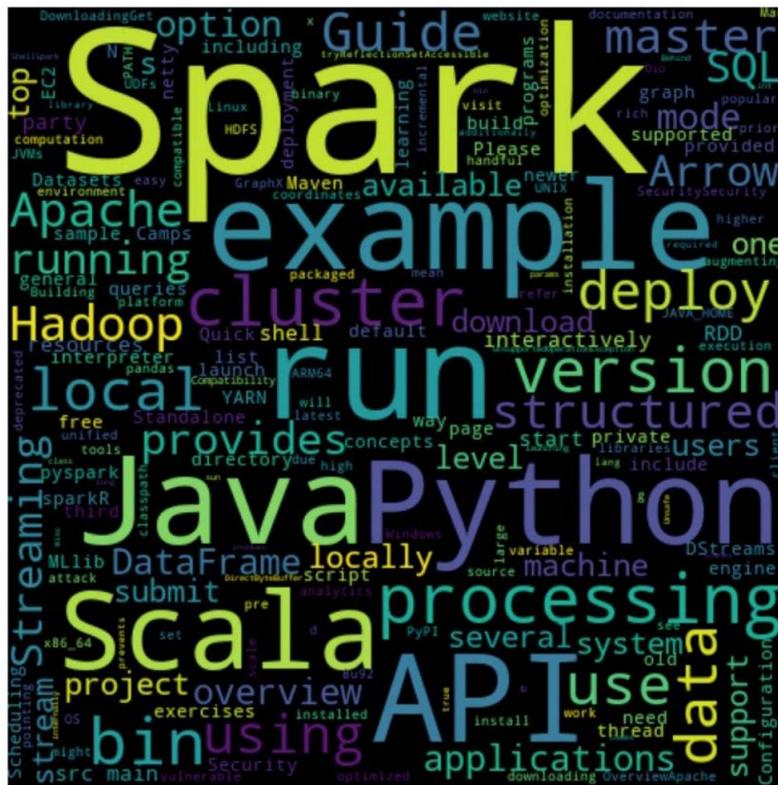


Figura 8.13 – Nube de palabras construida usando PySpark RDD

Tenga en cuenta que no hemos utilizado una muestra muy grande de datos textuales para esta ilustración. En el mundo real, los datos de origen pueden ser extremadamente grandes, lo que justifica el procesamiento con un clúster de Apache Spark.

Estos dos estudios de casos le han proporcionado habilidades para usar Apache Spark para procesamiento de datos a gran escala. Proporcionan una base para aquellos de ustedes que estén interesados en los campos del **procesamiento del lenguaje natural (PLN)**, análisis de texto y análisis sentimental. Estas habilidades son importantes para usted si es un científico de datos y su trabajo diario requiere procesamiento de datos para análisis y creación de algoritmos para NLP.

Resumen

En este capítulo, exploramos cómo ejecutar trabajos intensivos en datos en un grupo de máquinas para lograr un procesamiento paralelo. El procesamiento paralelo es importante para los datos a gran escala, que también se conocen como big data. Comenzamos evaluando las diferentes opciones de conglomerados disponibles para el procesamiento de datos. Brindamos un análisis comparativo de Hadoop MapReduce y Apache Spark, que son las dos principales plataformas competidoras para clústeres. El análisis mostró que Apache Spark tiene más flexibilidad en términos de lenguajes admitidos y sistemas de administración de clústeres, y supera a Hadoop MapReduce para el procesamiento de datos en tiempo real debido a su modelo de procesamiento de datos en memoria.

Una vez que establecimos que Apache Spark es la opción más adecuada para una variedad de aplicaciones de procesamiento de datos, comenzamos a analizar su estructura de datos fundamental, que es el RDD. Discutimos cómo crear RDD a partir de diferentes fuentes de datos e introdujimos dos tipos de operaciones, transformaciones y acciones.

En la parte central de este capítulo, exploramos el uso de PySpark para crear y administrar RDD mediante Python. Esto incluía varios ejemplos de código de operaciones de transformación y acción. También presentamos PySpark DataFrames para el siguiente nivel de procesamiento de datos de forma distribuida. Concluimos el tema presentando PySpark SQL con algunos ejemplos de código.

Finalmente, analizamos dos casos de estudio con Apache Spark y PySpark. Estos estudios de casos incluyeron el cálculo de Pi y la construcción de una nube de palabras a partir de datos de texto. También cubrimos en los estudios de casos cómo podemos configurar una instancia independiente de Apache Spark en una máquina local para fines de prueba.

Este capítulo le proporcionó mucha experiencia en la configuración local de Apache Spark, así como en la configuración de clústeres de Apache Spark mediante la virtualización. En este capítulo se proporcionan muchos ejemplos de código para que mejore sus habilidades prácticas. Esto es importante para cualquier persona que quiera procesar sus problemas de big data utilizando clústeres para lograr eficiencia y escalabilidad.

En el próximo capítulo, exploraremos opciones para aprovechar marcos como Apache Beam y ampliaremos nuestra discusión sobre el uso de nubes públicas para el procesamiento de datos.

Preguntas

1. ¿En qué se diferencia Apache Spark de Hadoop MapReduce?
2. ¿En qué se diferencian las transformaciones de las acciones en Apache Spark?
3. ¿Qué es la evaluación perezosa en Apache Spark?
4. ¿Qué es SparkSession?
5. ¿En qué se diferencia PySpark DataFrame de pandas DataFrame?

Otras lecturas

- Chispa en acción, segunda edición de Jean-Georges Perrin
- Aprendiendo PySpark por Tomasz Drabas, Denny Lee
- Recetas PySpark por Raju Kumar Mishra
- Documentación de Apache Spark para la versión que está utilizando (<https://spark.apache.org/docs/rel#>)
- Documentación Multipass disponible en <https://multipass.run/docs>

respuestas

1. Apache Spark es un motor de procesamiento de datos en memoria, mientras que Hadoop MapReduce tiene que leer y escribir en el sistema de archivos.
2. La transformación se aplica para convertir o traducir datos de un formulario a otro, y los resultados permanecen dentro del clúster. Las acciones son las funciones que se aplican a los datos para obtener los resultados que se devuelven al programa controlador.
3. La evaluación diferida se aplica principalmente a las operaciones de transformación, lo que significa que las operaciones de transformación no se ejecutan hasta que se activa una acción en un objeto de datos.
4. SparkSession es un punto de entrada a la aplicación Spark para conectarse a uno o más administradores de clústeres y trabajar con ejecutores para la ejecución de tareas.
5. PySpark DataFrame se distribuye y está destinado a estar disponible en varios nodos de un clúster de Apache Spark para el procesamiento en paralelo.

9

Pitón Programación para la Nube

La computación en la nube es un término amplio que se utiliza para una amplia variedad de casos de uso. Estos casos de uso incluyen una oferta de plataformas informáticas físicas o virtuales, plataformas de desarrollo de software, plataformas de procesamiento de big data, almacenamiento, funciones de red, servicios de software y muchos más. En este capítulo, exploraremos Python para la computación en la nube desde dos aspectos correlacionados. Primero, investigaremos las opciones de usar Python para crear aplicaciones para tiempos de ejecución en la nube. Luego, ampliaremos nuestra discusión sobre el procesamiento intensivo de datos, que comenzamos en el Capítulo 8, Escalamiento horizontal de Python mediante clústeres, de clústeres a entornos de nube. El enfoque de la discusión en este capítulo se centrará en gran medida en las tres plataformas de nube pública; es decir, **Google Cloud Platform (GCP)**, **Amazon Web Services (AWS)** y **Microsoft Azure**.

Cubriremos los siguientes temas en este capítulo:

- Aprender sobre las opciones en la nube para las aplicaciones de Python
- Creación de servicios web de Python para la implementación en la nube
- Uso de Google Cloud Platform para el procesamiento de datos

314 Programación Python para la Nube

Al final de este capítulo, sabrá cómo desarrollar e implementar aplicaciones en una plataforma en la nube y cómo usar Apache Beam en general y para Google Cloud Platform.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- Necesitará una cuenta de servicio para Google Cloud Platform. Una cuenta gratuita será trabajo bien.
- Necesitará el SDK de Google Cloud instalado en su computadora.
- Necesitará Apache Beam instalado en su computadora.

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter09>.

Comenzaremos nuestra discusión observando las opciones de nube que están disponibles para desarrollar aplicaciones para implementaciones en la nube.

Información sobre las opciones en la nube para las aplicaciones de Python

La computación en la nube es la última frontera para los programadores en estos días. En esta sección, investigaremos cómo podemos desarrollar aplicaciones de Python usando un entorno de desarrollo en la nube o usando un **kit de desarrollo de software (SDK)** específico para la implementación en la nube, y luego cómo podemos ejecutar el código de Python en un entorno de nube. También investigaremos opciones relacionadas con el procesamiento intensivo de datos, como Apache Spark en la nube. Comenzaremos con los entornos de desarrollo basados en la nube.

Introducción a los entornos de desarrollo de Python para la nube

Cuando se trata de configurar un entorno de desarrollo de Python para una de las tres nubes públicas principales, hay dos tipos de modelos disponibles:

- **Entorno de desarrollo integrado (IDE)** nativo de la nube
- IDE instalado localmente con una opción de integración para la nube

Discutiremos estos dos modos a continuación.

IDE nativo de la nube

Hay varios entornos de desarrollo nativos de la nube disponibles en general que no están vinculados específicamente a los tres proveedores de nube pública. Estos incluyen **PythonAnywhere**, **Repl.it**, **Trinket** y **Codeanywhere**. La mayoría de estos entornos en la nube ofrecen una licencia gratuita además de una de pago. Estas plataformas de nube pública ofrecen una combinación de herramientas para entornos de desarrollo, como se explica aquí:

- **AWS:** Esto ofrece un IDE de nube sofisticado en forma de **AWS Cloud9**, al que se puede acceder a través de un navegador web. Este IDE en la nube tiene un amplio conjunto de funciones para desarrolladores y la opción de admitir varios lenguajes de programación, incluido Python. Es importante comprender que AWS Cloud9 se ofrece como una aplicación alojada en una instancia de Amazon EC2 (una máquina virtual). No hay una tarifa directa por usar AWS Cloud9, pero habrá una tarifa por usar la instancia subyacente de Amazon EC2 y el espacio de almacenamiento, que es muy nominal para un uso limitado. La plataforma de AWS también ofrece herramientas para crear y probar el código para objetivos de **integración continua (CI)** y **entrega continua (CD)**.

AWS CodeBuild es otro servicio disponible que compila nuestro código fuente, ejecuta pruebas y crea paquetes de software para su implementación. Es un servidor de compilación similar a Bamboo. **AWS CodeStar** se usa comúnmente con AWS Cloud9 y ofrece una plataforma basada en proyectos para ayudar a desarrollar, construir e implementar software. AWS CodeStar ofrece plantillas de proyecto predefinidas para definir una cadena de herramientas de entrega continua completa hasta que se publique el código.

- **Microsoft Azure:** viene con el IDE de **Visual Studio**, que está disponible en línea (basado en la nube) si forma parte de la plataforma Azure DevOps. El acceso en línea al IDE de Visual Studio se basa en una suscripción paga. Visual Studio IDE es bien conocido por sus características y capacidades ricas para ofrecer un entorno para la colaboración a nivel de equipo. Microsoft Azure ofrece **Azure Pipelines** para compilar, probar e implementar su código en cualquier plataforma, como Azure, AWS y GCP. Azure Pipelines admite muchos lenguajes, como Node.js, Python, Java, PHP, Ruby, C/C++ y .NET, e incluso kits de herramientas de desarrollo móvil.

316 Programación Python para la Nube

- **Google:** Google ofrece **Cloud Code** para escribir, probar e implementar el código que se puede escribir a través de su navegador (como a través de AWS Cloud9) o usando un IDE local de su elección. Cloud Code viene con complementos para los IDE más populares, como IntelliJ IDE, Visual Studio Code y JetBrains PyCharm. Google Cloud Code está disponible de forma gratuita y está dirigido a entornos de tiempo de ejecución de contenedores.
Al igual que AWS CodeBuild y Azure Pipelines, Google ofrece un servicio equivalente que también se denomina **Cloud Build** para la creación, prueba e implementación continuas de software en múltiples entornos, como máquinas virtuales y contenedores. Google también ofrece Google **Colaboratory** o **Google Colab** que ofrece Jupyter Notebooks de forma remota. La opción Google Colab es popular entre los científicos de datos

Google Cloud también ofrece **Tekton** y el servicio **Jenkins** para crear modelos de desarrollo y entrega de CI/CD.

Además de todas estas herramientas y servicios dedicados, estas plataformas en la nube ofrecen entornos de shell instalados localmente y en línea. Estos entornos de shell también son una forma rápida de administrar el código en una capacidad limitada.

A continuación, analizaremos las opciones del IDE local para usar Python para la nube.

IDE local para desarrollo en la nube

El entorno de desarrollo nativo de la nube es una gran herramienta para tener opciones de integración nativas en el resto de su ecosistema de nube. Esto hace que crear instancias de recursos bajo demanda y luego implementarlos sea conveniente y no requiere tokens de autenticación. Pero esto viene con algunas advertencias. En primer lugar, aunque las herramientas son en su mayoría gratuitas, los recursos subyacentes que utilizan no lo son. La segunda advertencia es que la disponibilidad fuera de línea de estas herramientas nativas de la nube no es perfecta. A los desarrolladores les gusta escribir código sin vínculos en línea para poder hacerlo en cualquier lugar, como en un tren o en un parque.

Debido a estas advertencias, a los desarrolladores les gusta usar editores locales o IDE para desarrollar y probar el software antes de usar herramientas adicionales para implementar en una de las plataformas en la nube. Los IDE de Microsoft Azure, como Visual Studio y Visual Studio Code, están disponibles para máquinas locales. AWS y la plataforma de Google ofrecen sus propios SDK (entornos similares a shell) y complementos para integrarse con el IDE de su elección. Exploraremos estos modelos de desarrollo más adelante en este capítulo.

A continuación, analizaremos los entornos de tiempo de ejecución que están disponibles en las nubes públicas.

Presentamos opciones de tiempo de ejecución en la nube para Python

La forma más sencilla de obtener un entorno de tiempo de ejecución de Python es obtener una máquina virtual Linux o un contenedor con Python instalado. Una vez que tengamos una máquina virtual o un contenedor, también podemos instalar la versión de Python de nuestra elección. Para cargas de trabajo con uso intensivo de datos, el clúster de Apache Spark se puede configurar en los nodos de cómputo de la nube. Pero esto requiere que seamos dueños de todas las tareas y el mantenimiento relacionados con la plataforma en caso de que algo salga mal.

Casi todas las plataformas de nube pública ofrecen soluciones más elegantes para simplificar la vida de los desarrolladores y administradores de TI. Estos proveedores de nube ofrecen uno o más entornos de tiempo de ejecución preconstruidos en función de los tipos de aplicaciones. Analizaremos algunos de los entornos de tiempo de ejecución que están disponibles en los tres proveedores de nube pública: Amazon AWS, GCP y Microsoft Azure.

¿Qué es un entorno de ejecución?

Un entorno de tiempo de ejecución es una plataforma de ejecución que ejecuta código Python.

Opciones de tiempo de ejecución que ofrece Amazon AWS

Amazon AWS ofrece las siguientes opciones de tiempo de ejecución:

- **AWS Beanstalk:** esta oferta de **plataforma como servicio (PaaS)** se puede utilizar para implementar aplicaciones web que se han desarrollado con Java, .NET, PHP, Node.js, Python y muchas más. Este servicio también ofrece la opción de utilizar Apache, Nginx, Passenger o IIS como servidor web. Este servicio brinda flexibilidad en la administración de la infraestructura subyacente, que a veces se requiere para implementar aplicaciones complejas.
- **AWS App Runner:** este servicio se puede utilizar para ejecutar microservicios y aplicaciones web en contenedores con una API. Este servicio está completamente administrado, lo que significa que no tiene responsabilidades administrativas ni acceso a la infraestructura subyacente.
- **AWS Lambda:** este es un tiempo de ejecución de cómputo sin servidor que le permite ejecutar su código sin la preocupación de administrar ningún servidor subyacente. Este servidor admite varios idiomas, incluido Python. Aunque el código Lambda se puede ejecutar directamente desde una aplicación, esto es adecuado para los casos en los que debemos ejecutar una determinada pieza de código en caso de que los otros servicios de AWS activen un evento.
- **AWS Batch:** esta opción se utiliza para ejecutar trabajos informáticos en grandes volúmenes en forma de lotes. Esta es una opción de nube de Amazon que es una alternativa a las opciones de clúster de Apache Spark y Hadoop MapReduce.

318 Programación Python para la Nube

- **AWS Kinesis:** este servicio también es para procesamiento de datos, pero para transmisión de datos en tiempo real.

Opciones de tiempo de ejecución que ofrece GCP

Las siguientes son las opciones de tiempo de ejecución que están disponibles en GCP:

- **App Engine:** esta es una opción de PaaS de GCP que se puede usar para desarrollar y alojar aplicaciones web a escala. Las aplicaciones se implementan como contenedores en App Engine, pero la herramienta de implementación empaqueta su código fuente en un contenedor. Esta complejidad está oculta para los desarrolladores.
- **CloudRun:** esta opción se utiliza para alojar cualquier código que se haya construido como un contenedor. Las aplicaciones de contenedor deben tener puntos finales HTTP para implementarse en CloudRun. En comparación con App Engine, el paquete de aplicaciones en un contenedor es responsabilidad del desarrollador.
- **Función en la nube:** esta es una solución de propósito único, sin servidor y basada en eventos para hospedar código Python liviano. El código alojado generalmente se activa al escuchar eventos en otros servicios de GCP o mediante solicitudes HTTP directas. Esto es comparable al servicio AWS Lambda.
- **Dataflow:** Esta es otra opción serverless pero principalmente para procesamiento de datos con latencia mínima. Esto simplifica la vida de un científico de datos al eliminar la complejidad de la plataforma de procesamiento subyacente y ofrecer un modelo de canalización de datos basado en Apache Beam.
- **Dataproc:** Este servicio ofrece una plataforma informática basada en Apache Spark, Apache Flink, Presto y muchas herramientas más. Esta plataforma es adecuada para quienes tienen trabajos de procesamiento de datos con dependencias en un ecosistema Spark o Hadoop. Este servicio requiere que aprovisionemos manualmente los clústeres.

Opciones de tiempo de ejecución que ofrece Microsoft Azure

Microsoft Azure ofrece los siguientes entornos de tiempo de ejecución:

- **Servicio de aplicaciones:** este servicio se utiliza para crear e implementar aplicaciones web a escala. Esta aplicación web puede implementarse como un contenedor o ejecutarse en Windows o Linux.
- **Funciones de Azure:** se trata de un entorno de tiempo de ejecución controlado por eventos sin servidor que se utiliza para ejecutar código en función de un determinado evento o solicitud directa. Esto es comparable a AWS Lambda y GCP CloudRun.
- **Lote:** como su nombre indica, este servicio se utiliza para ejecutar trabajos a escala de la nube que requieren cientos o miles de máquinas virtuales.

- **Azure Databricks:** Microsoft se asoció con Databricks para ofrecer esta plataforma basada en Apache Spark para el procesamiento de datos a gran escala.
- **Azure Data Factory:** esta es una opción sin servidor de Azure que puede usar para procesar datos de transmisión y transformar los datos en resultados significativos.

Como hemos visto, los tres principales proveedores de nube ofrecen una variedad de entornos de ejecución basados en las aplicaciones y cargas de trabajo que están disponibles. Los siguientes casos de uso se pueden implementar en plataformas en la nube:

- Desarrollo de servicios web y aplicaciones web
- Procesamiento de datos usando un tiempo de ejecución en la nube
- Aplicaciones basadas en microservicios (contenedores) usando Python
- Funciones o aplicaciones sin servidor para la nube

Abordaremos los dos primeros casos de uso en las próximas secciones de este capítulo. Los casos de uso restantes se discutirán en los próximos capítulos, ya que requieren una discusión más extensa. En la siguiente sección, comenzaremos a crear un servicio web con Python y exploraremos cómo implementarlo en el entorno de tiempo de ejecución de GCP App Engine.

Creación de servicios web de Python para la implementación en la nube

Crear una aplicación para la implementación en la nube es ligeramente diferente a hacerlo para una implementación local. Hay tres requisitos clave que debemos tener en cuenta al desarrollar e implementar una aplicación en cualquier nube. Estos requisitos son los siguientes:

- **Interfaz web:** para la mayoría de las implementaciones en la nube, las aplicaciones que tienen **una interfaz de usuario (GUI)** o **la interfaz de programación de aplicaciones (API)** son los principales candidatos. Las aplicaciones basadas en la interfaz de línea de comandos no obtendrán su usabilidad desde un entorno de nube a menos que se implementen en una instancia de máquina virtual dedicada, y podemos ejecutarlas en una instancia de VM usando SSH o Telnet. Es por eso que seleccionamos una aplicación basada en interfaz web para nuestra discusión.
- **Configuración del entorno:** todas las plataformas de nube pública admiten varios idiomas, así como diferentes versiones de un solo idioma. Por ejemplo, GCP App Engine es compatible con las versiones 3.7, 3.8 y 3.9 de Python a partir de junio de 2021. A veces, los servicios en la nube también le permiten traer su propia versión para la implementación. Para las aplicaciones web, también es importante establecer un punto de entrada para acceder al código y la configuración a nivel de proyecto. Por lo general, se definen en un solo archivo (un archivo YAML, en el caso de la aplicación GCP App Engine).

• **Gestión de dependencias:** El principal reto en cuanto a la portabilidad de cualquier aplicación es su dependencia de bibliotecas de terceros. Para las aplicaciones de GCP App Engine, documentamos todas las dependencias en un archivo de texto (requirements.txt) manualmente o mediante el comando PIP freeze . También hay otras formas elegantes disponibles para resolver este problema. Una de estas formas es empaquetar todas las bibliotecas de terceros con aplicaciones en un solo archivo para la implementación en la nube, como el archivo de almacenamiento web de Java (archivo .war). Otro enfoque es agrupar todas las dependencias que contienen el código de la aplicación y la plataforma de ejecución de destino en un contenedor e implementar el contenedor directamente en una plataforma de alojamiento de contenedores.

Exploraremos las opciones de implementación basadas en contenedores en el Capítulo 11, Uso de Python para el desarrollo de microservicios.

Hay al menos tres opciones para implementar una aplicación de servicio web de Python en GCP App Engine, que son los siguientes:

- Uso del SDK de Google Cloud a través de la interfaz CLI
- Uso de la consola web de GCP (portal) junto con Cloud Shell (interfaz CLI)
- Usar un IDE de terceros como PyCharm

Discutiremos la primera opción en detalle y proporcionaremos un resumen de nuestra experiencia con las otras dos opciones.

Nota IMPORTANTE

Para implementar una aplicación de Python en AWS y Azure, los pasos del procedimiento son los mismos en principio, pero los detalles son diferentes, según el SDK y el soporte de API disponibles de cada proveedor de la nube.

Uso del SDK de Google Cloud

En esta sección, analizaremos cómo usar Google Cloud SDK (principalmente la interfaz CLI) para crear e implementar una aplicación de muestra. Esta aplicación de muestra se implementará en la plataforma **Google App Engine (GAE)** . GAE es una plataforma PaaS y es más adecuada para implementar aplicaciones web utilizando una amplia variedad de lenguajes de programación, incluido Python.

Para usar el SDK de Google Cloud para la implementación de la aplicación Python, debemos tener los siguientes requisitos previos en nuestra máquina local:

- Instale e inicialice el SDK de Cloud. Una vez instalado, puede acceder a él a través de la interfaz CLI y verificar su versión con el siguiente comando. Ten en cuenta que casi todos los comandos del SDK de Cloud comienzan con gcloud:

versión de gcloud

- Instale los componentes del SDK de Cloud para agregar la extensión de App Engine para Python 3.

Esto se puede hacer usando el siguiente comando:

componentes de gcloud instalar app-engine-python

- La API de GCP CloudBuild debe estar habilitada para el proyecto de nube de GCP.
- La facturación en la nube debe estar habilitada para el proyecto en la nube de GCP, incluso si está utilizando una cuenta de prueba, asociando su cuenta de facturación de GCP con el proyecto.
- Los privilegios de usuario de GCP para configurar una nueva aplicación de App Engine y habilitar la API los servicios deben realizarse a nivel de Propietario.

A continuación, describiremos cómo configurar un proyecto de nube de GCP, crear una aplicación de servicio web de muestra e implementarla en GAE.

Configurar un proyecto en la nube de GCP

El concepto de un proyecto de nube de GCP es el mismo que vemos en la mayoría de los IDE de desarrollo. Un proyecto en la nube de GCP consiste en un conjunto de configuraciones a nivel de proyecto que administran cómo nuestro código interactúa con los servicios de GCP y rastrea los recursos que usa el proyecto. Un proyecto de GCP debe estar asociado con una cuenta de facturación. Este es un requisito previo, en términos de facturación, para realizar un seguimiento de cuántos servicios y recursos de GCP se consumen por proyecto.

A continuación, explicaremos cómo configurar un proyecto usando Cloud SDK:

1. Inicie sesión en Cloud SDK con el siguiente comando. Esto lo llevará al navegador web para que pueda iniciar sesión, en caso de que aún no lo haya hecho:

inicio de gcloud

2. Cree un nuevo proyecto llamado time-wsproj. El nombre del proyecto debe ser corto y usar solo letras y números. Se permite el uso de - para una mejor legibilidad:

proyectos de gcloud crear tiempo-wsproj

322 Programación en Python para la nube

3. Cambie su alcance predeterminado del SDK de Cloud al proyecto recién creado, si aún no lo ha hecho, mediante el siguiente comando:

```
gcloud config establece el tiempo del proyecto-wsproj
```

Esto permitirá que Cloud SDK use este proyecto como un proyecto predeterminado para cualquier comando que insertemos a través de la CLI de Cloud SDK.

4. Cree una instancia de App Engine en un proyecto predeterminado o para cualquier proyecto mediante el atributo de proyecto con uno de los siguientes comandos :

```
aplicación gcloud crear #para proyecto predeterminado
```

```
gcloud app create --project=time-wsproj #para un proyecto específico
```

Tenga en cuenta que este comando reservará los recursos de la nube (principalmente computación y almacenamiento) y le pedirá que seleccione una región y una zona para alojar los recursos. Puede seleccionar la región y la zona más cercana a usted y también más apropiada desde el punto de vista de su audiencia.

5. Habilite el servicio API de Cloud Build para el proyecto actual. Como ya comentamos, el servicio Google Cloud Build se usa para compilar la aplicación antes de que se implemente en un tiempo de ejecución de Google, como App Engine. El servicio API de Cloud Build es más fácil de habilitar a través de la consola web de GCP, ya que solo requiere unos pocos clics. Para habilitarlo usando Cloud SDK, primero, necesitamos saber el nombre exacto del servicio.

Podemos obtener la lista de servicios de GCP disponibles mediante el comando `gcloud services list` .

Este comando le dará una larga lista de servicios de GCP para que pueda buscar un servicio relacionado con Cloud Build. También puede usar el formato, atribuido con cualquier comando, para embellecer la salida del SDK de Cloud. Para que esto sea aún más conveniente, puede usar la utilidad grep de Linux (si está usando Linux o macOS) con este comando para filtrar los resultados y luego habilitar el servicio usando el comando enable :

```
lista de servicios de gcloud --disponible | compilación en la nube grep
```

```
#salida será como: NOMBRE: cloudbuild.googleapis.com
```

```
Comando #Cloud SDK para habilitar este servicio
```

```
Los servicios de gcloud habilitan cloudbuild.googleapis.com
```

6. Para habilitar el servicio API de facturación de la nube para nuestro proyecto, primero debemos asociar una cuenta de facturación con nuestro proyecto. La compatibilidad con las cuentas de facturación en Cloud SDK aún no se ha logrado con **la disponibilidad general (GA)**, según la versión 343.0.0 de Cloud SDK. Se puede adjuntar una cuenta de facturación a un proyecto a través de la consola web de GCP. Pero también hay disponible una versión beta de los comandos de Cloud SDK para que pueda lograr lo mismo. Como primer paso, necesitamos saber el ID de la cuenta de facturación que se utilizará. Las cuentas de facturación asociadas con el usuario que inició sesión se pueden recuperar mediante el comando beta que se presenta aquí:

```
lista de cuentas de facturación de gcloud beta
#output incluirá las siguientes #billingAccounts/
0140E8-51G144-2AB62E
#habilitar la facturación en el proyecto actual mediante el enlace de
proyectos de facturación beta de gcloud time-wsproj --billing account
0140E8-51G144-2AB62E
```

Tenga en cuenta que si está utilizando los comandos beta por primera vez, se le pedirá que instale el componente beta. Deberías continuar e instalarlo. Si ya usa una versión de Cloud SDK con un componente de facturación incluido para GA, puede omitir el uso de la palabra clave beta o usar los comandos apropiados, según la documentación de la versión de Cloud SDK.

7. Habilite el servicio API de facturación de la nube para el proyecto actual siguiendo el mismo pasos que seguimos para habilitar la API de Cloud Build. Primero, debemos encontrar el nombre del servicio API y luego habilitarlo usando el siguiente conjunto de comandos de Cloud SDK:

```
lista de servicios de gcloud --disponible | grep facturación en la nube
#salida será: NOMBRE: cloudbilling.googleapis.com
#comando para habilitar este servicio
Los servicios de gcloud habilitan cloudbilling.googleapis.com
```

Los pasos que debe seguir para configurar un proyecto en la nube son sencillos para un usuario de la nube experimentado y no le llevarán más de unos minutos. Una vez que se ha configurado el proyecto, podemos obtener los detalles de configuración del proyecto ejecutando el siguiente comando:

```
los proyectos de gcloud describen time-wsproj
```

El resultado de este comando proporcionará el estado del ciclo de vida del proyecto, el nombre del proyecto, la ID del proyecto y el número del proyecto. El siguiente es un resultado de ejemplo:

```
tiempo de creación: '2021-06-05T12:03:31.039Z'
estado del ciclo de vida: ACTIVO
```

nombre: tiempo-wsproj**ID del proyecto: tiempo-wsproj****número de proyecto: '539807460484'**

Ahora que el proyecto se ha configurado, podemos comenzar a desarrollar nuestra aplicación web Python. Haremos esto en la siguiente sección.

Construyendo una aplicación de Python

Para las implementaciones en la nube, podemos crear una aplicación de Python con un IDE o un editor de sistema y luego emular el tiempo de ejecución de App Engine localmente con el SDK de la nube y el componente app-engine-python, que hemos instalado como requisito previo. Como ejemplo, construiremos una aplicación basada en un servicio web que nos proporcionará la fecha y la hora a través de una API REST. La aplicación se puede activar a través de un cliente API o mediante un navegador web. No habilitamos ninguna autenticación para mantener la implementación simple.

Para construir la aplicación de Python, configuraremos un entorno virtual de Python utilizando el paquete Python venv . Se utilizará un entorno virtual, creado con el paquete venv , para envolver las bibliotecas y secuencias de comandos del intérprete, el núcleo y de terceros de Python para mantenerlos separados del entorno de Python del sistema y otros entornos virtuales de Python.

Creación y gestión de un entorno virtual en Python utilizando el venv

El paquete ha sido compatible con Python desde v3.3. Hay otras herramientas disponibles para crear entornos virtuales, como virtualenv y pipenv. PyPA recomienda usar venv para crear un entorno virtual, por lo que lo seleccionamos para la mayoría de los ejemplos presentados en este libro.

Como primer paso, crearemos un directorio de proyecto de aplicación web llamado time-wsproj que contiene los siguientes archivos:

- aplicación.yaml
- principal.py
- requisitos.txt

Usamos el mismo nombre para el directorio que usamos para crear el proyecto en la nube solo por conveniencia, pero esto no es un requisito. Veamos estos archivos con más detalle.

archivo YAML

Este archivo contiene la configuración de implementación y tiempo de ejecución de una aplicación de App Engine, como el número de versión del tiempo de ejecución. Para Python 3, el archivo app.yaml debe tener al menos un parámetro de tiempo de ejecución (tiempo de ejecución: python38). Cada servicio de la aplicación web puede tener su propio archivo YAML. En aras de la simplicidad, usaremos solo un archivo YAML. En nuestro caso, este archivo YAML solo contendrá el atributo de tiempo de ejecución. Agregamos algunos atributos más al archivo YAML de muestra con fines ilustrativos:

```
tiempo de ejecución: python38
```

archivo de Python main.py

Seleccionamos la biblioteca Flask para construir nuestra aplicación de muestra. Flask es una biblioteca muy conocida para el desarrollo web, principalmente por las potentes funciones que ofrece, junto con su facilidad de uso. Cubriremos Flask en el próximo capítulo en detalle.

Este módulo Python main.py es el punto de entrada de nuestra aplicación. El código completo de la aplicación se presenta aquí:

```
de matraz importación Matraz
desde fecha y hora fecha de importación, fecha y hora
# Si el 'punto de entrada' no está definido en app.yaml, App Engine buscará #una variable de aplicación.
Este es el caso en nuestro archivo YAML

aplicación = Frasco (__nombre__)
@app.ruta('/')
definitivamente bienvenido():

volver 'Bienvenido Phyton Geek! Use el URI apropiado para la fecha
y tiempo'

@app.ruta('/fecha')
definitivamente hoy():
hoy = fecha.hoy()
devuelve "{fecha:" + hoy.strftime("%B %d, %Y") + '}'"

@app.ruta('/hora')
tiempo definido():
ahora = fechahora.ahora()
devuelve "{hora:" + ahora.strftime("%H:%M:%S") + '}'"

si __nombre__ == '__principal__':
```

326 Programación Python para la Nube

```
# Para pruebas locales
aplicación.ejecutar(host='127.0.0.1', puerto=8080, depuración=True)
```

Este módulo proporciona las siguientes características clave:

- Hay un punto de entrada predeterminado llamado aplicación que se define en este módulo. La aplicación
La variable se utiliza para redirigir las solicitudes que se envían a este módulo.
- Utilizando la anotación de Flask, hemos definido controladores para tres URL:
 - a) La raíz / URL activará una función llamada bienvenida. La función de bienvenida devolverá un mensaje de
saludo como una cadena.
 - b) La URL /fecha activará la función de hoy , que devolverá la fecha de hoy
en formato JSON.
 - c) La URL /hora ejecutará la función de hora , que devolverá la hora actual.
tiempo en formato JSON.
- Al final del módulo, agregamos una función __main__ para iniciar un servidor web local que viene con
Flask para fines de prueba.

Archivo de requisitos

Este archivo contiene una lista de dependencias de proyectos para bibliotecas de terceros. App Engine usará el contenido de este archivo para que las bibliotecas requeridas estén disponibles para nuestra aplicación. En nuestro caso, necesitaremos la biblioteca Flask para construir nuestra aplicación web de muestra. El contenido de este archivo para nuestro proyecto es el siguiente:

```
Frasco == 2.0.1
```

Una vez que hemos creado el directorio del proyecto y hecho estos archivos, debemos crear un entorno virtual dentro o fuera del directorio del proyecto y activarlo usando el comando fuente:

```
python -m venv mienv
fuente mienv/bin/activar
```

Después de activar el entorno virtual, debemos instalar las dependencias necesarias, según el archivo requirements.txt . Usaremos la utilidad pip del mismo directorio donde reside el archivo requirements.txt :

```
pip install -r requisitos.txt
```

Una vez que se hayan instalado la biblioteca Flask y sus dependencias, la estructura de directorios se verá así en nuestro PyCharm IDE:

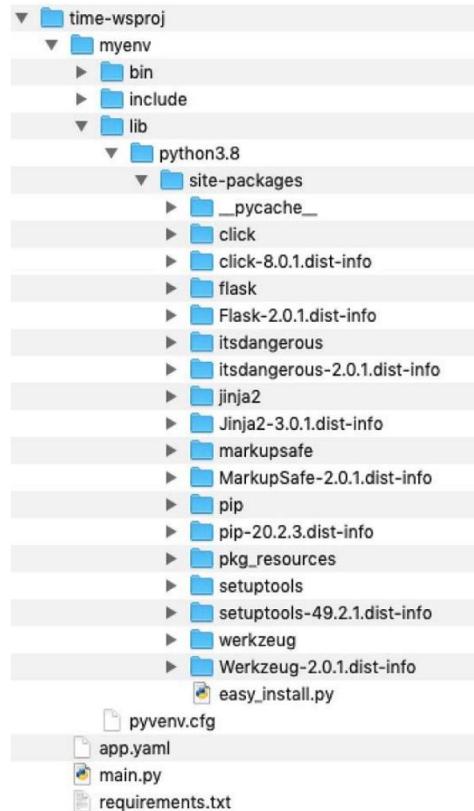


Figura 9.1 – Estructura de directorios para una aplicación web de muestra

Una vez configurado el archivo del proyecto y las dependencias, iniciaremos el servidor web localmente usando el siguiente comando:

```
python principal.py
```

El servidor comenzará con los siguientes mensajes de depuración, lo que deja en claro que esta opción de servidor es solo para fines de prueba y no para entornos de producción:

- * Aplicación de matriz de servicio 'principal' (carga diferida)
- * Medio ambiente: producción
- ADVERTENCIA: Este es un servidor de desarrollo. No lo use en un despliegue de producción.**
- Utilice un servidor WSGI de producción en su lugar.
- * Modo de depuración: activado
- * Ejecutándose en <http://127.0.0.1:8080/> (Presione CTRL+C para salir)
- * Reiniciar con stat

* ¡El depurador está activo!

* PIN del depurador: 668-656-035

Se puede acceder a nuestra aplicación de servicio web utilizando los siguientes URI:

- <http://localhost:8080/>
- <http://localhost:8080/fecha>
- <http://localhost:8080/hora>

La respuesta de los servidores web para estos URI se muestra aquí:

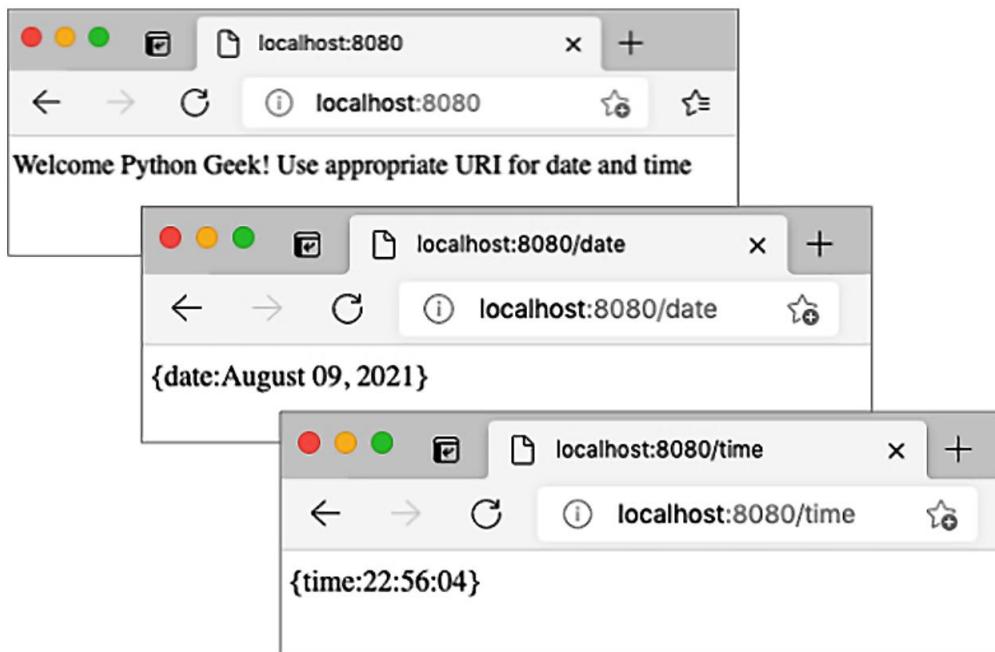


Figura 9.2 – Respuesta en el navegador web de nuestra aplicación de servicio web de muestra

El servidor web se detendrá antes de pasar a la siguiente fase, es decir, implementar esta aplicación en Google App Engine.

Implementación en Google App Engine

Para implementar nuestra aplicación de servicio web en GAE, debemos usar el siguiente comando desde el directorio del proyecto:

implementación de la aplicación gcloud

Cloud SDK leerá el archivo app.yaml , que proporciona información para crear una instancia de App Engine para esta aplicación. Durante la implementación, se crea una imagen de contenedor mediante el servicio Cloud Build; luego, esta imagen de contenedor se carga en el almacenamiento de GCP para su implementación. Una vez que se haya implementado con éxito, podemos acceder al servicio web usando el siguiente comando:

exploración de la aplicación gcloud

Este comando abrirá la aplicación usando el navegador predeterminado en su máquina.

La URL de la aplicación alojada variará según la región y la zona que se seleccionó durante la creación de la aplicación.

Es importante entender que cada vez que ejecutamos el comando de implementación , se creará una nueva versión de nuestra aplicación en App Engine, lo que significa que se consumirán más recursos. Podemos comprobar las versiones que se han instalado para una aplicación web mediante el siguiente comando:

lista de versiones de la aplicación gcloud

Las versiones anteriores de la aplicación todavía pueden estar en un estado de servicio con direcciones URL ligeramente diferentes asignadas a ellas. Las versiones anteriores se pueden detener, iniciar o eliminar con el comando gcloud app versions Cloud SDK y el ID de la versión. Una aplicación se puede detener o iniciar utilizando los comandos detener o iniciar , como se muestra aquí:

Las versiones de la aplicación gcloud **detienen** <id de versión>

Las versiones de la aplicación gcloud **comienzan** <ID de versión>

Las versiones de la aplicación gcloud **eliminan** <id de versión>

El ID de la versión está disponible cuando ejecutamos el comando de lista de versiones de aplicaciones de gcloud . Esto concluye nuestra discusión sobre la creación y la implementación de una aplicación web de Python en Google Cloud. A continuación, resumiremos cómo podemos aprovechar la consola de GCP para implementar la misma aplicación.

Usar la consola web de GCP

La consola de GCP proporciona un portal web fácil de usar para acceder y administrar proyectos de GCP, así como una versión en línea de Google **Cloud Shell**. La consola también ofrece paneles personalizables, visibilidad de los recursos de la nube utilizados por los proyectos, detalles de facturación, registro de actividades y muchas más funciones. Cuando se trata de desarrollar e implementar una aplicación web con la consola de GCP, tenemos algunas características que podemos usar gracias a la interfaz de usuario web, pero la mayoría de los pasos requerirán el uso de Cloud Shell. Este es un SDK de Cloud que está disponible en línea a través de cualquier navegador.

330 Programación Python para la Nube

Cloud Shell es más que Cloud SDK de varias maneras:

- Ofrece acceso a la CLI de gcloud , así como a la CLI de kubectl . se utiliza kubectl para administrar recursos en el motor GCP Kubernetes.
- Con Cloud Shell, podemos desarrollar, depurar, construir e implementar nuestras aplicaciones usando Redactor de la cáscara de la nube.
- Cloud Shell también ofrece un servidor de desarrollo en línea para probar una aplicación antes de implementarla en App Engine.
- Cloud Shell viene con herramientas para cargar y descargar archivos entre Cloud Shell plataforma y su máquina.
- Cloud Shell viene con la capacidad de obtener una vista previa de la aplicación web en el puerto 8080 o en un puerto de su elección.

Los comandos de Cloud Shell que se requieren para configurar un nuevo proyecto, compilar la aplicación e implementar en App Engine son los mismos que analizamos para Cloud SDK. Por eso te dejamos esto para que lo explores siguiendo los mismos pasos que te describimos en el apartado anterior. Ten en cuenta que el proyecto se puede configurar mediante la consola de GCP. La interfaz de Cloud Shell se puede habilitar usando el ícono de Cloud Shell en la barra de menú superior, en el lado derecho. Una vez que se haya habilitado Cloud Shell, aparecerá una interfaz de línea de comandos en la parte inferior de la página web de la consola. Esto se muestra en la siguiente captura de pantalla:

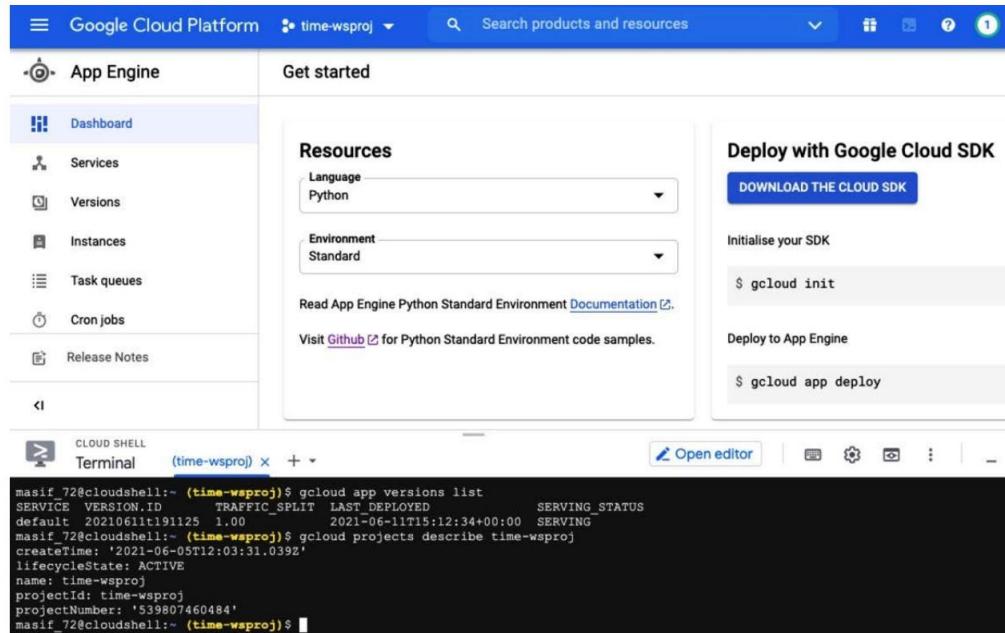


Figura 9.3 – Consola GCP con Cloud Shell

Como mencionamos anteriormente, Cloud Shell viene con una herramienta de edición que se puede iniciar con el botón **Abrir editor**. La siguiente captura de pantalla muestra el archivo de Python abierto dentro de **Cloud Shell Editor**:

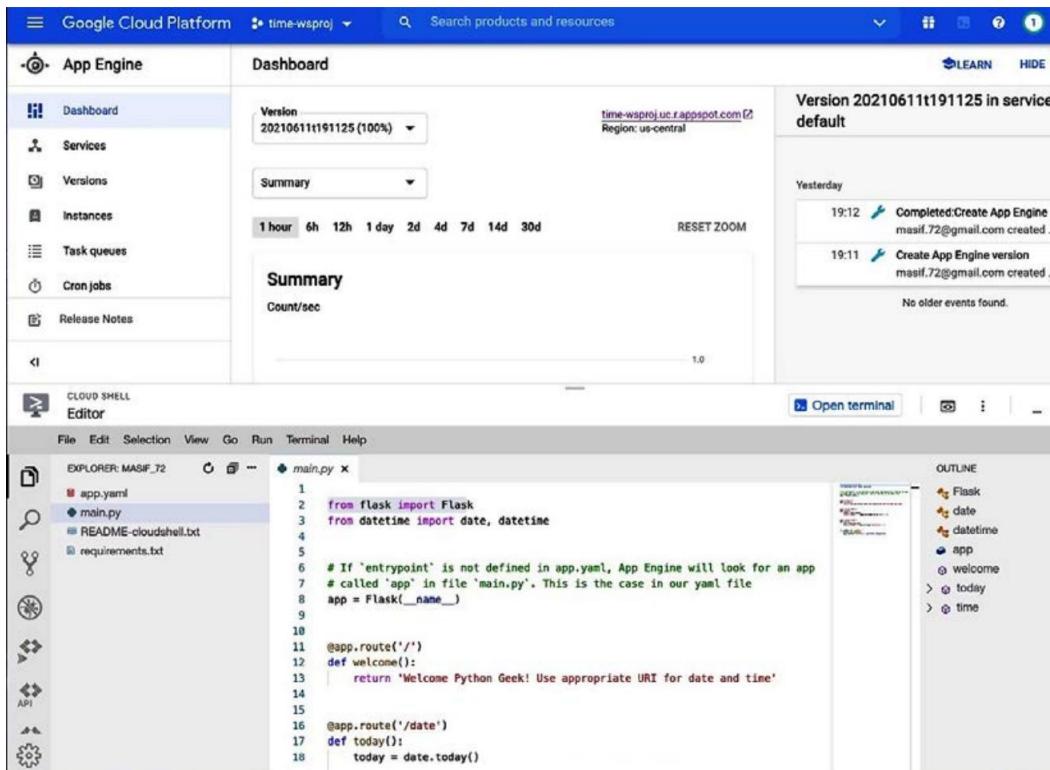


Figura 9.4 – Consola GCP con Cloud Shell Editor

Otra opción cuando se trata de crear e implementar aplicaciones web es usar IDE de terceros con complementos de Google App Engine. Según nuestra experiencia, los complementos que están disponibles para los IDE de uso común, como PyCharm y Eclipse, se crean principalmente para Python 2 y bibliotecas de aplicaciones web heredadas. La integración directa de IDE con GCP requiere más trabajo y evolución. Al momento de escribir, la mejor opción es usar Cloud SDK o Cloud Shell directamente junto con el editor o IDE de su elección para el desarrollo de aplicaciones.

En esta sección, cubrimos el desarrollo de aplicaciones web con Python y su implementación en la plataforma GCP App Engine. Amazon ofrece el servicio AWS Beanstalk para la implementación de aplicaciones web. Los pasos para implementar una aplicación web en AWS Beanstalk son casi los mismos que para GCP App Engine, excepto que AWS Beanstalk no necesita la configuración de proyectos como requisito previo. Por lo tanto, podemos implementar aplicaciones más rápido en AWS Beanstalk.

332 Programación Python para la Nube

Para implementar nuestra aplicación de servicio web en AWS Beanstalk, debemos proporcionar la siguiente información, ya sea mediante la consola de AWS o la CLI de AWS:

- Nombre de la aplicación
- Plataforma (Python versión 3.7 o 3.8, en nuestro caso)
- Versión del código fuente
- Código fuente, junto con un archivo requirements.txt

Recomendamos utilizar la AWS CLI para aplicaciones web que dependen de bibliotecas de terceros. Podemos cargar nuestro código fuente como archivo ZIP o como archivo web (archivo WAR) desde nuestra máquina local o copiarlos desde una ubicación de **Amazon S3** .

Los pasos exactos para implementar una aplicación web en AWS Beanstalk están disponibles en <https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deployment-python-flask.html>. Azure ofrece App Service para crear e implementar aplicaciones web. Puede encontrar los pasos para crear e implementar una aplicación web en Azure en <https://docs.microsoft.com/en-us/azure/app-service/quickstart-python>.

A continuación, exploraremos la creación de programas de controladores para el procesamiento de datos mediante plataformas en la nube.

Uso de Google Cloud Platform para el procesamiento de datos

Google Cloud Platform ofrece Cloud Dataflow como un servicio de procesamiento de datos para aplicaciones de transmisión de datos por lotes y en tiempo real. Este servicio está destinado a científicos de datos y desarrolladores de aplicaciones de análisis para que puedan configurar una **canalización** de procesamiento para el análisis y procesamiento de datos. Cloud Dataflow usa Apache Beam bajo el capó. **Apache Beam** se originó en Google, pero ahora es un proyecto de código abierto bajo Apache. Este proyecto ofrece un modelo de programación para la construcción de procesamiento de datos utilizando tuberías. Tales canalizaciones se pueden crear usando Apache Beam y luego ejecutarlas usando la nube Servicio de flujo de datos.

El servicio Google Cloud Dataflow es similar a Amazon Kinesis, Apache Storm, Apache Spark y Facebook Flux. Antes de discutir cómo usar Google Dataflow con Python, presentaremos Apache Beam y sus conceptos de canalización.

Aprender los fundamentos de Apache Beam

En la era actual, los datos son como una fuente de ingresos para muchas organizaciones. Las aplicaciones, los dispositivos y la interacción humana con los sistemas generan una gran cantidad de datos. Antes de consumir los datos, es importante procesarlos. Los pasos que se definen para el procesamiento de datos generalmente se denominan canalizaciones en la nomenclatura de Apache Beam. En otras palabras, una canalización de datos es una serie de acciones que se realizan en datos sin procesar que se originan en diferentes fuentes y luego mueven esos datos a un destino para el consumo por parte de aplicaciones comerciales o analíticas.

Apache Beam se utiliza para dividir el problema en pequeños paquetes de datos que se pueden procesar en paralelo. Uno de los principales casos de uso de Apache Beam son las aplicaciones de **extracción, transformación y carga (ETL)**. Estos tres pasos de ETL son fundamentales para una tubería cada vez que tenemos que mover los datos de una forma sin procesar a una forma refinada para el consumo de datos.

Los conceptos y componentes básicos de Apache Beam son los siguientes:

- **Canalización:** una canalización es un esquema para transformar los datos de un formulario en el otro como parte del procesamiento de datos.
- **PCollection:** una Pcollection, o colección paralela, es análoga a RDD en Apache Spark. Es un conjunto de datos distribuido que contiene una bolsa de elementos inmutable y desordenada. El tamaño del conjunto de datos puede ser fijo o limitado, similar al procesamiento por lotes, donde sabemos cuántos trabajos procesar en un lote. El tamaño también puede ser flexible o ilimitado en función de la fuente de datos de transmisión y actualización continua.
- **PTransforms:** Son las operaciones que se definen en un pipeline para transformar los datos. Estas operaciones se realizan en objetos PCollection.
- **SDK:** un kit de desarrollo de software específico del idioma que está disponible para Java, Python y Go para crear canalizaciones y enviarlas a un ejecutor para su ejecución.
- **Runner:** Esta es una plataforma de ejecución para pipelines de Apache Beam. El software Runner tiene que implementar un único método llamado run (Pipeline) que es asíncrono por defecto. Algunos corredores disponibles son Apache Flink, Apache Spark y Google Cloud Dataflow.
- **Funciones definidas por el usuario (ParDo/DoFn):** Apache Beam ofrece varios tipos de **funciones definidas por el usuario (UDF)**. La función más utilizada es DoFn, que opera por elemento. La implementación de DoFn provista está envuelta dentro de un objeto ParDo que está diseñado para ejecución paralela.

334 Programación Python para la Nube

Una canalización simple tiene el siguiente aspecto:



Figura 9.5 – Flujo de un pipeline con tres operaciones PTransform

Para diseñar una canalización, normalmente debemos considerar tres elementos:

1. Primero, necesitamos entender la fuente de los datos. ¿Se almacena en un archivo o en una base de datos, o viene como una secuencia? En base a esto, determinaremos qué tipo de operación Read Transform debemos implementar. Como parte de la operación de lectura o una operación separada, también necesitamos comprender el formato o la estructura de los datos.
2. El siguiente paso es definir y diseñar qué hacer con estos datos. Esta es nuestra(s) principal(es) operación(es) de transformación. Podemos tener múltiples operaciones de transformación una tras otra en forma serial o en paralelo sobre los mismos datos. Apache Beam SDK proporciona varias transformaciones preconstruidas que se pueden usar. También nos permite escribir nuestras propias transformaciones usando funciones ParDo/DoFn.
3. Por último, necesitamos saber cuál será el resultado de nuestra canalización y dónde almacenar los resultados de salida. Esto se muestra como una transformación de escritura en el diagrama anterior.

En esta sección, discutimos una estructura de tubería simple para explicar diferentes conceptos relacionados con Apache Beam y las tuberías. En la práctica, la tubería puede ser relativamente compleja. Una canalización puede tener varias fuentes de datos de entrada y varios receptores de salida.

Las operaciones de PTransforms pueden generar varios objetos PCollection, lo que requiere que las operaciones de PTransform se ejecuten en paralelo.

En las próximas secciones, aprenderemos cómo crear una nueva canalización y ejecutar una canalización mediante un ejecutor de Apache Beam o un ejecutor de Cloud Dataflow.

Introducción a las canalizaciones de Apache Beam

En esta sección, discutiremos cómo crear canalizaciones de Apache Beam. Como ya hemos discutido, una canalización es un conjunto de acciones u operaciones que se orquestan para lograr ciertos objetivos de procesamiento de datos. La canalización requiere una fuente de datos de entrada que puede contener datos en memoria, archivos locales o remotos o datos de transmisión.

El pseudocódigo para una canalización típica se verá de la siguiente manera:

[Colección de PC final] = ([Colección de PC de entrada inicial] | [Primera PTransformar] | [Segunda PTransform] | [Tercera PTransformada])

La PCollection inicial se utiliza como entrada para la operación First PTransform. La colección PC de salida de First PTransform se usará como entrada para Second PTransform, y así sucesivamente. El resultado final de PCollection de la última PTransform se capturará como un objeto Final PCollection y se utilizará para exportar los resultados al destino final.

Para ilustrar este concepto, construiremos algunas canalizaciones de ejemplo de diferentes niveles de complejidad. Estos ejemplos están diseñados para mostrar las funciones de diferentes componentes y bibliotecas de Apache que se utilizan para construir y ejecutar una canalización. Al final, crearemos una canalización para una famosa aplicación de conteo de palabras a la que también se hace referencia en la documentación de Apache Beam y GCP Dataflow. Es importante resaltar que debemos instalar la biblioteca Python apache-beam usando la utilidad pip para todos los ejemplos de código en este sección.

Ejemplo 1: creación de una canalización con datos de cadena en memoria

En este ejemplo, crearemos una PCollection de entrada a partir de una colección de cadenas en memoria, aplicaremos un par de operaciones de transformación y luego imprimiremos los resultados en la consola de salida. El siguiente es el código de ejemplo completo:

```
#pipeline1.py: cadenas separadas de una colección PC
import apache_beam as beam
# Create a Pipeline object
with beam.Pipeline() as pipeline:
    topics = (
        pipeline
        | 'Sujetos' >> beam.Create([
            'Inglés Matemáticas Ciencias Francés Artes',
        ])
        | 'Dividir sujetos' >> beam.FlatMap(str.split)
        | beam.Map(print)
```

Para este ejemplo, es importante resaltar algunos puntos:

- Utilizamos | para escribir diferentes operaciones de PTransform en una canalización. Este es un operador sobrecargado que es más como aplicar un PTransform a una PCollection para producir otra PCollection.
- Usamos el operador >> para nombrar cada operación PTransform para registrar y fines de seguimiento. La cadena entre | y >> se utiliza con fines de visualización y registro.

336 Programación Python para la Nube

- Utilizamos tres operaciones de transformación; todos son parte de la biblioteca Apache Beam:
 - a) La primera operación de transformación se utiliza para crear un objeto PCollection, que es una cadena que contiene cinco nombres de sujetos.
 - b) La segunda operación de transformación se utiliza para dividir los datos de cadena en una nueva PCollection utilizando un método de objeto de cadena integrado (división).
 - c) La tercera operación de transformación se usa para imprimir cada entrada en PCollection en la salida de la consola.

La salida de la consola mostrará una lista de nombres de sujetos, con un nombre en una línea.

Ejemplo 2: creación y procesamiento de una canalización con datos de tupla en memoria

En este ejemplo de código, crearemos una PCollection de tuplas. Cada tupla tendrá un nombre de asignatura y una calificación asociada a ella. La operación central de PTransform de esta canalización es separar el tema y su calificación de los datos. El código de ejemplo es el siguiente:

```
#pipeline2.py: Materias separadas con calificación de una colección PC
import apache_beam como viga

def mi_formato(sub, marcas):
    rendimiento '{}\t{}'.format(sub,marcas)

    con beam.Pipeline() como tubería:
        plantas = (
            tubería
            | 'Sujetos' >> haz.Crear([
                ('Inglés', 'A'),
                ('Matemáticas', 'B+'),
                ('Ciencia', 'A-'),
                ('francés', 'A'),
                ('Artes', 'A+'),
            ])
            | 'Dar formato a sujetos con marcas' >> beam.FlatMapTuple(my_
                formato)
            | haz.Mapa(imprimir))
```

En comparación con el primer ejemplo, usamos la operación de transformación FlatMapTuple con una función personalizada para formatear los datos de la tupla. La salida de la consola mostrará el nombre de cada materia, junto con su calificación, en una línea separada.

Ejemplo 3: creación de una canalización con datos de un archivo de texto

En los primeros dos ejemplos, nos enfocamos en construir una canalización simple para analizar datos de cadena de una cadena grande y dividir tuplas de una PColección de tuplas. En la práctica, estamos trabajando en un gran volumen de datos que se cargan desde un archivo o sistema de almacenamiento o que provienen de una fuente de transmisión. En este ejemplo, leeremos datos de un archivo de texto local para construir nuestro objeto PCollection inicial y también enviaremos los resultados finales a un archivo de salida.

El ejemplo de código completo es el siguiente:

```
#pipeline3.py: Leer datos de un archivo y devolver los resultados a otro archivo

import apache_beam como viga
desde apache_beam.io importar WriteToText, ReadFromText

con beam.Pipeline() como tubería:
    líneas = tubería | ReadFromText('muestra1.txt')

temas = (
    líneas
    | 'Asuntos' >> haz.FlatMap(str.split))

    temas | WriteToText(file_path_prefix='asuntos', file_name_suffix='.txt',
                        shard_name_template=")
```

En este ejemplo de código, aplicamos una operación PTransform para leer los datos de texto de un archivo antes de aplicar cualquier PTransform relacionado con el procesamiento de datos. Finalmente, aplicamos una operación PTransform para escribir los datos en un archivo de salida. Usamos dos funciones nuevas en este ejemplo de código llamadas ReadFromText y WriteToText, como se explica aquí:

- **ReadFromText:** esta función forma parte del módulo de E/S de Apache Beam y se utiliza para leer datos de archivos de texto en una colección de cadenas de PC. La ruta del archivo o el patrón del archivo se pueden proporcionar como un argumento de entrada para leer desde una ruta local. También podemos usar gs:// para acceder a cualquier archivo en las ubicaciones de almacenamiento de GCS.

338 Programación Python para la Nube

- `WriteToText`: esta función se utiliza para escribir datos de PCollection en un archivo de texto. Este requiere el argumento `file_path_prefix` como mínimo. También podemos proporcionar el argumento `file_path_suffix` para establecer la extensión del archivo. `fragmento_nombre_` la plantilla se establece en vacío para crear el archivo con un nombre usando los argumentos de prefijo y sufijo. Apache Beam admite una plantilla de nombre de fragmento para definir el nombre de archivo en función de una plantilla.

Cuando esta canalización se ejecuta localmente, creará un archivo denominado `subject.txt` con nombres de sujetos capturados en él, según la operación `PTransform`.

Ejemplo 4: creación de una canalización para un ejecutor de Apache Beam con argumentos

Hasta ahora, hemos aprendido cómo crear una canalización simple, cómo crear un objeto PCollection a partir de un archivo de texto y cómo volver a escribir los resultados en un archivo. Además de estos pasos básicos, debemos realizar algunos pasos más para asegurarnos de que nuestro programa de controlador esté listo para enviar el trabajo a un ejecutor de flujo de datos de GCP o cualquier otro ejecutor de nube. Estos pasos adicionales son los siguientes:

- En el ejemplo anterior, proporcionamos los nombres del archivo de entrada y el patrón del archivo de salida que se configuran en el programa controlador. En la práctica, deberíamos esperar que estos parámetros se proporcionen a través de argumentos de línea de comandos. Usaremos la biblioteca `argparse` para analizar y administrar los argumentos de la línea de comandos.
- Agregaremos argumentos extendidos como establecer un corredor. Este argumento será `runner` se usa para establecer el ejecutor de destino de la canalización mediante `DirectRunner` o un ejecutor de flujo de datos de GCP. Tenga en cuenta que `DirectRunner` es un tiempo de ejecución de canalización para su máquina local. Se asegura de que esas canalizaciones sigan el modelo de Apache Beam lo más fielmente posible.
- También implementaremos y utilizaremos la función `ParDo`, que utilizará nuestra función personalizada para analizar cadenas de datos de texto. Podemos lograr esto usando funciones de cadena, pero se ha agregado aquí para ilustrar cómo usar `ParDo` y `DoFn` con `PTransform`.

Aquí están los pasos:

1. Primero, construiremos el analizador de argumentos y definiremos los argumentos que esperamos de la línea de comando. Estableceremos los valores predeterminados para esos argumentos y configuraremos el texto de ayuda adicional para que se muestre con el interruptor de ayuda en la línea de comando. El atributo dest es importante porque se usa para identificar cualquier argumento que se usará en declaraciones de programación. También definiremos la función ParDo , que se utilizará para ejecutar la canalización. Aquí se presenta un código de ejemplo:

```
#pipeline4.py(partie 1): uso de un argumento para una canalización
```

```
importar re, argparse, apache_beam como haz  
desde apache_beam.io importar WriteToText, ReadFromText  
desde la importación apache_beam.options.pipeline_options  
PipelineOptions
```

```
clase WordParsingDoFn(haz.DoFn):
```

```
def proceso(auto, elemento):  
    volver re.findall(r'[w\']+', elemento, re.UNICODE)
```

```
def ejecutar(argv=Ninguno):  
    analizador = argparse.ArgumentParser()
```

```
    analizador.add_argument(  
        '--aporte',  
        destino='entrada',  
        predeterminado='muestra1.txt',  
        help='Archivo de entrada para procesar.')  
    analizador.add_argument(  
        '--producción',  
        dest='salida',  
        predeterminado='sujetos',  
        help='Archivo de salida para escribir los resultados.')
```

```
    analizador.add_argument(  
        '--extensión',  
        dest='ext',  
        predeterminado='.txt',  
        help='Extensión del archivo de salida a usar.')
```

340 Programación Python para la Nube

```
argumentos_conocidos, argumentos_de_tubería = analizador.parse_conocido_
argumentos(argv)
```

2. Ahora, configuraremos DirectRunner como nuestro tiempo de ejecución de canalización y nombraremos el trabajo que se ejecutará. El código de muestra para este paso es el siguiente:

```
#pipeline4.py(partie 2): bajo el método de ejecución
pipeline_args.extender([
    '--runner=DirectRunner',
    '--job_name=demo-local-job',
])
pipeline_options = PipelineOptions(pipeline_args)
```

3. Finalmente, crearemos una canalización usando el objeto pipeline_options que creado en el paso anterior. La canalización leerá datos de un archivo de texto de entrada, transformará los datos según nuestra función ParDo y luego guardará los resultados como salida:

```
#pipeline4.py(partie 3): bajo el método de ejecución
con beam.Pipeline(options=pipeline_options) como tubería:
línneas = tubería | ReadFromText(args_conocidos.entrada)
sujetos = (
    líneas
    | 'Sujetos' >> rayo.
    ParDo(WordParsingDoFn()).
    con_tipos_de_salida(str))
temas | WriteToText(argumentos_conocidos.salida,
                     argumentos_conocidos.ext)
```

Cuando ejecutamos este programa directamente a través de un IDE usando los valores predeterminados para el argumento o lo iniciamos desde una interfaz de línea de comandos usando el siguiente comando, obtendremos el mismo resultado:

```
python pipeline4.py --input sample1.txt --output myoutput --extension .txt
```

Las palabras del archivo de texto de entrada (sample1.txt) se analizan y se colocan como una palabra en una línea en el archivo de salida.

Apache Beam es un tema amplio, por lo que no es posible cubrir todas sus funciones sin escribir algunos capítulos sobre él. Sin embargo, cubrimos los aspectos básicos al brindar ejemplos de código que nos permitirán comenzar a escribir canalizaciones simples que podemos implementar en GCP Cloud Dataflow. Cubriremos esto en la siguiente sección.

Creación de canalizaciones para Cloud Dataflow

Los ejemplos de código que hemos discutido hasta ahora se enfocan en construir canalizaciones simples y ejecutarlas usando DirectRunner. En esta sección, crearemos un programa controlador para implementar una canalización de procesamiento de datos de recuento de palabras en Google Cloud Dataflow. Este programa controlador es importante para las implementaciones de Cloud Dataflow porque estableceremos todos los parámetros relacionados con la nube dentro del programa. Debido a esto, no será necesario usar Cloud SDK o Cloud Shell para ejecutar comandos adicionales.

La tubería de conteo de palabras será una versión extendida de nuestro ejemplo de tubería4.py .

Los componentes y pasos adicionales necesarios para implementar la canalización de recuento de palabras se resumen aquí:

1. Primero, crearemos un nuevo proyecto de nube de GCP siguiendo pasos similares a los que seguimos para nuestra aplicación de servicio web para la implementación de App Engine. Podemos usar Cloud SDK, Cloud Shell o la consola GCP para esta tarea.
2. Habilitaremos la API de Dataflow Engine para el nuevo proyecto.
3. A continuación, crearemos un depósito de almacenamiento para almacenar los archivos de entrada y salida y para proporcionar directorios temporales y provisionales para Cloud Dataflow. Podemos lograr esto usando la consola de GCP, Cloud Shell o Cloud SDK. Podemos usar el siguiente comando si estamos usando Cloud Shell o Cloud SDK para crear un nuevo depósito:

```
gsutil mb gs://<nombre del depósito>
```

4. Es posible que deba asociar una cuenta de servicio con el depósito recién creado si no está en el mismo proyecto que el trabajo de canalización de flujo de datos y seleccionar la función de administrador del objeto de almacenamiento para el control de acceso.
5. Debemos instalar Apache Beam con las bibliotecas gcp necesarias . Esto puede ser logrado mediante el uso de la utilidad pip , de la siguiente manera:

```
pip install apache-beam[gcp]
```

342 Programación Python para la Nube

6. Debemos crear una clave de autenticación para la cuenta de servicio de GCP utilizada para el proyecto de nube de GCP. Esto no es necesario si vamos a ejecutar el programa del controlador desde una plataforma GCP como Cloud Shell. La clave de la cuenta de servicio debe descargarse en su máquina local. Para que la clave esté disponible para Apache Beam SDK, debemos establecer la ruta del archivo de clave (un archivo JSON) en una variable de entorno llamada GOOGLE_APPLICATION_CREDENTIALS.

Antes de analizar cómo ejecutar una canalización en Cloud Dataflow, analizaremos rápidamente el programa controlador de conteo de palabras de muestra para este ejercicio. En este programa controlador, definiremos argumentos de línea de comandos, muy similares a los que hicimos en el ejemplo de código anterior (pipeline4.py), excepto que haremos lo siguiente:

- En lugar de configurar el entorno GOOGLE_APPLICATION_CREDENTIALS variable a través del sistema operativo, lo configuraremos usando nuestro programa controlador para facilitar la ejecución con fines de prueba.
- Subiremos el archivo sample.txt al almacenamiento de Google, que es el gs// directorio muasif/input en nuestro caso. Usaremos la ruta a este almacenamiento de Google como el valor predeterminado del argumento de entrada .

El código de ejemplo completo es el siguiente:

```
# wordcount.py(part 1): cuenta palabras en un archivo de texto

importar argparse, os, re, apache_beam como haz
desde apache_beam.io importar ReadFromText, WriteToText
desde apache_beam.options.pipeline_options importar PipelineOptions

desde apache_beam.options.pipeline_options importar SetupOptions

def ejecutar(argv=Ninguno, save_main_session=Verdadero):
    os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "alguna carpeta/
        clave.json"
    analizador = argparse.ArgumentParser()
    analizador.add_argument(
        '--aporte',
        destino='entrada',
        predeterminado='gs://muasif/input/muestra.txt',
        help='Archivo de entrada para procesar.')
```

anificador.add_argument(

```
'--salida', dest='salida',
default='gs://muasif/input/
result',
help='Archivo de salida para escribir los resultados.'
unknown_args, pipeline_args = parser.parse_known_args(argv)
```

En el siguiente paso, configuraremos argumentos extendidos para las opciones de canalización para ejecutar nuestra canalización en el tiempo de ejecución de Cloud Dataflow. Estos argumentos son los siguientes:

- Plataforma de tiempo de ejecución (runner) para ejecución de pipeline (DataflowRunner, en este caso)
- ID del proyecto en la nube de GCP
- Región GCP
- Rutas de depósitos de almacenamiento de Google para almacenar archivos temporales, de entrada y de salida
- Nombre del trabajo con fines de seguimiento

Con base en estos argumentos, crearemos un objeto de opciones de canalización que se usará para la ejecución de la canalización. El código de ejemplo para estas tareas es el siguiente:

wordcount.py (parte 2): bajo el método de ejecución

```
pipeline_args.extend([
    '--runner=DataflowRunner',
    '--project=word-count-316612', '--region=us-central1', '--staging_location=gs://muasif/staging', '--temp_location=gs://muasif/temp', '--job_name=my-wordcount-job',
])
pipeline_options = PipelineOptions(pipeline_args)
pipeline_options.view_as(SetupOptions).save_main_session = save_main_session
```

344 Programación Python para la Nube

Finalmente, implementaremos una canalización con las opciones de canalización que ya se han definido y agregaremos nuestras operaciones PTransform. Para este ejemplo de código, agregamos una operación PTransform adicional para crear un par de cada palabra con 1. En la operación PTransform de seguimiento, agrupamos los pares y aplicamos la operación de suma para contar su frecuencia.

Esto nos da el recuento de cada palabra en el archivo de texto de entrada:

```
# wordcount.py (parte 3): bajo el método de ejecución
con beam.Pipeline(options=pipeline_options) como p:

lineas = pag | ReadFromText(args_conocidos.entrada)
# Cuente las ocurrencias de cada palabra.
cuenta = (
    líneas
    | 'Dividir palabras' >> ( beam.FlatMap( lambda
        x: re.findall(r'[A-Za-z]+', x)). with_output_types(str))

    | 'Emparejar con 1' >> haz.Mapa(lambda x: (x, 1))
    | 'Grupo y Suma' >> haz.CombinePerKey(suma))

def format_result(word_count): (palabra, cuenta) =
    word_count return '%s: %s' % (palabra, cuenta)

salida = cuenta | 'Formato' >> beam.Map(format_result) salida | WriteToText(args_conocidos.salida)
```

Establecemos valores predeterminados para cada argumento dentro del programa controlador. Esto significa que podemos ejecutar el programa directamente con el comando `python wordcount.py` o podemos usar el siguiente comando para pasar los argumentos a través de la CLI:

```
python wordcount.py \
--proyecto word-count-316612 \
--región us-central1 \
--input gs://muasif/input/sample.txt \
--salida gs://muasif/salida/resultados \
--runner Corredor de flujo de datos \
--ubicación_temp gs://muasif/temp \
--staging_ubicación gs://muasif/staging
```

El archivo de salida contendrá los resultados, junto con el recuento de cada palabra del archivo. GCP Cloud Dataflow proporciona herramientas adicionales para monitorear el progreso de los trabajos enviados y para comprender la utilización de recursos para realizar el trabajo. La siguiente captura de pantalla de la consola de GCP muestra una lista de trabajos que se enviaron a Cloud Dataflow. La vista de resumen muestra sus estados y algunas métricas clave:

Jobs									CREATE JOB FROM TEMPLATE	CREATE JOB FROM SQL	ENABLE SORTING	REFRESH	LEARN
<input checked="" type="checkbox"/> Running	Filter	Filter jobs											
	Name	Type	End time	Elapsed time	Start time	Status	SDK version	ID				Region	
	my-wordcount-job	Batch	14 Jun 2021, 21:39:43	5 min 14 sec	14 Jun 2021, 21:34:29	Succeeded	2.30.0	2021-06-14_10_34_27-17633507493411316047				us-central1	
	my-wordcount-job	Batch	14 Jun 2021, 21:30:54	5 min 3 sec	14 Jun 2021, 21:25:51	Succeeded	2.30.0	2021-06-14_10_25_49-927632515235435464				us-central1	
	my-wordcount-job	Batch	14 Jun 2021, 13:56:34	4 min 43 sec	14 Jun 2021, 13:51:51	Failed	2.30.0	2021-06-14_02_51_50-7355109158749375121				us-central1	

Figura 9.6: resumen de trabajos de Cloud Dataflow

346 Programación Python para la Nube

Podemos navegar a la vista detallada del trabajo (haciendo clic en cualquier nombre de trabajo), como se muestra en la siguiente captura de pantalla. Esta vista muestra los detalles del trabajo y el entorno en el lado derecho y un resumen del progreso de los diferentes PTransforms que definimos para nuestra canalización. Cuando el trabajo se está ejecutando, el estado de cada operación de PTransform se actualiza en tiempo real, como se muestra aquí:

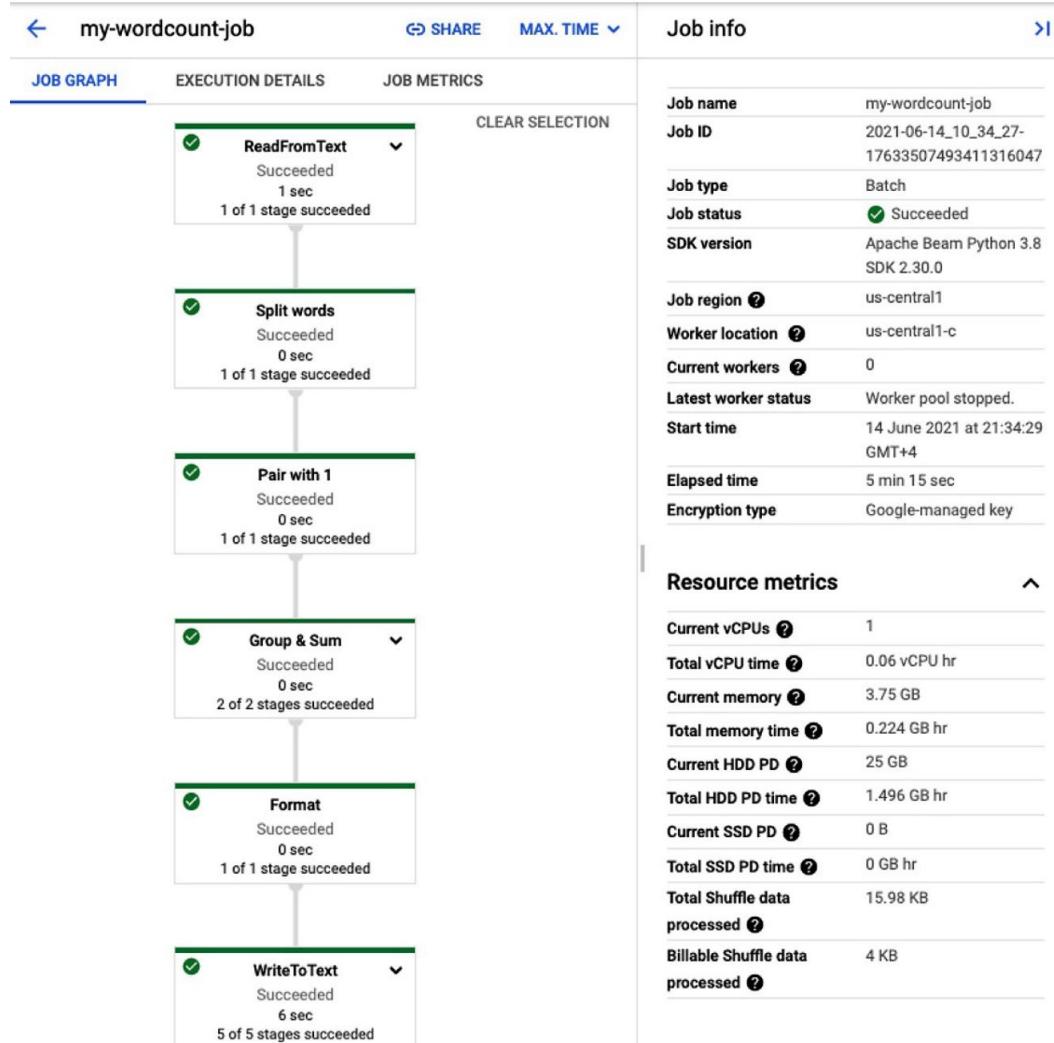


Figura 9.7: Vista detallada del trabajo de Cloud Dataflow con un diagrama de flujo y métricas

Un punto muy importante a tener en cuenta es que las diferentes operaciones de PTransform se nombran de acuerdo con las cadenas que usamos con el operador `>>`. Esto es útil para visualizar las operaciones convenientemente. Esto concluye nuestra discusión sobre la creación e implementación de una canalización para Google Dataflow. En comparación con Apache Spark, Apache Beam proporciona más flexibilidad para el procesamiento de datos paralelo y distribuido. Con la disponibilidad de opciones de procesamiento de datos en la nube, podemos centrarnos completamente en modelar las canalizaciones y dejar el trabajo de ejecutar las canalizaciones a los proveedores de la nube.

Como mencionamos anteriormente, Amazon ofrece un servicio similar (AWS Kinesis) para implementar y ejecutar canalizaciones. AWS Kinesis se centra más en flujos de datos para datos en tiempo real. Al igual que AWS Beanstalk, AWS Kinesis no requiere que configuremos un proyecto como requisito previo.

Las guías de usuario para el procesamiento de datos con AWS Kinesis están disponibles en <https://docs.aws.amazon.com/kinesis/>.

Resumen

En este capítulo, analizamos el rol de Python para desarrollar aplicaciones para la implementación en la nube en general, así como el uso de Apache Beam con Python para implementar canalizaciones de procesamiento de datos en Google Cloud Dataflow. Comenzamos este capítulo comparando tres proveedores principales de nube pública en términos de lo que ofrecen para desarrollar, construir e implementar diferentes tipos de aplicaciones. También comparamos las opciones disponibles de cada proveedor de nube para entornos de tiempo de ejecución. Aprendimos que cada proveedor de la nube ofrece una variedad de motores de ejecución basados en la aplicación o el programa.

Por ejemplo, tenemos motores de tiempo de ejecución separados para aplicaciones web clásicas, aplicaciones basadas en contenedores y funciones sin servidor. Para explorar la eficacia de Python para las aplicaciones web nativas de la nube, creamos una aplicación de muestra y aprendimos a implementar dicha aplicación en Google App Engine mediante el SDK de la nube. En la última sección, ampliamos nuestra discusión sobre el proceso de datos, que comenzamos en el capítulo anterior.

Presentamos un nuevo enfoque de modelado para el procesamiento de datos (canalizaciones) utilizando Apache Beam. Una vez que aprendimos a crear canalizaciones con algunos ejemplos de código, ampliamos nuestra discusión sobre cómo crear canalizaciones para la implementación de Cloud Dataflow.

Este capítulo proporcionó un análisis comparativo de las ofertas de servicios de nube pública. A esto le siguió el conocimiento práctico de la creación de aplicaciones web y aplicaciones de procesamiento de datos para la nube. Los ejemplos de código incluidos en este capítulo le permitirán comenzar a crear proyectos en la nube y escribir código para Apache Beam. Este conocimiento es importante para cualquier persona que quiera resolver sus problemas de big data utilizando servicios de procesamiento de datos basados en la nube.

En el próximo capítulo, exploraremos el poder de Python para desarrollar aplicaciones web usando los frameworks Flask y Django.

Preguntas

1. ¿En qué se diferencia AWS Beanstalk de AWS App Runner?
2. ¿Qué es el servicio GCP Cloud Function?
3. ¿Qué servicios de GCP están disponibles para el procesamiento de datos?
4. ¿Qué es una canalización de Apache Beam?
5. ¿Cuál es el papel de PCollection en una canalización de procesamiento de datos?

Otras lecturas

- Desarrollo Web Flask, por Miguel Grinberg.
- Guía avanzada para la programación de Python 3, por John Hunt. • Apache Beam: una guía completa, por Gerardus Blokdyk. • Google Cloud Platform para desarrolladores, por Ted Hunter, Steven Porter. • La documentación de Google Cloud Dataflow está disponible en <https://cloud.google.com/dataflow/docs>.
- La documentación de AWS Elastic Beanstalk está disponible en <https://docs.aws.amazon.com/elastic-beanstalk>.
- La documentación de Azure App Service está disponible en <https://docs.microsoft.com/en-us/azure/app-service/>.
- La documentación de AWS Kinesis está disponible en <https://docs.aws.amazon.com/kinesis/>.

respuestas

1. AWS Beanstalk es una oferta de PaaS de propósito general para implementar aplicaciones web, mientras que AWS App Runner es un servicio totalmente administrado para implementar aplicaciones web basadas en contenedores.
2. GCP Cloud Function es un servicio basado en eventos sin servidor para ejecutar un programa. El evento especificado se puede activar desde otro servicio de GCP o a través de una solicitud HTTP.

3. Cloud Dataflow y Cloud Dataproc son dos servicios populares para el procesamiento de datos ofrecidos por GCP.
4. Una canalización de Apache Beam es un conjunto de acciones que se han definido para cargar los datos, transformar los datos de un formulario a otro y escribir los datos en un destino.
5. PCollection es como un RDD en Apache Spark que contiene elementos de datos. En el procesamiento de datos de canalización, una operación PTransform típica toma uno o más objetos PCollection como entrada y produce los resultados como uno o más objetos PCollection.

Sección 4:

Usando Python para Web, nube y Casos de uso de red

No terminaremos nuestro viaje hasta que apliquemos lo que hemos aprendido hasta ahora para crear soluciones modernas, especialmente para la nube. Esta es una parte central de nuestro viaje, donde desafiamos nuestro progreso de aprendizaje resolviendo problemas del mundo real. Primero, analizamos cómo crear aplicaciones web y API REST utilizando marcos de Python como Flask. A continuación, profundizamos en la implementación de aplicaciones sin servidor para la nube utilizando la arquitectura de microservicios y las funciones sin servidor. Cubrimos las opciones de implementación local y en la nube para nuestros ejercicios de código y estudios de casos. Más adelante, exploramos cómo usar Python para crear modelos de aprendizaje automático e implementarlos en la nube. Concluimos nuestro viaje investigando el papel de Python para la automatización de redes con casos de uso relacionados con la administración de redes del mundo real.

Esta sección contiene los siguientes capítulos:

- Capítulo 10, Uso de Python para desarrollo web y API REST
- Capítulo 11, Uso de Python para el desarrollo de microservicios
- Capítulo 12, Creación de funciones sin servidor mediante Python
- Capítulo 13, Python y el aprendizaje automático
- Capítulo 14, Uso de Python para la automatización de redes

10

Usando Python para Desarrollo web y API REST

Una aplicación web es un tipo de aplicación alojada y ejecutada por un **servidor web** dentro de una intranet o en Internet y a la que se accede a través de un navegador web en un dispositivo cliente. El uso de navegadores web como interfaz de cliente facilita que los usuarios accedan a las aplicaciones desde cualquier lugar sin instalar ningún software adicional en una máquina local. Esta facilidad de acceso ha contribuido al éxito y la popularidad de las aplicaciones web durante más de dos décadas. El uso de aplicaciones web varía, desde ofrecer contenido estático y dinámico como Wikipedia y periódicos, comercio electrónico, juegos en línea, redes sociales, capacitación, contenido multimedia, encuestas y blogs, hasta aplicaciones complejas **de planificación de recursos empresariales (ERP)**.

354 Uso de Python para desarrollo web y API REST

Las aplicaciones web son de varios niveles por naturaleza, normalmente aplicaciones de tres niveles. Los tres niveles son una interfaz de usuario, lógica empresarial y acceso a la base de datos. Por lo tanto, desarrollar aplicaciones web implica interactuar con servidores web para una interfaz de usuario, un **servidor de aplicaciones** para la lógica comercial y sistemas de bases de datos para datos persistentes. En la era de las aplicaciones móviles, la interfaz de usuario puede ser una aplicación móvil que requiera acceso al nivel de lógica empresarial a través de una API REST. La disponibilidad de una API REST o cualquier tipo de interfaz de servicios web se ha convertido en un requisito fundamental para las aplicaciones web. Este capítulo analiza cómo usar Python para crear aplicaciones web de varios niveles. Hay varios marcos disponibles en Python para desarrollar aplicaciones web, pero seleccionamos **Flask** para nuestra discusión en este capítulo debido a que es rico en funciones pero liviano. Las aplicaciones web también se denominan aplicaciones web para diferenciarlas de las aplicaciones móviles destinadas a dispositivos pequeños.

Cubriremos los siguientes temas en este capítulo:

- Requisitos de aprendizaje para el desarrollo web
- Presentamos el marco Flask
- Interactuar con bases de datos usando Python
- Construyendo una API REST usando Python
- Caso de estudio: Construcción de una aplicación web utilizando la API REST

Al final de este capítulo, podrá utilizar el marco Flask para desarrollar aplicaciones web, interactuar con bases de datos y crear una API REST o servicios web.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- Biblioteca Python Flask 2.x con sus extensiones instaladas sobre Python 3.7 o posterior.

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter10>.

Comenzaremos nuestra discusión con los requisitos clave para desarrollar aplicaciones web y una API REST.

Requisitos de aprendizaje para el desarrollo web.

El desarrollo de una aplicación web incluye la creación de una interfaz de usuario, el enrutamiento de las solicitudes de los usuarios o las acciones de los usuarios a los extremos de la aplicación, la traducción de los datos de entrada del usuario, la escritura de la lógica comercial para las solicitudes de los usuarios, la interacción con la capa de datos para leer o escribir datos y la entrega de los resultados a los usuarios. Todos estos componentes de desarrollo pueden requerir diferentes plataformas y, a veces, incluso utilizar diferentes lenguajes de programación para su implementación.

En esta sección, comprenderemos los componentes y herramientas necesarios para el desarrollo web, comenzando con los marcos de aplicaciones web o marcos web.

Marcos web

Desarrollar aplicaciones web desde cero requiere mucho tiempo y es tedioso. Para que sea conveniente para los desarrolladores web, los marcos de aplicaciones web se introdujeron en los primeros días del desarrollo web. Los marcos web proporcionan un conjunto de bibliotecas, estructuras de directorios, componentes reutilizables y herramientas de implementación. Los marcos web suelen seguir una arquitectura que permite a los desarrolladores crear aplicaciones complejas en menos tiempo y de forma optimizada.

Hay varios marcos web disponibles para Python: **Flask** y **Django** son los más populares. Ambos marcos son gratuitos y de código abierto. Flask es un marco liviano y viene con funcionalidades estándar que se requieren para crear una aplicación web, pero también permite que se usen bibliotecas o extensiones adicionales según sea necesario. Por otro lado, Django es un marco de trabajo de pila completa que viene con todas las funciones listas para usar sin necesidad de bibliotecas adicionales. Ambos enfoques tienen ventajas y desventajas, pero al final, podemos desarrollar cualquier aplicación web con cualquiera de estos marcos.

Flask se considera una mejor opción si desea tener un control total de su aplicación con la opción de usar bibliotecas externas según corresponda. Flask también es una buena opción cuando los requisitos del proyecto cambian con mucha frecuencia. Django es adecuado si desea tener todas las herramientas y bibliotecas disponibles listas para usar y desea concentrarse solo en implementar la lógica comercial. Django es una buena opción para proyectos a gran escala, pero puede ser excesivo para proyectos simples. La curva de aprendizaje de Django es empinada y requiere experiencia previa en desarrollo web. Si está mirando el desarrollo web con Python por primera vez, Flask es el camino a seguir. Una vez que aprende Flask, es fácil adoptar el marco Django para el siguiente nivel de proyectos web.

356 Uso de Python para desarrollo web y API REST

Al crear aplicaciones web o cualquier aplicación de interfaz de usuario, a menudo nos encontramos con el término patrón de diseño **Model View Controller (MVC)**. Este es un patrón de diseño arquitectónico que divide una aplicación en tres capas:

- **Modelo:** la capa del modelo representa los datos que normalmente se almacenan en una base de datos.
- **Ver:** esta capa es la interfaz de usuario con la que interactúan los usuarios.
- **Controlador:** la capa del controlador está diseñada para proporcionar una lógica para manejar las interacciones del usuario con la aplicación a través de la interfaz de usuario. Por ejemplo, un usuario puede querer crear un nuevo objeto o actualizar un objeto existente. La lógica de qué interfaz de usuario (vista) presentar al usuario para la solicitud de creación o actualización con o sin un modelo (datos) se implementa en la capa del controlador.

Flask no brinda soporte directo para los patrones de diseño de MVC, pero se puede implementar a través de la programación. Django proporciona una implementación lo suficientemente cercana para MVC, pero no completamente. El controlador en el marco de trabajo de Django es administrado por el mismo Django y no está disponible para escribir nuestro propio código en él. Django y muchos otros frameworks web para Python siguen el patrón de diseño **Model View Template (MVT)**, que es como MVC excepto por la capa de plantilla. La capa de plantilla en el MVT proporciona plantillas con formato especial para producir la interfaz de usuario esperada con la capacidad de insertar contenido dinámico dentro de HTML.

Interfaz de usuario

Una **interfaz** de usuario es la capa de presentación de la aplicación y, a veces, se incluye como parte de los marcos web. Pero lo estamos discutiendo por separado aquí para resaltar las tecnologías clave y las opciones disponibles para esta capa. En primer lugar, el usuario interactúa a través de un navegador en **lenguaje de marcado de hipertexto (HTML)** y **hojas de estilo en cascada (CSS)**. Podemos construir nuestras interfaces escribiendo HTML y CSS directamente, pero es tedioso y no factible para entregar contenido dinámico de manera oportuna. Hay algunas tecnologías disponibles para facilitarnos la vida al crear una interfaz de usuario:

- **Marco de la interfaz de usuario:** se trata principalmente de bibliotecas HTML y CSS que proporcionan diferentes clases (estilos) para crear la interfaz de usuario. Todavía necesitamos escribir o generar partes HTML centrales de la interfaz de usuario, pero sin preocuparnos por cómo embellecer nuestras páginas web. Un ejemplo popular de un marco de interfaz de usuario es **bootstrap**, que se basa en CSS. Fue presentado por Twitter para uso interno, pero luego se hizo de código abierto para que cualquiera lo use. **ReactJS** es otra opción popular, pero es más una biblioteca que un marco y fue presentado por Facebook.

- **Motor de plantillas:** un motor de plantillas es otro mecanismo popular para producir contenido web de forma dinámica. Una plantilla es más como una definición de la salida deseada que contiene datos estáticos, así como marcadores de posición para contenidos dinámicos. Los marcadores de posición son cadenas tokenizadas que se reemplazan por valores en tiempo de ejecución. La salida puede tener cualquier formato, como HTML, XML, JSON o PDF. **Jinja2** es uno de los motores de plantillas más populares utilizados con Python y también se incluye con el marco Flask. Django viene con su propio motor de plantillas.
- **Secuencias de comandos del lado del cliente:** una secuencia de comandos del lado del cliente es un programa que se descarga del servidor web y se ejecuta mediante el navegador web del cliente. JavaScript es el lenguaje de secuencias de comandos del lado del cliente más popular. Hay muchas bibliotecas de JavaScript disponibles para facilitar el desarrollo web.

Podemos utilizar más de una tecnología para desarrollar interfaces web. En un proyecto web típico, los tres se utilizan en diferentes capacidades.

servidor web/servidor de aplicaciones

Un servidor web es un software que escucha las solicitudes de los clientes a través de HTTP y entrega contenido (como páginas web, scripts, imágenes) según el tipo de solicitud. El trabajo fundamental del servidor web es servir solo recursos estáticos y no es capaz de ejecutar código.

El servidor de aplicaciones es más específico de un lenguaje de programación. El trabajo principal de un servidor de aplicaciones es brindar acceso a la implementación de una lógica comercial que se escribe utilizando un lenguaje de programación como Python. Para muchos entornos de producción, el servidor web y el servidor de aplicaciones se agrupan como un solo software para facilitar la implementación. Flask viene con su propio servidor web incorporado, llamado **Werkzeug**, para la fase de desarrollo, pero no se recomienda su uso en producción. Para la producción, debemos usar otras opciones como los motores de tiempo de ejecución **Gunicorn**, **uWSGI** y **GCP**.

Base de datos

Este no es un componente obligatorio, pero es casi esencial para cualquier aplicación web interactiva. Python ofrece varias bibliotecas para acceder a sistemas de bases de datos de uso común, como **MySQL**, **MariaDB**, **PostgreSQL** y **Oracle**. Python también viene equipado con un servidor de base de datos liviano, **SQLite**.

358 Uso de Python para desarrollo web y API REST

Seguridad

La seguridad es fundamental para las aplicaciones web, principalmente porque el público objetivo suele ser los usuarios de Internet, y la privacidad de los datos es el requisito más importante en dichos entornos. **Capa de sockets seguros (SSL)** y la **seguridad de la capa de transporte** introducida más recientemente (**TLS**) son los estándares de seguridad mínimos aceptables para asegurar la transmisión de datos entre los clientes y el servidor. Los requisitos de seguridad a nivel de transporte generalmente se manejan en el servidor web o, a veces, en el nivel del servidor proxy. La seguridad a nivel de usuario es el siguiente requisito fundamental, con requisitos mínimos de nombre de usuario y contraseña. La seguridad del usuario es seguridad a nivel de aplicación y los desarrolladores son los principales responsables de diseñarla e implementarla.

API

La capa de lógica empresarial en las aplicaciones web puede ser consumida por clientes adicionales. Por ejemplo, una aplicación móvil puede usar la misma lógica comercial para un conjunto limitado o igual de características. Para las aplicaciones **Business to Business (B2B)**, una aplicación remota puede enviar solicitudes directamente a la capa de lógica empresarial. Todo esto es posible si exponemos interfaces estándar como una API REST para nuestra capa de lógica empresarial. En la era actual, acceder a la capa de lógica empresarial a través de una API es una buena práctica para preparar la API desde el primer día.

Documentación

La documentación es tan importante como escribir código de programación. Esto es especialmente cierto para las API. Cuando decimos que tenemos una API para nuestra aplicación, la primera pregunta de los consumidores de API es si puede compartir la documentación de la API con nosotros. La mejor manera de tener documentación de API es usar herramientas integradas que vienen o posiblemente se integren con nuestro marco web. **Swagger** es una herramienta popular que se utiliza para generar documentación automáticamente a partir de los comentarios que se agregan en el momento de la codificación.

Ahora que hemos discutido los requisitos clave del desarrollo web, profundizaremos en cómo desarrollar una aplicación web usando Flask en la siguiente sección.

Presentamos el marco Flask

Flask es un marco de desarrollo micro web para Python. El término micro indica que el núcleo de Flask es liviano, pero con la flexibilidad de ser extensible. Un ejemplo simple es interactuar con un sistema de base de datos. Django viene con las bibliotecas necesarias para interactuar con las bases de datos más comunes. Por otro lado, Flask permite el uso de una extensión según el tipo de base de datos o según el enfoque de integración para lograr el mismo objetivo. Otra filosofía de Flask es usar la convención sobre la configuración, lo que significa que si seguimos las convenciones estándar del desarrollo web, tenemos que hacer menos configuración. Esto hace que Flask sea la mejor opción para que los principiantes aprendan desarrollo web con Python. Seleccionamos Flask para nuestro desarrollo web, no solo por su facilidad de uso, sino también porque nos permite introducir diferentes conceptos en un enfoque gradual.

En esta sección, aprenderemos los siguientes aspectos de las aplicaciones web que usan Flask:

- Creación de una aplicación web básica con enrutamiento
- Manejo de solicitudes con diferentes tipos de métodos HTTP
- Representación de contenidos estáticos y dinámicos utilizando Jinja2
- Extracción de argumentos de una solicitud HTTP
- Interactuar con sistemas de bases de datos
- Manejo de errores y excepciones

Antes de comenzar a trabajar con ejemplos de código que se utilizarán en las siguientes secciones, se debe instalar Flask 2.x en nuestro entorno virtual. Comenzaremos con una aplicación web básica usando Flask.

Creación de una aplicación básica con enrutamiento

Ya usamos Flask para crear una aplicación de muestra para la implementación de GCP App Engine en el último capítulo, Programación de Python para la nube. Actualizaremos nuestro conocimiento sobre el uso de Flask para desarrollar una aplicación web simple. Comenzaremos con un ejemplo de código para comprender cómo se construye una aplicación web y cómo funciona el enrutamiento en ella.

El ejemplo de código completo es el siguiente:

```
#app1.py: enrutamiento en una aplicación Flask
```

```
de matraz importación Matraz
```

```
aplicación = Frasco (__nombre__)
```

360 Uso de Python para desarrollo web y API REST

@app.ruta('/')

```
definitivamente hola():  
    volver '¡Hola Mundo!'
```

@app.ruta('/saludo')

```
definitivamente saludo():  
    return '¡Saludos desde la aplicación web Flask!'
```

```
si __nombre__ == '__principal__':  
    aplicación.ejecutar()
```

Analicemos este ejemplo de código paso a paso:

1. **Inicialización:** una aplicación Flask debe crear una instancia de aplicación (aplicación en nuestro caso) como primer paso. El servidor web pasará todas las solicitudes de los clientes a esta instancia de aplicación utilizando un protocolo conocido como **Interfaz de puerta de enlace del servidor web (WSGI)**. La instancia de la aplicación se crea mediante la instrucción `app = Flask(__name__)`.

Es importante pasar el nombre del módulo como argumento al constructor Flask .

Flask usa este argumento para conocer la ubicación de la aplicación, que se convierte en una entrada para determinar la ubicación de otros archivos, como recursos estáticos, plantillas e imágenes. Usar `__name__` es la convención (sobre la configuración) para pasar al constructor de Flask , y Flask se encarga del resto.

2. **Ruta:** una vez que llega una solicitud a la instancia de la aplicación Flask , ahora es responsabilidad de la instancia ejecutar una determinada pieza de código para manejar la solicitud. Este fragmento de código, que suele ser una función de Python, se denomina controlador. La buena noticia es que cada solicitud generalmente (no todo el tiempo) está asociada con una sola URL, lo que hace posible definir una asignación entre una URL y una función de Python. Esta asignación de función de URL a Python se denomina ruta. En nuestro ejemplo de código, seleccionamos un enfoque simple para definir este mapeo usando el decorador de rutas . Por ejemplo, la URL /hola se asigna a la función hola y el /saludo

La URL está asignada a la función de saludo . Si preferimos definir todas las rutas en un solo lugar, podemos usar `add_url_rule` con la instancia de la aplicación para todas las definiciones de ruta.

3. **Función** de controlador: la función de controlador después de procesar la solicitud debe enviar una respuesta al cliente. Una respuesta puede ser una cadena simple con o sin HTML, o puede ser una página web compleja que puede ser estática o dinámica basada en una plantilla. En nuestro ejemplo de código, devolvemos una cadena simple con fines ilustrativos.

4. **Servidor web:** las aplicaciones de Flask vienen con un servidor de desarrollo que se puede iniciar con el método `app.run()`, o usando el comando de ejecución de Flask en un shell.

Cuando iniciamos este servidor web, busca un módulo `app.py` o `wsgi.py` por defecto, y se cargará automáticamente con el servidor si usamos la aplicación.

py nombre para nuestro archivo de módulo (nuevamente, convención sobre configuración). Pero si estamos usando un nombre diferente para nuestro módulo (es nuestro caso), debemos configurar una variable de entorno, `FLASK_APP = <nombre del módulo>`, que será utilizada por el servidor web para cargar el módulo.

Si ha creado un proyecto de Flask con un IDE como **PyCharm Pro**, la variable de entorno se establecerá automáticamente como parte de la configuración del proyecto. En caso de que esté utilizando un shell de línea de comandos, puede configurar la variable de entorno utilizando el siguiente comando según su sistema operativo:

```
export FLASK_APP= app1.py. #para macOS y Linux  
establecer FLASK_APP = app1.py. #para MS windows
```

Cuando se inicia el servidor, escucha las solicitudes de los clientes en la URL `http://localhost:5000/` y solo se puede acceder a él en su máquina local de forma predeterminada.

Si queremos iniciar el servidor usando un nombre de host y un puerto diferentes, podemos usar el siguiente comando (o la declaración de Python equivalente):

```
Flask run --host <dirección_ip> --port <número Puerto>
```

5. **Cliente web:** Podemos probar nuestra aplicación a través de un navegador ingresando la URL en la barra de direcciones o usando una utilidad `curl` para solicitudes HTTP simples. Para nuestro ejemplo, podemos probar nuestra aplicación usando los siguientes comandos curl :

```
curl -X OBTENER http://localhost:5000/  
curl -X OBTENER http://localhost:5000/saludo
```

Ahora que hemos terminado nuestra discusión sobre los fundamentos de una aplicación Flask, comenzaremos a explorar temas relacionados con el consumo de la solicitud y el envío de una respuesta dinámica a los clientes.

Manejo de solicitudes con diferentes tipos de métodos HTTP

HTTP se basa en un modelo de **solicitud-respuesta** entre un cliente y un servidor. Un cliente (por ejemplo, un navegador web) puede enviar diferentes verbos o, más apropiadamente, llamar a métodos para identificar el tipo de solicitud para el servidor. Estos métodos incluyen GET, POST, PUT, DELETE, HEAD, PATCH y OPTIONS. GET y POST son los métodos HTTP más utilizados, por lo que solo los cubriremos para ilustrar nuestros conceptos de desarrollo web.

362 Uso de Python para desarrollo web y API REST

Pero antes de analizar estos dos métodos, también es importante comprender los dos componentes clave de HTTP, que son **la solicitud HTTP** y **la respuesta HTTP**. Una solicitud HTTP se divide en tres partes:

- **Línea de solicitud:** Esta línea incluye el método HTTP a utilizar, la URI del solicitud y el protocolo HTTP (versión) que se utilizará:

OBTENER /casa HTTP/1.1

- **Campos de encabezado:** los encabezados son metadatos que brindan la información relacionada con la solicitud. Cada entrada de encabezado se proporciona como un par clave-valor, separados por dos puntos (:)
- **Cuerpo** (opcional): Este es un marcador de posición donde podemos agregar datos adicionales. Para una aplicación web, podemos enviar datos de formulario con solicitudes POST dentro del cuerpo de la solicitud HTTP. Para una API REST, podemos enviar datos para solicitudes PUT o POST dentro del cuerpo.

Cuando enviamos una solicitud HTTP a un servidor web, obtendremos como resultado una respuesta HTTP. La respuesta HTTP tendrá partes similares a la solicitud HTTP:

- **Línea de estado:** esta línea indica si la respuesta es exitosa o viene con un error. Se resalta un código de error en la línea de estado:

HTTP/1.1 200 Aceptar

El código de estado 200 o algo en el rango de 200-299 indica éxito. Los códigos de error están en el rango de 400-499 para errores del lado del cliente y en el rango de 500-599 para errores del lado del servidor.

- **Encabezado:** los campos de encabezado son similares a los campos de encabezado de solicitud HTTP.
- **Cuerpo** (opcional): aunque es opcional, esta es la parte clave de una respuesta HTTP. Esto puede incluir páginas HTML para aplicaciones web o datos en cualquier otro formato.

GET se utiliza para enviar una solicitud de un determinado recurso identificado en la URL con la opción de agregar una **cadena de consulta** como parte de la URL. Él ? se agrega en la URL para mantener la cadena de consulta separada de la URL base. Por ejemplo, si buscamos la palabra Python en Google, veremos una URL como la siguiente en el navegador:

<https://www.google.com/search?q=Python>

En esta URL, q=Python es una cadena de consulta. Una cadena de consulta se utiliza para transportar datos en forma de pares clave-valor. Este enfoque de acceso a los recursos es popular debido a su simplicidad, pero tiene sus limitaciones. Los datos de una cadena de consulta están visibles en la URL, lo que significa que no podemos enviar información confidencial, como un nombre de usuario y una contraseña, como una cadena de consulta. La longitud de la cadena de consulta no puede tener más de 255 caracteres. Sin embargo, el método GET se usa para buscar sitios web como Google y YAHOO por razones de simplicidad. En el caso del método POST , los datos se envían a través del cuerpo de la solicitud HTTP, lo que elimina las limitaciones del método GET . Los datos no aparecen como parte de la URL y no hay límite en cuanto a los datos que podemos enviar al servidor HTTP. Tampoco hay límite en cuanto a los tipos de datos admitidos mediante el método POST .

Flask proporciona algunas formas convenientes de identificar si una solicitud se envía mediante GET o POST o está utilizando cualquier otro método. En nuestro próximo ejemplo de código, ilustramos dos enfoques; el primer enfoque usa el decorador de rutas , con una lista exacta de los tipos de métodos esperados, y el segundo enfoque usa un decorador específico para el tipo de método HTTP, como el decorador get y el decorador post . El uso de ambos enfoques se ilustra en el siguiente ejemplo de código, seguido de un análisis detallado:

```
#app2.py: solicitud de mapa con tipo de método
de matraz importación Matraz, solicitud
aplicación = Frasco (__nombre__)
@app.route('/enviar', métodos=['GET'])
def req_with_get():
    volver "Recibí una solicitud de obtención"

@app.post('/enviar')
def req_with_post():
    volver "Recibí una solicitud de publicación"

@app.route('/submit2', métodos = ['GET', 'POST'])
def ambos_get_post():
    if solicitud.método == 'POST':
        return "Recibí una solicitud de publicación 2"
    demás:
        devuelve "Recibí una solicitud de obtención 2"
```

364 Uso de Python para desarrollo web y API REST

Analicemos las tres definiciones de ruta y las funciones correspondientes en nuestro código de muestra una por una:

- En la primera definición de ruta (@app.route('/submit', method=['GET'])), usamos el decorador de rutas para mapear una URL, con solicitudes del tipo GET a una función de Python. Con esta configuración de decorador, nuestra función de Python manejará las solicitudes con el método GET solo para / URL de envío .
- En la definición de la segunda ruta (@app.post('/submit')), usamos la publicación decorador y solo especifique la URL de solicitud con él. Esta es una versión simplificada de mapear una solicitud con el método POST a una función de Python. Esta nueva configuración es equivalente a la primera definición de ruta, pero con el tipo de método POST en una forma simplificada. Podemos lograr lo mismo para un método GET usando el decorador get .
- En la definición de la tercera ruta (@app.route('/submit2', métodos = ['GET', 'POST'])), mapeamos una sola URL con solicitudes usando los métodos POST y GET a una sola función de Python. Este es un enfoque conveniente cuando esperamos manejar cualquier método de solicitud mediante el uso de un solo controlador (función de Python). Dentro de la función de Python, usamos el atributo de método del objeto de solicitud para identificar si la solicitud es del tipo GET o POST . Tenga en cuenta que el servidor web pone a disposición el objeto de solicitud para nuestra aplicación Flask una vez que importamos el paquete de solicitud a nuestro programa. Este enfoque brinda flexibilidad para que los clientes envíen solicitudes utilizando cualquiera de los dos métodos utilizando la misma URL y, como desarrollador, los asignamos a una sola función de Python.

Podemos probar este ejemplo de código de manera más conveniente a través de la utilidad curl porque no será fácil enviar una solicitud POST sin definir un formulario HTML. Los siguientes comandos curl se pueden usar para enviar solicitudes HTTP a nuestra aplicación web:

```
curl -X OBTENER http://localhost:5000/enviar
curl -X POST http://localhost:5000/enviar
curl -X OBTENER http://localhost:5000/submit2
curl -X POST http://localhost:5000/submit2
```

A continuación, discutiremos cómo generar una respuesta desde páginas estáticas y plantillas.

Representación de contenidos estáticos y dinámicos

Los contenidos estáticos son importantes para una aplicación web ya que incluyen archivos CSS y JavaScript. Los archivos estáticos pueden ser servidos directamente por el servidor web. Flask también puede hacer que esto suceda si creamos un directorio llamado estático en nuestro proyecto y redirigimos al cliente a la ubicación del archivo estático.

Los contenidos dinámicos se pueden crear utilizando Python, pero es tedioso y requiere un gran esfuerzo para mantener dicho código en Python. El enfoque recomendado es usar un motor de plantillas como **Jinja2**. Flask viene con una biblioteca Jinja2, por lo que no hay una biblioteca adicional para instalar, ni necesitamos agregar ninguna configuración adicional para configurar Jinja2. A continuación se muestra un código de muestra con dos funciones, una que maneja una solicitud de contenido estático y la otra para contenido dinámico:

```
#app3.py: representación de contenidos estáticos y dinámicos
desde matraz importar Flask, render_template, url_for, redirect

aplicación = Frasco (__nombre__)

@app.ruta('/hola')
definitivamente hola():
    hola_url = url_for ('estática', nombre de archivo='app3_s.html')
    devolver redirección (hello_url)

@app.ruta('/saludo')
definitivamente saludo():
    mensaje = "Hola desde Python"
    volver render_template('app3_d.html', saludo=mensaje)
```

Para una mejor comprensión de este código de ejemplo, destacaremos los puntos clave:

- Importamos módulos adicionales de Flask como url_for, redirect y render_template.
- Para la ruta /hello , construimos una URL usando la función url_for con el directorio estático y el nombre del archivo HTML como argumentos. Enviamos la respuesta, que es una instrucción al navegador, para redirigir al cliente a la URL de una ubicación de archivo estático. Las instrucciones de redireccionamiento se indican al navegador web mediante el uso de un código de estado en el rango de 300-399, que Flask configura automáticamente cuando usamos la función de redireccionamiento .

366 Uso de Python para desarrollo web y API REST

- Para la ruta /greeting , renderizamos una plantilla Jinja, app3_d.html, usando la función render_template . También pasamos una cadena de mensaje de saludo como valor a una variable para la plantilla. La variable de saludo estará disponible para la plantilla Jinja, como se muestra en el siguiente extracto de plantilla del archivo app3_d.html :

```
<!DOCTYPEhtml>
<cuerpo>
{%
    si saludo %}
<h1> {{saludo}}!</h1>
{%
    terminara si %}
</cuerpo>
</html>
```

Esta es la plantilla Jinja más simple con una instrucción if encerrada entre <% %>, y la variable de Python se incluye usando los dos corchetes {{}} formato. No entraremos en detalles sobre las plantillas de Jinja2, pero le recomendamos que se familiarice con las plantillas de Jinja2 a través de su documentación en línea (<https://jinja.palletsprojects.com/>).

Se puede acceder a esta aplicación web de muestra usando un navegador web y usando el curl utilidad. En la siguiente sección, discutiremos cómo extraer parámetros de diferentes tipos de solicitudes.

Extraer parámetros de una solicitud HTTP

Las aplicaciones web se diferencian de los sitios web porque interactúan con los usuarios y esto no es posible sin el intercambio de datos entre un cliente y un servidor. En esta sección, discutiremos cómo extraer datos de una solicitud. Según el tipo de método HTTP utilizado, adoptaremos un enfoque diferente. Cubriremos los siguientes tres tipos de solicitudes de la siguiente manera:

- Parámetros como parte de la URL de solicitud
- Parámetros como cadena de consulta con una solicitud GET
- Parámetros como datos de formulario HTML con una solicitud POST

A continuación se muestra un código de muestra con tres rutas diferentes para cubrir los tres tipos de solicitud antes mencionados. Estamos renderizando una plantilla Jinja (app4.html), que es la misma que usamos para el archivo app3_d.html , excepto que el nombre de la variable es nombre en lugar de saludo:

```
#app4.py: extracción de parámetros de diferentes solicitudes
de matraz importar Matraz, solicitud, render_template

aplicación = Frasco (__nombre__)

@app.ruta('/hola')
@app.route('/hola/<fname> <lname>')
def hola_usuario(fname=Ninguno, apellido=Ninguno):
    devuelve render_template('aplicación4.html', nombre=f'{fnombre}{lnombre}')

@app.get('/enviar')
def proceso_get_request_data():
    fname = solicitud.args['fname']
    lname = request.args.get('lname', '')
    devuelve render_template('aplicación4.html', nombre=f'{fnombre}{lnombre}')

@app.post('/enviar')
def proceso_post_solicitud_datos():
    fname = solicitud.formulario['fname']
    nombre = solicitud.formulario.get('nombre', '')
    devuelve render_template('aplicación4.html', nombre=f'{fnombre}{lnombre}')
```

A continuación, discutiremos el enfoque de extracción de parámetros para cada caso:

- Para el primer conjunto de rutas (app.route), definimos una ruta de manera que cualquier texto después de /hola/ se considere un parámetro con la solicitud. Podemos establecer cero o uno o dos parámetros, y nuestra función de Python puede manejar cualquier combinación y devuelve el nombre (que puede estar vacío) a la plantilla como respuesta. Este enfoque es bueno para casos simples de paso de parámetros al programa del servidor. Esta es una opción popular en el desarrollo de API REST para acceder a una sola instancia de recurso.

- Para la segunda ruta (app.get), estamos extrayendo parámetros de cadena de consulta del objeto de diccionario args . Podemos obtener el valor del parámetro usando su nombre como clave de diccionario o usando el método GET con el segundo argumento como valor predeterminado. Usamos una cadena vacía como valor predeterminado con el método GET .
Mostramos ambas opciones, pero recomendamos usar el método GET si desea establecer un valor predeterminado en caso de que no exista ningún parámetro en la solicitud.
- Para la tercera ruta (app.post), los parámetros vienen como datos de formulario como parte del cuerpo de la solicitud HTTP y usaremos el objeto de diccionario de formulario para extraer estos parámetros. Nuevamente, usamos el nombre del parámetro como clave de diccionario y también usamos el método GET con fines ilustrativos.
- Para probar estos escenarios, recomendamos usar la utilidad curl , especialmente para POST peticiones. Probamos la aplicación con los siguientes comandos:

```
curl -X OBTENER http://localhost:5000/hola  
curl -X OBTENER http://localhost:5000/hello/jo%20so  
curl -X GET 'http://localhost:5000/submit?fname=jo&lname=so'  
curl -d "fname=jo&lname=so" -X POST http://localhost:5000/  
entregar
```

En la siguiente sección, discutiremos cómo interactuar con una base de datos en Python.

Interactuar con sistemas de bases de datos.

Una aplicación web de pila completa requiere la persistencia de datos estructurados, por lo que el conocimiento y la experiencia de trabajar con una base de datos son requisitos previos para el desarrollo web. Python y Flask pueden integrarse con la mayoría de los sistemas de bases de datos SQL o no SQL. Python en sí viene con una base de datos SQLite ligera con el nombre de módulo sqlite3. Usaremos SQLite porque no requiere configurar un servidor de base de datos separado y funciona muy bien para aplicaciones de pequeña escala. Para entornos de producción, debemos usar otros sistemas de base de datos, como MySQL o MariaDB, o PostgreSQL. Para acceder e interactuar con un sistema de base de datos, usaremos una de las extensiones de Flask, Flask-SQLAlchemy. La extensión Flask-SQLAlchemy se basa en la biblioteca SQLAlchemy de Python y hace que la biblioteca esté disponible para nuestra aplicación web. La biblioteca SQLAlchemy proporciona un **asignador relacional de objetos (ORM)**, lo que significa que asigna nuestras tablas de base de datos a objetos de Python. El uso de una biblioteca ORM no solo acelera el ciclo de desarrollo, sino que también proporciona la flexibilidad para cambiar el sistema de base de datos subyacente sin cambiar nuestro código. Por lo tanto, recomendaremos usar SQLAlchemy o una biblioteca similar para trabajar con sistemas de bases de datos.

Para interactuar con cualquier sistema de base de datos desde nuestra aplicación, debemos crear nuestra instancia de aplicación Flask como de costumbre. El siguiente paso es configurar la instancia de la aplicación con la URL para la ubicación de nuestra base de datos (un archivo en el caso de SQLite3). Una vez que se crea la instancia de la aplicación, crearemos una instancia de SQLAlchemy pasándola a la instancia de la aplicación. Al usar una base de datos como SQLite, solo tenemos que inicializar la base de datos la primera vez. Se puede iniciar desde un programa de Python, pero no favoreceremos este enfoque para evitar que la base de datos se reinicie cada vez que iniciemos nuestra aplicación. Se recomienda inicializar la base de datos una sola vez desde una línea de comando utilizando la instancia de SQLAlchemy. Discutiremos los pasos exactos para inicializar la base de datos después de nuestro ejemplo de código.

Para ilustrar el uso de la biblioteca SQLAlchemy con nuestra aplicación web, crearemos una aplicación simple para agregar, enumerar y eliminar objetos de estudiantes de una tabla de base de datos. Aquí hay un código de muestra de la aplicación para inicializar la aplicación Flask y la instancia de la base de datos (una instancia de SQLAlchemy) y también para crear un objeto Modelo de la clase Estudiante :

```
#app5.py (parte 1): interactuar con db para crear, eliminar y listar objetos

desde matraz importar Flask, solicitud, render_template, redirigir
desde el matraz_sqlalchemy importar SQLAlchemy

aplicación = Frasco (__nombre__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///student.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = Falso
db = SQLAlchemy(aplicación)

estudiante de clase (db.Modelo):
    id = db.Column(db. Entero, clave_principal=Verdadero)
    nombre = db.Column(db.String(80), anulable=Falso)
    grado = db.Column(db.String(20), anulable=True)

def __repr__(uno mismo):
    return '<Estudiante %r>' % self.name
```

370 Uso de Python para desarrollo web y API REST

Una vez que hemos creado la instancia de SQLAlchemy, db, podemos trabajar con los objetos de la base de datos. La belleza de una biblioteca ORM como SQLAlchemy es que podemos definir un esquema de base de datos, conocido como **modelo** en la terminología ORM, dentro de Python como clases de Python. Para nuestro ejemplo de código, creamos una clase, Student, que se hereda de una clase base, db.Model. En esta clase modelo, definimos los atributos id, name y grade que corresponderán a tres columnas en una tabla de base de datos, Student, en la instancia de la base de datos SQLite3 . Para cada atributo, definimos su tipo de datos con la longitud máxima, si tiene una clave principal y si es anulable. Estas definiciones de atributos adicionales son importantes para configurar las tablas de la base de datos de forma optimizada.

En el siguiente fragmento de código, ilustraremos una función de Python, list_students, para obtener una lista de objetos de estudiantes de la base de datos. Esta función está asignada a /list URL de nuestra aplicación web de muestra y devuelve todos los objetos Student de la tabla de la base de datos utilizando el método all en la instancia de consulta (un atributo de la base de datos instancia). Tenga en cuenta que la instancia de consulta y sus métodos están disponibles en la clase base, db.Model:

```
#app5.py (parte 2)
@app.get('/lista')
def lista_estudiantes():
    lista_estudiantes = Estudiante.consulta.todo()
    volver render_template('app5.html',
        estudiantes=lista_estudiantes)
```

En el siguiente fragmento de código, escribiremos una función (add_student) para agregar estudiantes a la tabla de la base de datos. Esta función está asignada a /add URL y espera que el nombre del estudiante y su calificación se pasen como parámetros de solicitud mediante el método GET . Para agregar un nuevo objeto a la base de datos, crearemos una nueva instancia de la clase Student con los valores de atributo requeridos y luego usaremos la instancia db.Session para agregar a la capa ORM usando la función de agregar . La función de agregar no agregará la instancia a la base de datos por sí misma.

Usaremos el método de confirmación para enviarlo a la tabla de la base de datos. Una vez que se agrega un nuevo estudiante a nuestra tabla de base de datos, redirigimos el control a la URL /list . La razón por la que usamos una redirección a esta URL es que queremos devolver la última lista de estudiantes después de agregar una nueva y reutilizar la función list_students , que ya implementamos. El código completo para la función add_student es el siguiente:

```
#app5.py(partie 3)
@app.get('/añadir')
def agregar_estudiante():
    fname = solicitud.args['fname']
    lname = request.args.get('lname', '')
```

```
grado = solicitud.args.get('grado', '') estudiante =
Estudiante(nombre=f'{fname} {lname}', grado=grado)
db.session.add(estudiante)
db.sesión.commit()
devolver redirección("/lista")
```

En la última parte de este ejemplo de código, escribiremos una función de Python (`delete_student`) para eliminar un estudiante de la tabla de la base de datos. Esta función está asignada a la URL /`delete<int:id>`. Tenga en cuenta que esperamos que el cliente envíe la identificación del estudiante (que enviamos con una lista de estudiantes usando la solicitud de lista). Para eliminar un estudiante, primero, consultamos la instancia exacta del estudiante usando la identificación del estudiante. Esto se logra utilizando el método `filter_by` en la instancia de consulta. Una vez que tenemos la instancia de Estudiante exacta, usamos el método de eliminación de la instancia de `db.Session` y luego confirmamos los cambios. Al igual que con la función `add_student`, redirigimos al cliente a la URL /list para devolver una lista actualizada de estudiantes a nuestra plantilla Jinja:

```
#app5.py (parte 4)
@app.get('/delete/<int:id>') def
del_student(id): todelete =
Student.query.filter_by(id=id).first()
db.session.delete(para eliminar)
db.sesión.commit()
devolver redirección("/lista")
```

Para mostrar una lista de estudiantes en un navegador, creamos una plantilla Jinja simple (`app5.html`). El archivo de plantilla `app5.html` proporcionará una lista de estudiantes en formato de tabla. Es importante tener en cuenta que usamos un bucle `for` de Jinja para construir las filas de la tabla HTML dinámicamente, como se muestra en la siguiente plantilla de Jinja:

```
<!DOCTYPEhtml>
<body>
<h2>Estudiantes</


## 


```

372 Uso de Python para desarrollo web y API REST

```
</tr>
</thead>
<tbody>
{%- para estudiante en estudiantes%}
<tr>
<th scope="row">{{student.id}}</th>
<td>{{estudiante.nombre}}</td>
<td>{{estudiante.grado}}</td>
</tr>
{%- endfor%}
</tbody>
</tabla>
{%- endif%}
</cuerpo>
</html>
```

Antes de iniciar esta aplicación, debemos inicializar el esquema de la base de datos como un paso único. Esto se puede hacer usando un programa de Python, pero debemos asegurarnos de que el código se ejecute solo una vez o solo cuando la base de datos aún no esté inicializada. Un enfoque recomendado es hacer este paso manualmente usando el shell de Python. En el shell de Python, podemos importar la instancia de db desde nuestro módulo de aplicación y luego usar db.create_all método para inicializar la base de datos según las clases modelo definidas en nuestro programa. Estos son los comandos de muestra que se utilizarán para nuestra aplicación para la inicialización de la base de datos:

```
>>> desde app5 importar db
>>> db.create_all()
```

Estos comandos crearán un archivo student.db en el mismo directorio donde tenemos nuestro programa. Para restablecer la base de datos, podemos eliminar el archivo student.db y volver a ejecutar los comandos de inicialización, o podemos usar el método db.drop_all en el shell de Python.

Podemos probar la aplicación usando la utilidad curl o a través de un navegador usando las siguientes URLs:

- <http://localhost:5000/lista>
- <http://localhost:5000/add?fname=John&Lee=asif&grade=9>
- <http://hostlocal:5000/eliminar/<id>>

A continuación, analizaremos cómo manejar los errores en una aplicación web basada en Flask.

Manejo de errores y excepciones en aplicaciones web

En todos nuestros ejemplos de código, no prestamos atención a cómo lidiar con situaciones cuando un usuario ingresa una URL incorrecta en su navegador o envía un conjunto incorrecto de argumentos a nuestra aplicación. Esta no era una intención de diseño, pero el objetivo era centrarse primero en los componentes clave de las aplicaciones web. La belleza de los marcos web es que, por lo general, admiten el manejo de errores de forma predeterminada. Si se produce algún error, se devuelve automáticamente un código de estado apropiado. Los códigos de error están bien definidos como parte del protocolo HTTP.

Por ejemplo, los códigos de error del 400 al 499 indican errores con las solicitudes del cliente, y los códigos de error del 500 al 599 indican problemas con el servidor al ejecutar la solicitud. A continuación se resumen algunos errores comúnmente observados:

Error Code	Name	Description
400	Bad Request	This error indicates either the URL or the request contents are incorrect.
401	Unauthorized	This error appears when the username or password is not provided or not correctly provided with the request.
403	Forbidden	This error shows that a user is trying to access a resource that is not allowed for that user.
404	Not Found	This error is returned when we are trying to access a resource that is not available. This is usually for the wrong URL.
500	Internal Server Error	This error indicates that the request is correct but that something happened on the server side.

Tabla 10.1 – Errores HTTP comúnmente observados

Una lista completa de códigos de estado HTTP y errores está disponible en <https://httpstatus.com/>.

El marco Flask también viene con un marco de manejo de errores. Al manejar las solicitudes de los clientes, si nuestro programa falla, se devuelve un error interno del servidor 500 de forma predeterminada. Si un cliente solicita una URL que no está asignada a ninguna función de Python, Flask devolverá un error 404 No encontrado al cliente. Estos diferentes tipos de errores se implementan como subclases de la clase `HTTPException`, que forma parte de la biblioteca Flask.

374 Uso de Python para desarrollo web y API REST

Si queremos manejar estos errores o excepciones con un comportamiento personalizado o un mensaje personalizado, podemos registrar nuestro controlador con la aplicación Flask. Tenga en cuenta que un controlador de errores es una función en Flask que solo se activa cuando ocurre un error, y podemos asociar un error específico o una excepción genérica con nuestros controladores. Construimos un código de muestra para ilustrar el concepto a un alto nivel. Primero, ilustraremos una aplicación web simple con dos funciones (hola y saludo) para manejar dos URL, como se muestra en el siguiente código de ejemplo:

```
#app6.py(part 1): manejo de errores y excepciones
importar json
desde matraz importar Flask, render_template, abortar
de werkzeug.Exceptions importar HTTPException

aplicación = Frasco (__nombre__)

@app.ruta('/')
definitivamente hola():
    volver '!Hola Mundo!'

@app.ruta('/saludo')
definitivamente saludo():
    X = 10/0
    return '!Saludos desde la aplicación web Flask!'
```

Para manejar los errores, registraremos nuestro controlador en la instancia de la aplicación mediante el decorador `errorHandler`. Para nuestro código de muestra (que se muestra a continuación), registramos una página _controlador `not_found` contra el código de error 404 para la aplicación Flask. Para el código de error 500, registramos una función `internal_error` como controlador de errores. Al final, registramos `generic_handler` para la clase `HTTPException`. Este controlador genérico detectará el error o la excepción que no sean 404 y 500. A continuación, se muestra un código de muestra con los tres controladores:

```
#app6.py(part 2)
@app.controlador de errores(404)
def página_no_encontrada(error):
    devolver render_template('error404.html'), 404
@app.controlador de errores(500)
def error_interno(error):
    devolver render_template('error500.html'), 500
```

```
@app.controlador de errores (Excepción HTTP)
def generic_handler(error):
    error_detail = json.dumps({
        "código": error.código,
        "nombre": error.nombre,
        "descripción": error.descripcion,
    })
    devuelve render_template('error.html',
        err_msg=error_detail), error.code
```

Con fines ilustrativos, también escribimos plantillas Jinja básicas con mensajes personalizados; error404.html, error500.html y error.html. Las plantillas error404.html y error500.html utilizan el mensaje que está codificado en la plantilla.

Sin embargo, la plantilla error.html espera que el mensaje personalizado provenga del servidor web. Para probar estas aplicaciones de muestra, solicitaremos lo siguiente a través de un navegador o la utilidad curl :

- GET <http://localhost:5000/>: Esperaremos una respuesta normal en este caso.
- GET <http://localhost:5000/hello>: Esperaremos un error 404 como no hay ninguna función de Python asignada a esta URL y la aplicación Flask generará una plantilla error404.html .
- OBTENER <http://localhost:5000/saludo>: Esperaremos un error 500 porque tratamos de dividir un número por cero para elevar el ZeroDivisionError error. Dado que este es un error del lado del servidor (500), activará nuestro l_error interno handler, que genera una plantilla genérica error500.html .
- POST <http://localhost:5000/>: para emular el rol de un controlador genérico, enviaremos una solicitud que active un código de error que no sea 404 y 500. Esto es posible fácilmente mediante el envío de una solicitud POST para una URL que es esperando un GET solicitud y el servidor generará un error 405 en este caso (para un método HTTP no compatible). No tenemos un controlador de errores específico para el código de error 405 en nuestra aplicación, pero hemos registrado un controlador genérico con HTTPException. Este controlador genérico manejará este error y generará una plantilla error.html genérica.

376 Uso de Python para desarrollo web y API REST

Esto concluye nuestra discusión sobre el uso del marco Flask para el desarrollo de aplicaciones web. A continuación, exploraremos la creación de una API REST utilizando extensiones de Flask.

Creación de una API REST

REST, o ReST, es un acrónimo de **Representational State Transfer**, que es una arquitectura para que las máquinas cliente soliciten información sobre los recursos que existen en una máquina remota. **API** significa **Interfaz de programación de aplicaciones**, que es un conjunto de reglas y protocolos para interactuar con el software de aplicación que se ejecuta en diferentes máquinas. La interacción de diferentes entidades de software no es un requisito nuevo. En las últimas décadas, se han propuesto e inventado muchas tecnologías para hacer que las interacciones a nivel de software sean fluidas y convenientes. Algunas tecnologías notables incluyen la llamada a **procedimiento remoto (RPC)**, la **invocación de método remoto (RMI)**, CORBA y los servicios web SOAP. Estas tecnologías tienen limitaciones en términos de estar vinculadas a un determinado lenguaje de programación (por ejemplo, RMI) o vinculadas a un mecanismo de transporte patentado, o utilizando solo un determinado tipo de formato de datos. Estas limitaciones han sido eliminadas casi por completo por la API RESTful, que se conoce comúnmente como la API REST.

La flexibilidad y la simplicidad del protocolo HTTP lo convierten en un candidato favorable para usar como mecanismo de transporte para una API REST. Otra ventaja de usar HTTP es que permite varios formatos de datos para el intercambio de datos (como texto, XML y JSON) y no está limitado a un formato, como XML es el único formato para las API basadas en SOAP. La API REST no está vinculada a ningún idioma específico, lo que la convierte en una opción de facto para crear API para interacciones web. A continuación se muestra una vista arquitectónica de una llamada API REST de un cliente REST a un servidor REST mediante HTTP:

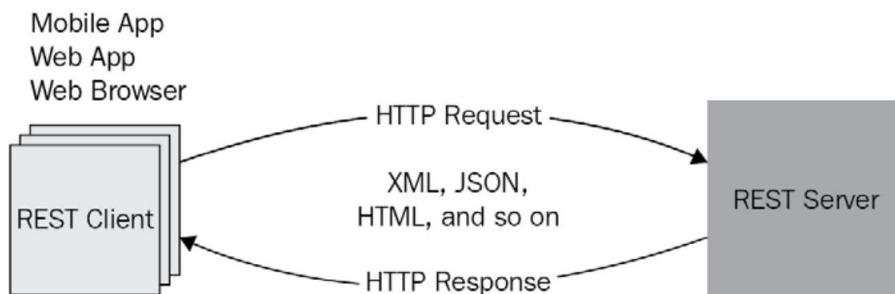


Figura 10.1 – Interacción API REST entre clientes y un servidor

Una API REST se basa en solicitudes HTTP y utiliza sus métodos nativos, como GET, PUT, POST y DELETE. El uso del método HTTP simplifica la implementación del software del lado del cliente y del lado del servidor desde una perspectiva de diseño de API. Se desarrolla una API REST teniendo en cuenta el concepto de operaciones CRUD. **CRUD** significa **Crear, Leer, Actualizar y Eliminar**.

Aquí es donde los métodos HTTP se alinean uno a uno con las operaciones CRUD, por ejemplo, GET para leer, POST para crear, PUT para actualizar y DELETE para eliminar .

Al crear una API REST con métodos HTTP, debemos tener cuidado al elegir el método correcto en función de sus capacidades de idempotencia. Una operación se considera idempotente en matemáticas si la operación da los mismos resultados incluso si se repite varias veces. Desde la perspectiva del resto del diseño de la API, el método POST no es idempotente, lo que significa que debemos asegurarnos de que los clientes de la API no inicien una solicitud POST varias veces para el mismo conjunto de datos. OBTENER , PONER y ELIMINAR Los métodos son idempotentes, aunque es muy posible obtener un código de error 404 si intentamos eliminar el mismo recurso por segunda vez. Sin embargo, este es un comportamiento aceptable desde el punto de vista de la idempotencia.

Uso de Flask para una API REST

Se puede construir una API REST en Python utilizando diferentes bibliotecas y marcos. Los marcos más populares para construir una API REST son Django, Flask (usando **Flask-RESTful** extensión) y **FastAPI**. Cada uno de estos marcos tiene ventajas y desventajas. Django es una opción adecuada para crear una API REST si la aplicación web también se está creando con Django. Sin embargo, usar Django solo para el desarrollo de API sería excesivo. La extensión Flask-RESTful funciona a la perfección con una aplicación web Flask. Tanto Django como Flask cuentan con un fuerte apoyo de la comunidad, que a veces es un factor importante al seleccionar una biblioteca o un marco. FastAPI se considera el mejor en rendimiento y es una buena opción si el objetivo es solo crear una API REST para su aplicación. Sin embargo, el soporte de la comunidad para FastAPI no está al mismo nivel que tenemos para Django y Flask.

Seleccionamos una extensión Flask RESTful para el desarrollo de la API REST para continuar con nuestra discusión que habíamos comenzado para el desarrollo de aplicaciones web. Tenga en cuenta que podemos crear una API web simple usando solo Flask, y lo hicimos en el capítulo anterior cuando desarrollamos una aplicación basada en un servicio web de muestra para la implementación de Google Cloud. En esta sección, nos centraremos en el uso del estilo arquitectónico REST para crear la API. Esto significa que usaremos el método HTTP para realizar una operación en un recurso que será representado por un objeto de Python.

378 Uso de Python para desarrollo web y API REST

Importante

La compatibilidad con Flask-RESTful es única en la forma en que proporciona una forma conveniente de configurar el código de respuesta y el encabezado de respuesta como parte de las declaraciones de devolución.

Para usar Flask y la extensión Flask-RESTful, se nos pedirá que instalamos la extensión Flask-RESTful. Podemos instalarlo en nuestro entorno virtual usando el siguiente comando pip :

instalación de pip Flask-RESTful

Antes de discutir cómo implementar una API REST, es conveniente familiarizarse con algunos términos y conceptos relacionados con la API.

Recurso

Un recurso es un elemento clave para una API REST, y funciona con la extensión Flask-RESTful. Un objeto de recurso se define extendiendo nuestra clase desde el recurso base class, que está disponible en la biblioteca de extensiones Flask-RESTful. El recurso base class ofrece varias funciones mágicas para ayudar al desarrollo de API y asocia automáticamente los métodos HTTP con los métodos de Python definidos en nuestro objeto de recurso.

punto final de la API

Un punto final de API es un punto de entrada para establecer la comunicación entre el software del cliente y el software del servidor. En términos simples, un punto final de API es una terminología alternativa para una URL de un servidor o servicio donde un programa está escuchando solicitudes de API. Con la extensión Flask-RESTful, definimos un punto final de API al asociar una determinada URL (o URL) con un objeto de recurso. En la implementación de Flask, implementamos un objeto de recurso al extenderlo desde la clase de recurso base .

Enrutamiento

El concepto de enrutamiento para API es similar al enrutamiento de aplicaciones web en Flask, con la única diferencia de que, en el caso de una API, necesitamos asignar un objeto de recurso a una o más URL de punto final.

Análisis de argumentos

El análisis de argumentos de solicitud para una API es posible mediante el uso de la cadena de consulta o datos codificados en formulario HTML. Sin embargo, este no es un enfoque preferido porque tanto la cadena de consulta como los formularios HTML no están destinados ni diseñados para usarse con una API. El enfoque recomendado es extraer los argumentos directamente de una solicitud HTTP. Para facilitar esto, la extensión Flask-RESTful ofrece una clase especial, `reqparse`. Esta clase `reqparse` es similar a `argparse`, que es una opción popular para analizar argumentos de línea de comandos.

A continuación, aprenderemos a crear una API REST para acceder a los datos de un sistema de base de datos.

Desarrollo de una API REST para el acceso a la base de datos

Para ilustrar el uso de Flask y la extensión Flask-RESTful para crear una API REST, revisaremos nuestra aplicación web (`app5.py`) y ofreceremos acceso al objeto Student (un objeto Resource) utilizando el estilo arquitectónico REST. Esperamos los argumentos enviados para los métodos PUT y POST dentro del cuerpo de la solicitud y la API devolverá la respuesta en formato JSON. El código revisado de `app5.py` con una interfaz REST API se muestra a continuación:

```
#api_app.py: aplicación REST API para recursos de estudiantes
```

```
desde el matraz_sqlalchemy importar SQLAlchemy
```

```
de matraz importación Matraz
```

```
desde el recurso de importación de flask_restful, Api, reqparse
```

```
aplicación = Frasco (__nombre__)
```

```
API = API (aplicación)
```

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///student.db'
```

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = Falso
```

```
db = SQLAlchemy(aplicación)
```

380 Uso de Python para desarrollo web y API REST

En el fragmento de código anterior, comenzamos con la inicialización de la aplicación Flask y la instancia de la base de datos. Como siguiente paso, creamos una instancia de API utilizando la instancia de Flask. Logramos esto por medio de la instrucción `api = Api(app)`. Esta instancia de API es la clave para desarrollar el resto de la aplicación de API y la usaremos.

A continuación, debemos configurar la instancia de reqparse registrando el argumento que esperamos analizar de la solicitud HTTP. En nuestro ejemplo de código, registramos dos argumentos del tipo de cadena, nombre y calificación, como se muestra en el siguiente fragmento de código:

```
anificador = reqparse.RequestParser()
parser.add_argument('nombre', tipo=cadena)
anificador.add_argument('grado', tipo=cadena)
```

El siguiente paso es crear un objeto de modelo `Student`, que es prácticamente igual que hicimos en `app5.py`, excepto que agregaremos un método de serialización para convertir nuestro objeto en formato JSON. Este es un paso importante para serializar la respuesta JSON antes de devolverla a los clientes de la API. Hay otras soluciones disponibles para lograr lo mismo, pero seleccionamos esta opción por razones de simplicidad. El código de ejemplo preciso para la creación de un Estudiante objeto es el siguiente:

```
estudiante de clase (db.Modelo):
id = db.Column(db. Entero, clave_principal=Verdadero)
nombre = db.Column(db.String(80), anutable=False)
grado = db.Column(db.String(20), anutable=True)

def serializar (auto):
devolver {
'id': self.id,
'nombre': propio.nombre,
'grado': self.grade
}
```

A continuación, creamos dos clases de recursos para acceder a los objetos de la base de datos de los estudiantes. Estos son `StudentDao` y `StudentListDao`. Estos se describen a continuación:

- `StudentDao` ofrece métodos como `get` y `delete` en la instancia de recurso individual, y estos métodos se asignan a los métodos GET y DELETE del protocolo HTTP.

- StudentListDao ofrece métodos como get y post. El método GET es se agrega para proporcionar una lista de todos los recursos del tipo Estudiante usando el método GET HTTP, y se incluye el método POST para agregar un nuevo objeto de recurso usando el método POST HTTP. Este es un patrón de diseño típico que se usa para implementar la funcionalidad CRUD para un recurso web.

En cuanto a los métodos implementados para las clases StudentDao y StudentListDao , devolvemos el código de estado y el objeto mismo en una sola declaración. Esta es una comodidad que ofrece la extensión Flask-RESTful.

A continuación se muestra un código de muestra para la clase StudentDao :

```
clase StudentDao(Recurso):  
  
def get(self, id_estudiante):  
    estudiante = Estudiante.consulta.filter_by(id=student_id)\\  
    first_or_404(description='El registro con id={} no está disponible'.format(student_id))  
  
    devolver estudiante. serializar ()  
  
  
def delete(self, id_estudiante):  
    estudiante = Estudiante.consulta.filter_by(id=student_id)\\  
    first_or_404(description='El registro con id={} no está disponible'.format(student_id))  
  
    db.session.delete(estudiante)  
    db.sesión.commit()  
    volver ", 204"
```

Un código de muestra para la clase StudentListDao es el siguiente:

```
clase StudentListDao(Recurso):  
  
def obtener(auto):  
    estudiantes = Estudiante.consulta.todos()  
    volver [Estudiante. serializar (estudiante) para estudiante en estudiantes]  
  
  
def publicar (auto):  
    argumentos = analizador.parse_args()  
    nombre = argumentos['nombre']  
    calificación = args['calificación']  
    alumno = alumno(nombre=nombre, grado=grado)
```

382 Uso de Python para desarrollo web y API REST

```
db.session.add(estudiante)
db.sesión.commit()
estudiante de regreso , 200
```

Para nuestro método de publicación de la clase StudentListDao , usamos el reqparse analizador para extraer el nombre y los argumentos de calificación de la solicitud. El resto de la implementación en el método POST es el mismo que hicimos para el ejemplo app5.py.

En las siguientes dos líneas de nuestra aplicación API de muestra, asignamos URL a nuestro recurso objetos. Todas las solicitudes que lleguen para /students/<student_id> se redirigirán a la clase de recursos StudentDao . Cualquier solicitud que llegue para /estudiantes se redirigirá a la clase de recursos StudentListDao :

```
api.add_resource(StudentDao, '/students/<student_id>')
api.add_resource(StudentListDao, '/estudiantes')
```

Tenga en cuenta que omitimos la implementación del método PUT de la clase StudentDao , pero está disponible con el código fuente proporcionado para este capítulo para completarlo. Para este ejemplo de código, no agregamos el manejo de errores y excepciones para mantener el código conciso para nuestra discusión, pero es muy recomendable tener esto incluido en nuestra implementación final.

En esta sección, hemos cubierto los conceptos básicos y los principios de implementación para desarrollar API REST que sean adecuadas para que cualquier persona comience a crear una API REST.

En la siguiente sección, ampliaremos nuestro conocimiento para construir una aplicación web completa basada en una API REST.

Estudio de caso: creación de una aplicación web con la API REST

En este capítulo, aprendimos cómo crear una aplicación web simple usando Flask y cómo agregar una API REST a una capa de lógica empresarial usando una extensión de Flask. En el mundo real, las aplicaciones web suelen tener tres niveles: capa web, capa de lógica empresarial y capa de acceso a datos.

Con la popularidad de las aplicaciones móviles, la arquitectura ha evolucionado para tener una API REST como elemento básico para la capa empresarial. Esto brinda la libertad de crear aplicaciones web y aplicaciones móviles utilizando la misma capa de lógica empresarial. Además, la misma API puede estar disponible para interacciones B2B con otros proveedores. Este tipo de arquitectura se captura en la Figura 10.2:

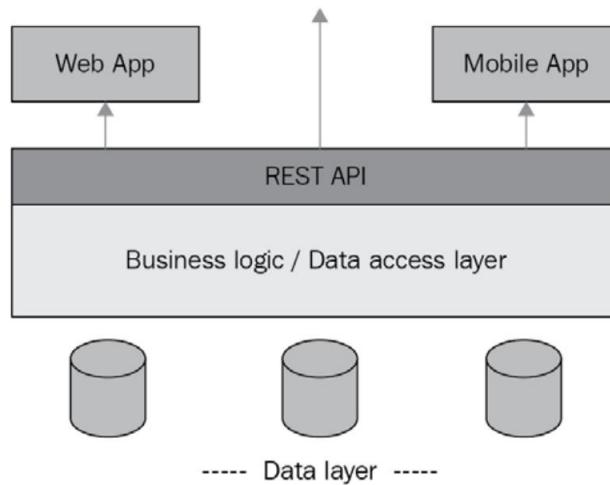


Figura 10.2 – Arquitectura de aplicaciones web/móviles

En nuestro caso de estudio, desarrollaremos una aplicación web sobre la aplicación REST API que se desarrolló para el objeto del modelo Student en el ejemplo de código anterior.

En un nivel alto, tendremos los componentes en nuestra aplicación como se muestra aquí:

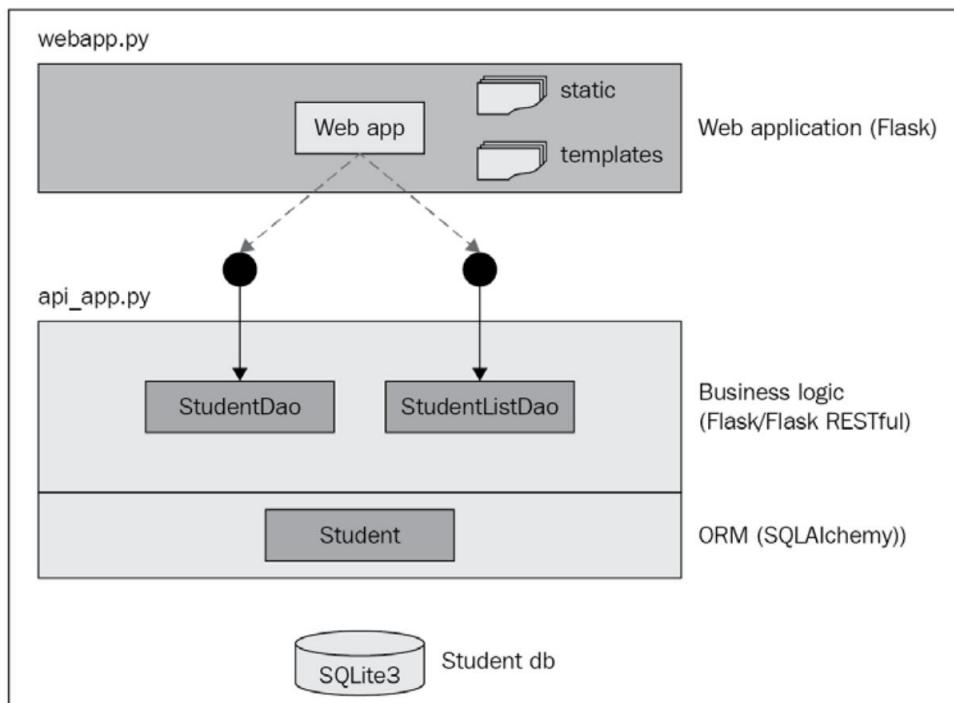


Figura 10.3: aplicación web de muestra con una API REST como motor de back-end

384 Uso de Python para desarrollo web y API REST

Ya desarrollamos una capa de lógica empresarial y una capa de acceso a datos (ORM) y expusimos la funcionalidad a través de dos puntos finales de API. Esto se trata en la sección Uso de Flask para una API REST. Desarrollaremos una parte de la aplicación web para el acceso web y consumiremos la API que ofrece la lógica comercial.

La aplicación web webapp.py estará basada en Flask. La aplicación webapp.py (denominada webapp en adelante) será independiente de api_app.py aplicación (denominada apiapp en el futuro) en el sentido de que las dos aplicaciones se ejecutarán por separado como dos instancias de Flask, idealmente en dos máquinas separadas.

Pero si estamos ejecutando las dos instancias de Flask en la misma máquina con fines de prueba, debemos usar diferentes puertos y usar la dirección IP de la máquina local como host. Flask usa la dirección 127.0.0.1 como host para ejecutar su servidor web integrado, que puede no estar permitido para ejecutar dos instancias. Las dos aplicaciones se comunicarán entre sí solo a través de una API REST.

Además, desarrollaremos algunas plantillas de Jinja para enviar solicitudes de operaciones de creación, actualización y eliminación. Reutilizaremos el código de la aplicación api_py.py tal como está, pero desarrollaremos la aplicación webapp.py con funciones como listar estudiantes, agregar un nuevo estudiante, eliminar un estudiante y actualizar los datos de un estudiante. Agregaremos funciones de Python para cada característica una por una:

1. Comenzaremos con la inicialización de la instancia Flask como lo hemos hecho para

ejemplos de código anteriores. El código de ejemplo es el siguiente:

```
#webapp.py: interacción con la capa empresarial a través de la API REST
# para crear, eliminar y listar objetos
desde matraz importar Flask, render_template, redirigir, solicitar
solicitudes de importación, json

aplicación = Frasco (__nombre__)
```

2. A continuación, agregaremos una función de lista para manejar las solicitudes con / URL de la siguiente manera:

```
@aplicación.get('/')
lista definida():
    respuesta = solicitudes.get('http://localhost:8080
        /estudiantes')
    datos = json.loads(respuesta.texto)
    volver render_template('main.html', estudiantes=datos)
```

Estudio de caso: creación de una aplicación web con REST API 385

En todas nuestras funciones de Python, usamos la biblioteca de solicitudes para enviar una solicitud de API REST a la aplicación apiapp que está alojada en la misma máquina en nuestro entorno de prueba.

3. A continuación, implementaremos una función de agregar para procesar la solicitud de agregar un nuevo estudiante a la base de datos. Solo la solicitud con el tipo de método POST se asigna a esta función de Python. El código de ejemplo es el siguiente:

```
@aplicación.post('/')
definitivamente añadir():

fname = solicitud.formulario['fname']
nombre = solicitud.formulario['nombre']
calificación = solicitud.formulario['calificación']
carga útil = {'nombre': f'{fnombre} {lnombre}', 'grado': grado}

respuesta = solicitudes.post('http://localhost:8080
/estudiantes', datos=carga útil)
devolver redirección("/")
```

Tenga en cuenta que para la llamada API posterior a nuestra aplicación apiapp , construimos la carga útil objeto y lo pasó como un atributo de datos al método POST de las solicitudes módulo.

4. A continuación, agregaremos una función DELETE para manejar una solicitud para eliminar un archivo existente . alumno. Se espera que el tipo de solicitud asignado a este método proporcione la identificación del estudiante como parte de la URL.

```
@app.get('/eliminar/<int:id>')
def eliminar (id):

respuesta = solicitudes.eliminar('http://localhost:8080
/estudiantes/' +str(id))
devolver redirección("/")
```

386 Uso de Python para desarrollo web y API REST

5. A continuación, agregaremos dos funciones para manejar la función de actualización. Una función (`actualizar`) se utiliza para actualizar los datos de un estudiante de la misma manera que la publicación la función lo hace. Pero antes de activar la función de actualización , nuestra aplicación webapp ofrecerá un formulario al usuario con los datos actuales de un objeto de estudiante . La segunda función (`load_student_for_update`) obtendrá un objeto de estudiante y lo enviará a una plantilla de Jinja para que los usuarios lo editen. El código para ambas funciones es el siguiente:

```
@app.post('/actualizar/<int:id>')
    actualización de definición (id):
        fname = solicitud.formulario['fname']
        nombre = solicitud.formulario['nombre']
        calificación = solicitud.formulario['calificación']
        carga útil = {'nombre' : f'{fnombre} {Inombre}', 'calificación':calificación}
        respuesta = solicitudes.put('http://localhost:8080
        /estudiantes/' + str(id), datos = carga útil)
        devolver redirección("/")
    
@app.get('/actualizar/<int:id>')
def load_student_for_update(id):
    respuesta = solicitudes.get('http://localhost:8080
    /estudiantes/' +str(id))
    estudiante = json.loads(respuesta.texto)
    fname = estudiante['nombre'].split()[0]
    lname = estudiante['nombre'].split()[1]
    return render_template('update.html', fname=fname, lname=lname, estudiante=
    estudiante)
```

El código dentro de estas funciones no es diferente de lo que ya discutimos en relación con los ejemplos anteriores. Por lo tanto, no entraremos en los detalles de cada línea del código, pero destacaremos los puntos clave de esta aplicación web y su interacción con una aplicación REST API:

- Para nuestra aplicación web, estamos usando dos plantillas Jinja (`main.html` y `actualizar.html`). También estamos usando una plantilla (la llamamos `base.html`) que es común a ambas plantillas. La plantilla `base.html` se crea principalmente con el marco de interfaz de usuario de arranque. No discutiremos los detalles de las plantillas Jinja y el bootstrap, pero lo animamos a que se familiarice con ambos utilizando las referencias proporcionadas al final de este capítulo. Las plantillas Jinja de muestra con código de arranque están disponibles con el código fuente de este capítulo.

- La raíz / URL de nuestra aplicación web abrirá la página principal (main.html), que nos permite agregar un nuevo estudiante y también proporciona una lista de los estudiantes existentes. La siguiente captura de pantalla muestra la captura de pantalla de la página principal, que se representará con nuestra plantilla main.html :

The screenshot shows two parts of a web application. The top part is a modal window titled 'Add a Student' with fields for 'First name' and 'Last name'. The bottom part is a table titled 'Students' listing two entries: 'John Lee' (Grade 10) and 'Brian Miles' (Grade 7), each with 'Update' and 'Delete' buttons.

No	Name	Grade	Actions
1	John Lee	10	<button>Update</button> <button>Delete</button>
2	Brian Miles	7	<button>Update</button> <button>Delete</button>

Figura 10.4 – Página principal de la aplicación webapp

- Si agregamos el nombre y apellido de un estudiante con una calificación y hacemos clic en **Enviar** botón, esto activará una solicitud POST con datos de estos tres campos de entrada. Nuestra aplicación webapp delegará esta solicitud a la función de agregar . La función de agregar utilizará la API REST correspondiente de la aplicación apiapp para agregar un nuevo estudiante y la función de agregar volverá a mostrar la página principal con una lista actualizada de estudiantes (incluidos los nuevos estudiantes).
- En la página principal de la aplicación web (main.html), agregamos dos botones (**Actualizar** y **Eliminar**) con cada registro de estudiante. Al hacer clic en el botón **Eliminar** , el navegador activará una solicitud GET con la URL /delete/<id> . Esta solicitud se delegará a la función de eliminación . La función de eliminación utilizará la API REST de la aplicación apiapp para eliminar al estudiante de la base de datos SQLite3 y volverá a mostrar la página principal con una lista actualizada de estudiantes.

388 Uso de Python para desarrollo web y API REST

- Al hacer clic en el botón **Actualizar**, el navegador activará una solicitud GET con /actualizar/<id> URL. Esta solicitud se delegará a load_student_for_ función de actualización . Esta función primero cargará los datos de los estudiantes utilizando la API REST de la aplicación apiapp , establecerá los datos en la respuesta y generará la actualización. plantilla html La plantilla update.html le mostrará al usuario un formulario HTML lleno de datos del estudiante para permitir la edición. El formulario que desarrollamos para el escenario de actualización se muestra aquí:

The screenshot shows a web application interface for updating student information. At the top, there's a header with the word "Students" and a menu icon. Below the header, the title "Update Student" is centered. There are three input fields: "First name" containing "John", "Last name" containing "Lee", and "Grade" containing "10". At the bottom of the form is a prominent "Update" button.

Figura 10.5: un formulario de muestra para

actualizar a un estudiante. Después de los cambios, si un usuario envía el formulario haciendo clic en el botón **Actualizar**, el navegador activará una solicitud POST con la URL /update/<id>. Hemos registrado la función de actualización para esta solicitud. La función de actualización extraerá datos de la solicitud y los pasará a la API REST de la aplicación apiapp . Una vez que se actualice la información del estudiante, volveremos a mostrar la página main.html con una lista actualizada de los estudiantes.

En este capítulo, hemos omitido los detalles de las tecnologías web puras, como HTML, Jinja, CSS y los marcos de interfaz de usuario en general. La belleza de los marcos web es que permiten usar cualquier tecnología web para las interfaces de los clientes, especialmente si estamos construyendo nuestras aplicaciones usando una API REST.

Esto concluye nuestra discusión sobre la creación de aplicaciones web y el desarrollo de una API REST utilizando Flask y sus extensiones. El desarrollo web no se limita a un lenguaje o un marco. Los principios básicos y la arquitectura siguen siendo los mismos en todos los marcos y lenguajes web. Los principios de desarrollo web que aprendió aquí lo ayudarán a comprender cualquier otro marco web para Python o cualquier otro lenguaje.

Resumen

En este capítulo, discutimos cómo usar Python y marcos web como Flask para desarrollar aplicaciones web y una API REST. Comenzamos el capítulo analizando los requisitos para el desarrollo web, que incluyen un marco web, un marco de interfaz de usuario, un servidor web, un sistema de base de datos, compatibilidad con API, seguridad y documentación. Más tarde, presentamos cómo usar el marco Flask para crear aplicaciones web con varios ejemplos de código. Discutimos diferentes tipos de solicitudes con diferentes métodos HTTP y cómo analizar los datos de la solicitud con ejemplos de código relevantes. También aprendimos sobre el uso de Flask para interactuar con un sistema de base de datos usando una biblioteca ORM como SQLAlchemy. En la última parte del capítulo, presentamos el rol de una API web para aplicaciones web, aplicaciones móviles y aplicaciones de empresa a empresa.

Investigamos una extensión de Flask para desarrollar una API REST con un análisis detallado mediante el uso de una aplicación de API de muestra. En la última sección, discutimos un estudio de caso del desarrollo de una aplicación web para estudiantes. La aplicación web se crea utilizando dos aplicaciones independientes, ambas ejecutándose como aplicaciones Flask. Una aplicación ofrece una API REST para la capa de lógica empresarial sobre un sistema de base de datos. La otra aplicación proporciona una interfaz web a los usuarios y consume la interfaz REST API de la primera aplicación para brindar acceso a los objetos de recursos de los estudiantes.

Este capítulo proporciona amplios conocimientos prácticos sobre la creación de aplicaciones web y una API REST con Flask. Los ejemplos de código incluidos en este capítulo le permitirán comenzar a crear aplicaciones web y escribir una API REST. Este conocimiento es fundamental para cualquier persona que busque una carrera en desarrollo web y trabaje en la creación de una API REST.

En el próximo capítulo, exploraremos cómo usar Python para desarrollar microservicios, un nuevo paradigma de desarrollo de software.

390 Uso de Python para desarrollo web y API REST

Preguntas

1. ¿Cuál es el propósito de TLS?
2. ¿Cuándo Flask es una opción superior al framework Django?
3. ¿Cuáles son los métodos HTTP más utilizados?
4. ¿Qué es CRUD y cómo se relaciona con una API REST?
5. ¿Una API REST solo usa JSON como formato de datos?

Otras lecturas

- Desarrollo Web Flask, por Miguel Grinberg
- Guía avanzada para la programación de Python 3, por John Hunt
- API REST con Flask y Python, por Jason Myers y Rick Copeland
- Essential SQLAlchemy, 2.^a edición, por José Salvatierra
- Inicio rápido de Bootstrap 4, por Jacob Lett
- Documentación en línea de Jinja, disponible en <https://jinja.palletsprojects.es/>

respuestas

1. El objetivo principal de TLS es proporcionar cifrado de datos que se intercambian entre los dos sistemas en internet.
2. Flask es una mejor opción para aplicaciones de tamaño pequeño a mediano y especialmente cuando se espera que los requisitos del proyecto cambien con frecuencia.
3. OBTENER y PUBLICAR.
4. **CRUD** significa operaciones de **creación, lectura, actualización y eliminación** en software desarrollo. Desde la perspectiva de la API, cada operación CRUD se asigna a uno de los métodos HTTP (GET, POST, PUT y DELETE).
5. Una API REST puede admitir cualquier formato basado en cadenas, como JSON, XML o HTML.
La compatibilidad con el formato de datos está más relacionada con la capacidad de HTTP para transportar los datos como parte del elemento del cuerpo HTTP.

11

Usando Python para microservicios Desarrollo

Las aplicaciones monolíticas que se construyen como un software de un solo nivel han sido una opción popular para desarrollar aplicaciones durante muchos años. Sin embargo, no es eficiente implementar aplicaciones monolíticas en plataformas en la nube en términos de reserva y utilización de recursos. Esto es cierto incluso para la implementación de aplicaciones monolíticas a gran escala en máquinas físicas. Los costes de mantenimiento y desarrollo de este tipo de aplicaciones son siempre elevados. Las aplicaciones de varios niveles han resuelto este problema hasta cierto punto para las aplicaciones web al dividir las aplicaciones en varios niveles.

Para cumplir con las demandas dinámicas de recursos y reducir los costos de desarrollo/mantenimiento, el verdadero salvador es una **arquitectura de microservicios**. Esta nueva arquitectura fomenta que las aplicaciones se construyan sobre servicios débilmente acoplados y se implementen en plataformas dinámicamente escalables, como contenedores. Organizaciones como Amazon, Netflix y Facebook ya han pasado de un modelo monolítico a una arquitectura basada en microservicios. Sin este cambio, estas organizaciones no podrían haber atendido a una gran cantidad de clientes.

392 Uso de Python para el desarrollo de microservicios

Cubriremos los siguientes temas en este capítulo:

- Presentación de microservicios
- Aprendizaje de mejores prácticas para microservicios
- Creación de aplicaciones basadas en microservicios

Después de completar este capítulo, aprenderá acerca de los microservicios y podrá crear aplicaciones basadas en microservicios.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- Biblioteca Python Flask con extensiones RESTful instaladas sobre Python 3.7 o liberación posterior.
- Python Django con la biblioteca Django Rest Framework sobre Python 3.7 o posterior liberación.
- Debe tener una cuenta de registro de Docker e instalar Docker Engine y Docker Compose en su máquina.
- Para implementar un microservicio en GCP Cloud Run, necesitará una cuenta de GCP (una prueba gratuita funcionará bien).

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter11>.

Comenzaremos nuestra discusión con la introducción de microservicios.

Introduciendo microservicios

Un microservicio es una entidad de software independiente que debe tener las siguientes características:

- **Combinado libremente** con otros servicios y sin ninguna dependencia de otro software componentes
- **Fácil de desarrollar y mantener** por un pequeño equipo sin depender de otros equipos

- **Instalable de forma independiente** como una entidad separada, preferiblemente en un contenedor
- Ofrece interfaces **fáciles de consumir** mediante protocolos sincrónicos como API REST o protocolos asincrónicos como **Kafka** o **RabbitMQ**.

Las palabras clave en términos de software que se denomina microservicio se implementan de forma independiente, se acoplan libremente y se pueden mantener fácilmente. Cada microservicio puede tener sus propios servidores de base de datos para evitar compartir recursos con otros microservicios. Esto asegurará la eliminación de dependencias entre los microservicios.

El estilo de arquitectura de microservicios es un paradigma de desarrollo de software para desarrollar aplicaciones utilizando puramente microservicios. Esta arquitectura incluso incluye la entidad de interfaz principal de la aplicación, como una aplicación web. A continuación se ilustra un ejemplo de una aplicación basada en microservicios:

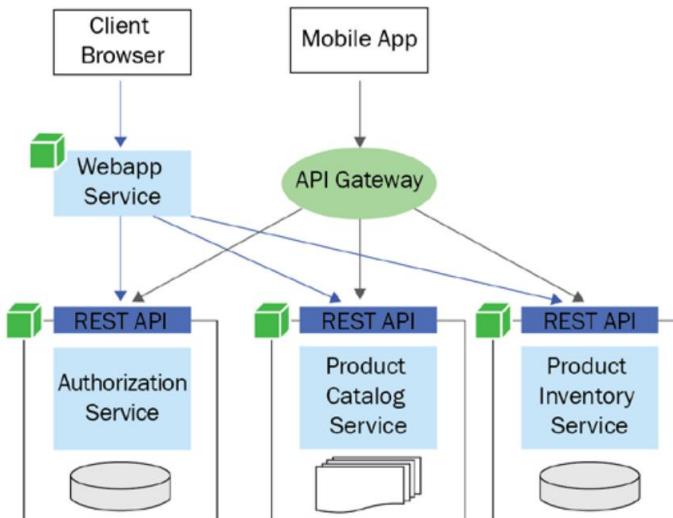


Figura 11.1: una aplicación de estilo de arquitectura de microservicios de muestra

En esta aplicación de ejemplo, tenemos microservicios individuales como el Servicio de **autorización**, el **Servicio de catálogo de productos** y el **Servicio de inventario de productos**. Creamos una aplicación web, también como un microservicio, que utiliza los tres microservicios individuales a través de la API REST. Para clientes móviles, se puede crear una aplicación móvil utilizando los mismos microservicios individuales a través de una puerta de enlace API. Podemos ver una ventaja inmediata de la arquitectura de microservicios, que es la reutilización. Algunas otras ventajas de usar microservicios son las siguientes:

- Somos flexibles cuando se trata de seleccionar cualquier tecnología y cualquier lenguaje de programación que se adapte a cualquier requisito de microservicio individual. Incluso podemos reutilizar el código heredado escrito en cualquier idioma si podemos exponerlo mediante una interfaz API.

394 Uso de Python para el desarrollo de microservicios

- Podemos desarrollar, probar y mantener microservicios individuales por pequeños equipos independientes. Es crucial contar con pequeños equipos independientes y autónomos para el desarrollo de aplicaciones a gran escala.
- Uno de los desafíos con las aplicaciones monolíticas es la gestión de versiones de biblioteca en conflicto, que nos vemos obligados a incluir debido a las diferentes funciones agrupadas en una sola aplicación. Con los microservicios se minimizan las posibilidades de conflicto en cuanto a las versiones de estas bibliotecas.
- Podemos implementar y parchear los microservicios individuales de forma independiente. Esto nos permite utilizar **la integración continua/entrega continua (CI/CD)** para aplicaciones complejas. Esto también es importante cuando necesitamos aplicar un parche o actualizar una característica de una aplicación. Para aplicaciones monolíticas, volveremos a implementar toda la aplicación, lo que significa que hay posibilidades de romper otras partes de la aplicación.
Con los microservicios, solo se volverán a implementar uno o dos servicios sin riesgos de romper nada más en otros microservicios.
- Podemos aislar fallas y errores a nivel de microservicio en lugar de a nivel de aplicación. Si hay una falla o falla con un servicio, podemos depurarlo, arreglarlo y parchearlo o detenerlo para mantenimiento sin afectar el resto de la funcionalidad de la aplicación. En el caso de aplicaciones monolíticas, un problema en un componente puede provocar la caída de toda la aplicación.

A pesar de varias ventajas, existen algunas desventajas asociadas con el uso del estilo de arquitectura de microservicios:

- El primero es la mayor complejidad de crear aplicaciones basadas en microservicios. La complejidad surge principalmente del hecho de que cada microservicio tiene que exponer una API y el servicio o programa del consumidor tiene que interactuar con los microservicios usando una API. La seguridad por microservicio es otro contribuyente a la complejidad.
- La segunda desventaja es el aumento de los requisitos de recursos en comparación con las aplicaciones monolíticas. Cada microservicio requiere que la memoria adicional se aloje de forma independiente en un contenedor o una máquina virtual, incluso si se trata de una **máquina virtual Java (JVM)**.
- La tercera desventaja es que se requieren esfuerzos adicionales para depurar y solucionar un problema en diferentes microservicios que pueden implementarse en contenedores o sistemas separados.

A continuación, estudiaremos las mejores prácticas para crear microservicios.

Aprendizaje de mejores prácticas para microservicios

Al iniciar una nueva aplicación, la primera y más importante pregunta que debemos hacernos es si la arquitectura de microservicios es adecuada. Esto comienza con un análisis de los requisitos de la aplicación y la capacidad de dividir los requisitos en componentes separados e individuales. Si ve que sus componentes con frecuencia dependen unos de otros, esto es un indicador de que la segregación de los componentes puede requerir una reelaboración, o que esta aplicación puede no ser adecuada para la arquitectura de microservicios.

Es importante tomar esta decisión de usar o no microservicios en la fase inicial de una aplicación. Hay una escuela de pensamiento que dice que es mejor comenzar a construir una aplicación utilizando una arquitectura monolítica para evitar los costos adicionales de los microservicios al principio. Sin embargo, este no es un enfoque aconsejable. Una vez que hemos construido una aplicación monolítica, es difícil transformarla en una arquitectura de microservicios, especialmente si la aplicación ya está en producción. Empresas como Amazon y Netflix lo han hecho, pero lo hicieron como parte de su evolución tecnológica y ciertamente, cuentan con los recursos humanos y tecnológicos disponibles para acometer esta transformación.

Una vez que hayamos tomado la decisión de crear nuestra próxima aplicación utilizando microservicios, las siguientes mejores prácticas lo guiarán en la toma de decisiones de diseño e implementación:

- **Independientes y débilmente acoplados:** Estos requisitos son parte del definición de microservicios. Cada microservicio debe construirse independientemente de los otros microservicios y debe estar acoplado de la manera más flexible posible.
- **Diseño basado en dominios (DDD):** el propósito de la arquitectura de microservicios no es tener tantos microservicios pequeños como sea posible. Siempre debemos recordar que cada microservicio tiene sus costos generales. Deberíamos construir tantos microservicios como requiera el negocio o el dominio. Recomendamos considerar DDD, que fue presentado por Eric Evans en 2004.

Si tratamos de aplicar DDD a los microservicios, sugiere tener primero un diseño estratégico para definir diferentes contextos combinando dominios comerciales relacionados y sus subdominios. El diseño estratégico puede ser seguido por un diseño táctico que se enfoca en desglosar los dominios centrales en bloques de construcción y entidades de grano fino. Este desglose proporcionará pautas claras para asignar los requisitos a posibles microservicios.

- **Interfaces de comunicación:** debemos usar interfaces de microservicio bien definidas, preferiblemente una API REST o una API basada en eventos, para la comunicación. Los microservicios deben evitar llamarse entre sí directamente.

396 Uso de Python para el desarrollo de microservicios

- **Utilice una puerta de enlace API:** los microservicios y sus aplicaciones de consumo deben interactuar con microservicios individuales mediante una puerta de enlace API. La puerta de enlace API puede ocuparse de los aspectos de seguridad, como la autenticación y el equilibrio de carga, de forma inmediata. Además, con una nueva versión de un microservicio, podemos usar la puerta de enlace API para redirigir las solicitudes de los clientes a la versión más nueva sin afectar el software del lado del cliente.
- **Límite la pila de tecnología:** aunque la arquitectura de microservicios permite el uso de cualquier lenguaje de programación y marco por servicio, no es recomendable desarrollar microservicios usando diferentes tecnologías en ausencia de razones comerciales o de reutilización. Una pila de tecnología diversa puede ser atractiva por razones académicas, pero traerá complejidad operativa en el mantenimiento y la resolución de problemas de la aplicación.
- **Modelo de implementación:** No es obligatorio implementar un microservicio en un contenedor, pero es recomendable. Los contenedores traen muchas características integradas, como implementación automatizada, soporte multiplataforma e interoperabilidad. Además, mediante el uso de contenedores, podemos asignar los recursos al servicio según sus requisitos y garantizar una distribución justa de los recursos entre los diferentes microservicios.
- **Control de versiones:** deberíamos usar un sistema de control de versiones separado para cada microservicio.
- **Organización del equipo:** la arquitectura de microservicios brinda la oportunidad de tener equipos dedicados por microservicio. Debemos tener en cuenta este principio al organizar equipos para un proyecto a gran escala. El tamaño de un equipo debe basarse en la filosofía de las dos pizzas, que establece que debemos formar un equipo con tantos ingenieros que puedan alimentarse con dos pizzas grandes. Un equipo puede poseer uno o más microservicios en función de su complejidad.
- **Registro/supervisión centralizados:** como se mencionó anteriormente, la solución de problemas Los problemas en una aplicación de estilo de arquitectura de microservicio pueden llevar mucho tiempo, especialmente si los microservicios se ejecutan en contenedores. Deberíamos usar herramientas profesionales o de código abierto para monitorear y solucionar problemas de los microservicios para reducir dichos costos operativos. Algunos ejemplos de tales herramientas son **Splunk, Grafana, Elk y App Dynamics**.

Ahora que hemos cubierto la introducción y las mejores prácticas de los microservicios, a continuación, profundizaremos en el aprendizaje para crear una aplicación utilizando microservicios.

Creación de aplicaciones basadas en microservicios

Antes de entrar en los detalles de implementación de los microservicios, es importante analizar varios marcos de microservicios y opciones de implementación. Comenzaremos con un marco de microservicios disponible en Python.

Aprender opciones de desarrollo de microservicios en Python

En Python, tenemos una gran cantidad de marcos y bibliotecas disponibles para el desarrollo de microservicios. No podemos enumerar todas las opciones disponibles, pero vale la pena destacar las más populares y las que tienen algunos conjuntos de características diferentes. Estas opciones se describen a continuación:

- **Flask:** se trata de un marco ligero que se puede utilizar para crear microservicios basados en **Web Service Gateway Interface (WSGI)**. Tenga en cuenta que WSGI se basa en un patrón de diseño síncrono de solicitud y respuesta. Ya usamos Flask y su extensión RESTful para crear una aplicación de API REST en el Capítulo 10, Uso de Python para desarrollo web y API REST. Dado que Flask es un marco de desarrollo de API y web popular, es una opción de fácil adopción para muchos desarrolladores que ya usan Flask.
- **Django:** Django es otro marco web popular con una gran comunidad de desarrolladores. Con **Django Rest Framework (DRF)**, podemos construir microservicios con interfaces REST API. Django ofrece microservicios basados en WSGI y **Asynchronous Service Gateway Interface (ASGI)**. ASGI se considera un sucesor de la interfaz WSGI. ASGI es una excelente opción si está interesado en desarrollar su aplicación basada en **Asyncio**, un tema que tratamos en detalle en el Capítulo 7, Multiprocesamiento, Multihilo y Programación asíncrona.
- **Falcon:** este es también un marco web popular después de Flask y Django. No viene con un servidor web incorporado, pero está bien optimizado para microservicios. Al igual que Django, es compatible con ASGI y WSGI.
- **Nameko:** este marco está diseñado específicamente para el desarrollo de microservicios en Python y no es un marco de aplicación web. Nameko viene con soporte integrado para **llamadas a procedimientos remotos (RPC)**, eventos asíncronos y RPC basados en WebSocket. Si su aplicación requiere alguna de estas interfaces de comunicación, debería considerar Nameko.

398 Uso de Python para el desarrollo de microservicios

- **Bottle:** este es un marco de microservicios súper ligero basado en WSGI. Todo el marco se basa en un solo archivo y aprovecha solo una biblioteca estándar de Python para sus operaciones.
- **Tornado:** Se basa en E/S de red sin bloqueo. Tornado puede manejar mucho tráfico con gastos generales extremadamente bajos. Esta también es una opción adecuada para conexiones basadas en WebSocket y de sondeo largo.

Para el desarrollo de nuestros microservicios de muestra, podemos usar cualquiera de los marcos mencionados anteriormente. Pero usaremos Flask y Django por dos razones. Primero, estos dos son los más populares para desarrollar aplicaciones web y microservicios. En segundo lugar, reutilizaremos una aplicación API de ejemplo que desarrollamos en el capítulo anterior. Se desarrollará un nuevo microservicio usando Django e ilustrará cómo usar Django para el desarrollo web y API.

A continuación, analizaremos las opciones de implementación de microservicios.

Presentación de opciones de implementación para microservicios

Una vez que escribimos microservicios, la siguiente pregunta es cómo implementarlos como una entidad aislada e independiente. En aras de la discusión, supondremos que los microservicios se construyen con interfaces HTTP/REST. Podemos implementar todos los microservicios en el mismo servidor web como diferentes aplicaciones web o alojar un servidor web para un microservicio. Un microservicio en un servidor web separado se puede implementar en una sola máquina (física o virtual) o en máquinas separadas o incluso en contenedores separados. Hemos resumido todos estos diferentes modelos de implementación en el siguiente diagrama:

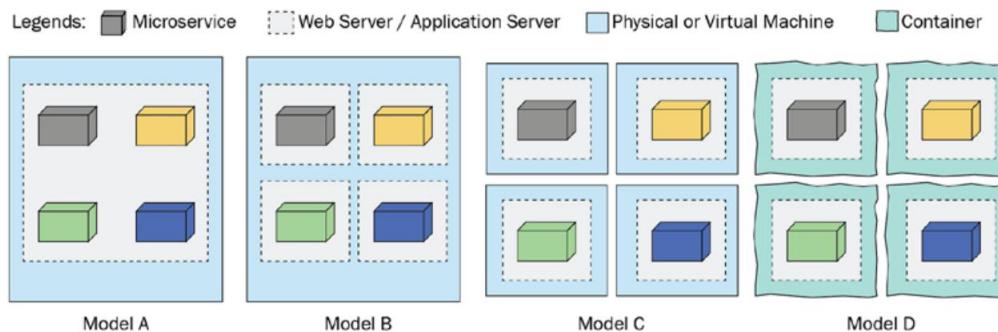


Figura 11.2 – Modelos de implementación de microservicios

Los cuatro modelos de implementación que se muestran en la Figura 11.2 se describen a continuación:

- **Modelo A:** en este modelo, implementamos cuatro microservicios diferentes en el mismo servidor web. Existe una buena posibilidad de que los microservicios en este caso estén compartiendo bibliotecas, estando en un solo servidor web. Esto puede generar conflictos de biblioteca y no es un modelo recomendado para implementar microservicios.
- **Modelo B:** para este modelo, implementamos cuatro microservicios en una sola máquina, pero usamos un microservicio por servidor web para hacerlos independientes. Este modelo está bien para entornos de desarrollo, pero puede no ser adecuado a escala de producción.
- **Modelo C:** este modelo utiliza cuatro máquinas virtuales para alojar cuatro microservicios. Cada máquina aloja solo un microservicio con un servidor web. Este modelo es adecuado para la producción si no es posible usar contenedores. La principal advertencia de este modelo son los costos adicionales debido a los gastos generales de recursos que cada la máquina virtual traerá consigo.
- **Modelo D:** en este modelo, implementamos cada microservicio como un contenedor en una sola máquina o en varias máquinas. Esto no solo es rentable, sino que también proporciona una manera fácil de cumplir con las especificaciones de los microservicios. Este es el modelo recomendado siempre que sea factible su uso.

Analizamos diferentes modelos de implementación para comprender qué opción es más apropiada que otras. Para el desarrollo de nuestra aplicación basada en microservicios de muestra, utilizaremos una combinación de un microservicio basado en contenedores y un microservicio alojado solo en un servidor web. Este modelo mixto ilustra que técnicamente podemos usar cualquier opción, aunque se recomienda la implementación basada en contenedores. Posteriormente, llevaremos uno de nuestros microservicios a la nube para demostrar la portabilidad de los microservicios.

Después de analizar las opciones de desarrollo e implementación de microservicios, es hora de comenzar a crear una aplicación utilizando dos microservicios en la siguiente sección.

Desarrollo de una aplicación basada en microservicios de muestra

Para la aplicación de muestra, desarrollaremos dos microservicios y una aplicación web utilizando los marcos Flask y Django. Nuestra aplicación de muestra será una extensión de **Student** aplicación web que se desarrolló como caso de estudio en el capítulo anterior. La arquitectura de la aplicación aparecerá como se muestra aquí:

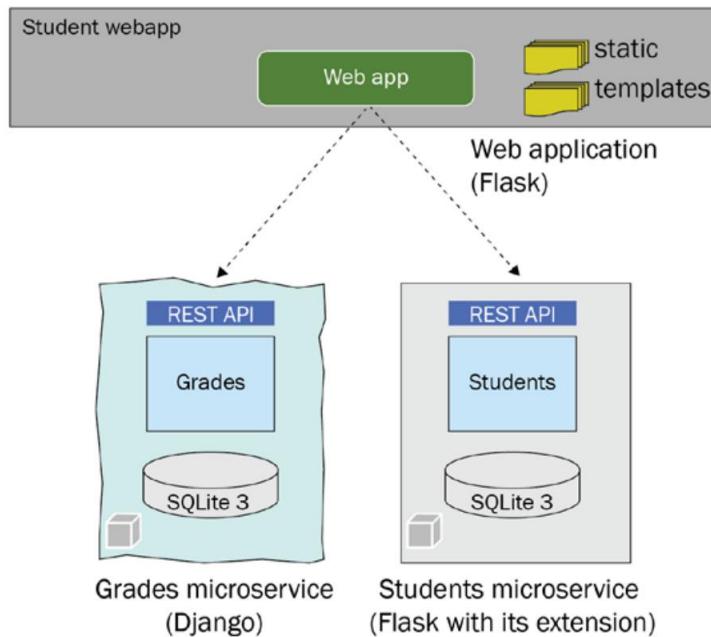


Figura 11.3 – Arquitectura basada en microservicios C de una aplicación de muestra

Para desarrollar esta aplicación de muestra, desarrollaremos los siguientes componentes:

- Cree un nuevo microservicio **Grades** utilizando el marco Django e impleméntelo con un **Docker Engine**. Docker Engine es un software de código abierto para contener nuestra aplicación. El microservicio Grades proporcionará información adicional sobre cada grado, como el nombre del edificio y el nombre del profesor de la clase, y estos atributos no se almacenan con el modelo Student en el microservicio Students.
- Reutilizar la aplicación apiapp del capítulo anterior. Se nombra como un Microservicio para estudiantes para esta aplicación de muestra. No habrá cambios en el código de esta aplicación/módulo.
- Actualizaremos la aplicación webapp del estudio de caso del capítulo anterior para consumir el microservicio Grades y agregue atributos Grade adicionales con cada objeto Student . Esto también requerirá actualizaciones menores a las plantillas de Jinja.

Comenzaremos construyendo el microservicio Grades con Django.

Creación de un microservicio de calificaciones

Para desarrollar un microservicio usando Django, usaremos **Django Rest Framework (DRF)**.

Django usa varios componentes de su marco web para construir una API REST y microservicios. Por lo tanto, este ejercicio de desarrollo también le dará una idea de alto nivel sobre el desarrollo de aplicaciones web con Django.

Dado que comenzamos con Flask y ya estamos familiarizados con los conceptos web básicos del desarrollo web, será una transición conveniente para nosotros comenzar a usar Django. Ahora entendamos los pasos involucrados:

1. Lo primero es lo primero, crearemos un directorio de proyectos o crearemos un nuevo proyecto en nuestro IDE favorito con un entorno virtual. Si no está utilizando un IDE, puede crear y activar un entorno virtual en el directorio de su proyecto utilizando los siguientes comandos:

```
python -m venv myenv  
fuente myenv/bin/activar
```

Para cualquier aplicación web, es vital crear un entorno virtual para cada aplicación. El uso de un entorno global para las dependencias de la biblioteca puede generar errores difíciles de solucionar.

2. Para construir una aplicación Django, necesitaremos al menos dos bibliotecas que puedan ser instalado usando los siguientes comandos pip :

```
pip instalar django  
pip install django-rest-framework
```

3. Una vez que hayamos instalado Django, podemos usar la utilidad de línea de comandos django-admin para crear un proyecto Django. El comando que se muestra a continuación creará un proyecto de calificaciones de Django para nuestro microservicio:

```
django-admin startproject calificaciones
```

402 Uso de Python para el desarrollo de microservicios

Este comando creará una aplicación web de administración en el directorio de calificaciones y agregará un archivo manage.py a nuestro proyecto. La aplicación web de administración incluye secuencias de comandos de inicio de servidor web integradas, un archivo de configuración y un archivo de enrutamiento de URL. manage.py también es una utilidad de línea de comandos, como django-admin, y ofrece características similares pero en el contexto de un proyecto de Django. La estructura de archivos del directorio del proyecto se verá de la siguiente manera cuando creamos un nuevo proyecto Django:

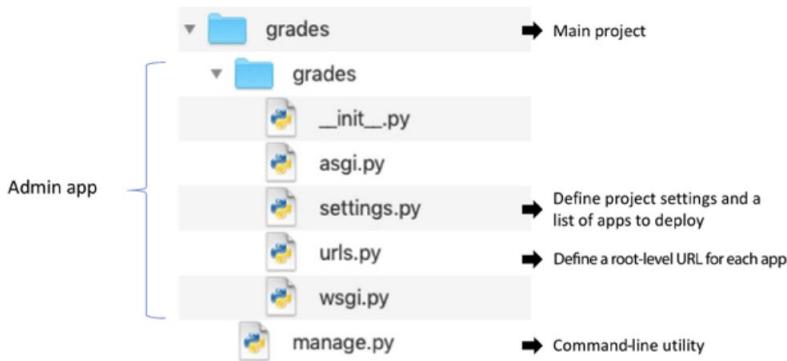


Figura 11.4: estructura de archivos de un nuevo proyecto de

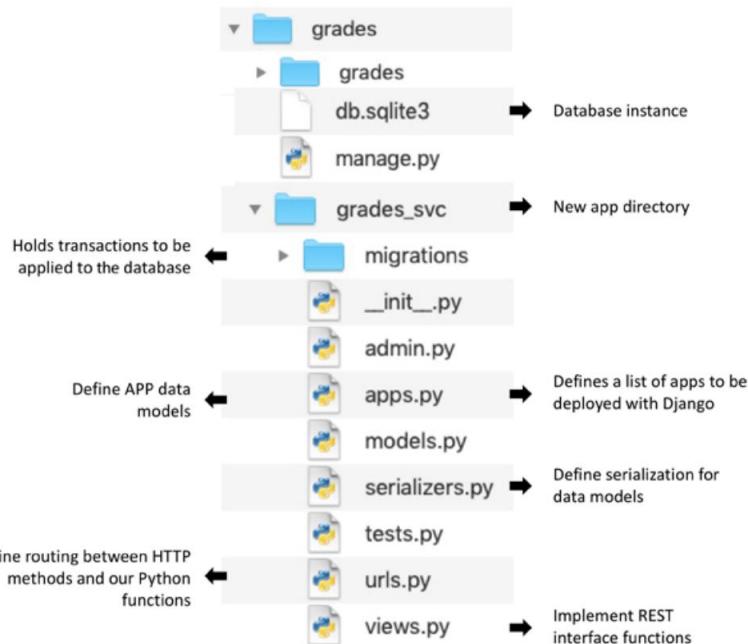
Django Como se muestra en la Figura 11.4, el archivo settings.py contiene una configuración a nivel de proyecto que incluye una lista de aplicaciones para implementar con el servidor web. El archivo urls.py contiene información de enrutamiento para diferentes aplicaciones implementadas. Actualmente, solo la aplicación de administración está incluida en este archivo. asgi.py y wsgi.py están disponibles para iniciar el servidor web ASGI o WSGI, y la opción de cuál usar se establece en el archivo settings.py .

4. El siguiente paso es crear una nueva aplicación Django (nuestro microservicio Grades) usando el siguiente comando en el directorio principal del proyecto grades :

```
python3 manage.py startapp grades_svc
```

Este comando creará una nueva aplicación (con componentes web) en un directorio separado con el mismo nombre que le dimos a este comando, grades_svc. Esto también creará una instancia de base de datos SQLite3 predeterminada . La opción de usar la base de datos SQLite3 por defecto está disponible en el archivo settings.py , pero se puede cambiar si decidimos usar cualquier otra base de datos.

5. Además de los archivos creados automáticamente en el directorio grades_svc , agregaremos dos archivos más: urls.py y serializers.py. En la Figura 11.5 se muestra una estructura completa de directorios del proyecto con dos archivos adicionales. Las funciones de los diferentes archivos relevantes para nuestro proyecto también se elaboran en este diagrama:

Figura 11.5 – Estructura completa de directorios con la aplicación `grades_svc`

6. A continuación, agregaremos el código necesario para nuestro microservicio en estos archivos uno por uno.

Comenzaremos definiendo nuestra clase de modelo Grade extendiendo la clase Model del paquete de modelos de base de datos de Django . El código completo del archivo `models.py` es el siguiente:

```
desde modelos de importación django.db
Grado de clase (modelos.Modelo):
grade_id = modelos.CharField(max_length=20)
edificio = modelos.CharField(max_length=200)
profesor = modelos.CharField(max_length=200)

def __str__(uno mismo):
    devolver self.grade_id
```

7. Para que nuestro modelo sea visible en el panel de la aplicación de administración , debemos registrar nuestra clase de calificación modelo en el archivo `admin.py` de la siguiente manera:

```
desde django.contrib.administrador de importación
de .models importar Grado
admin.sitio.registrar(Grado)
```

404 Uso de Python para el desarrollo de microservicios

8. A continuación, implementaremos un método para recuperar una lista de objetos Grade de la base de datos. Agregaremos una clase GradeViewSet extendiendo ViewSet en el archivo views.py de la siguiente manera:

```
desde rest_framework importar conjuntos de vistas, estado  
de rest_framework.response importar Respuesta  
de .models importar Grado  
de .serializers importar GradeSerializer  
  
clase GradeViewSet(viewsets.ViewSet):  
  
    lista de definición (auto, solicitud):  
        grades_list = Grade.objects.all()  
        serializador = GradeSerializer(grades_list, many=True)  
  
        respuesta de retorno (serializador.datos)  
    def crear (auto, solicitud):  
        aprobar:  
    def recuperar(auto, solicitud, id=Ninguno):  
        aprobar:
```

Tenga en cuenta que también agregamos métodos para agregar un nuevo objeto Grade y para obtener un objeto Grade según su ID en la implementación real para la integridad de nuestro microservicio. Solo mostramos el método de lista porque este es el único método relevante para nuestra aplicación de muestra. También es importante resaltar que los objetos de vista deben implementarse como clases y debemos evitar poner la lógica de la aplicación en los objetos de vista.

Una vez que implementemos nuestros métodos principales en la aplicación grades_svc , agregaremos nuestra aplicación al proyecto Django para su implementación y agregaremos rutas en la aplicación, así como en el nivel de la API:

1. Primero, agregaremos nuestra aplicación grades_svc y también el resto del marco a la lista de INSTALLED_APPS en el archivo settings.py de la siguiente manera:

```
APLICACIONES_INSTALADAS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',
```

```
'django.contrib.messages',
'django.contrib.staticfiles',
'grados_svc',
'marco_resto',
]
```

Un error común que cometan los desarrolladores es seguir agregando nuevos componentes a un solo archivo de configuración, lo cual es difícil de mantener para un proyecto grande. La mejor práctica es dividir el archivo en varios archivos y cargarlos en el archivo de configuración principal.

2. Esto también asegurará que nuestra aplicación esté visible en la aplicación de administración . El siguiente paso es agregar la configuración de URL en el nivel de la aplicación de administración y luego en el nivel de la aplicación. Primero, agregaremos la URL de nuestra aplicación en el archivo urls.py bajo el administrador

aplicación de la siguiente manera:

```
patrones de URL = [
    ruta('admin/', admin.sitio.urls),
    ruta("", include('calificaciones_svc.urls')),
]
```

En el archivo urls.py de la aplicación de administración, estamos redirigiendo cada solicitud a nuestro microservicio, excepto la que viene con la URL admin/ .

3. El siguiente paso es establecer rutas en nuestra aplicación basadas en diferentes métodos HTTP. Esto requiere que agreguemos el archivo urls.py a nuestro directorio grades_svc con las siguientes definiciones de ruta:

```
desde la ruta de importación django.urls
desde .views importar GradeViewSet

patrones de URL = [
    ruta(calificaciones '/', GradeViewSet.as_view({
        'get': 'list', #relevante para nuestra aplicación de muestra
        'publicar': 'crear'
    })),
    ruta('calificaciones/<str:id>', GradeViewSet.as_view({
        'obtener': 'recuperar'
    }))
]
```

406 Uso de Python para el desarrollo de microservicios

En este archivo, adjuntamos los métodos GET y POST de solicitudes HTTP con la URL `grades/` a la lista y creamos métodos de la clase `GradeViewSet` que implementamos anteriormente en el archivo `views.py`. Del mismo modo, adjuntamos el GET solicitud con las calificaciones/`<str:id>` URL al método de recuperación de la clase `GradeViewSet`. Al usar este archivo, podemos agregar un mapeo de URL adicional a las funciones/métodos de Python.

Esto concluye nuestra implementación de nuestro microservicio Grades. El siguiente paso es ejecutar este servicio bajo el servidor web Django para su validación. Pero antes de ejecutar el servicio, nos aseguraremos de que los objetos del modelo se transfieran a la base de datos. Esto es equivalente a inicializar la base de datos en el caso de Flask. En el caso de Django, ejecutamos los siguientes dos comandos para preparar los cambios y luego ejecutarlos:

```
python3 manage.py hacer migraciones
```

```
python3 administrar.py migrar
```

A menudo, los desarrolladores pasan por alto este importante paso y obtienen errores al intentar iniciar la aplicación. Por lo tanto, asegúrese de que todos los cambios se ejecuten antes de iniciar el servidor web mediante el siguiente comando:

```
python3 manage.py servidor de ejecución
```

Esto iniciará un servidor web en un puerto predeterminado, 8000, en nuestra máquina host local. Tenga en cuenta que la configuración predeterminada, incluida la base de datos y el servidor web con atributos de host y puerto, se puede cambiar en el archivo `settings.py`. Además, recomendamos configurar una cuenta de usuario para la aplicación de administración mediante el siguiente comando:

```
python3 manage.py crear superusuario
```

Este comando le pedirá que seleccione un nombre de usuario, una dirección de correo electrónico y una contraseña para la cuenta de administrador. Una vez que nuestro microservicio está realizando las funciones como se esperaba, es hora de empaquetarlo en un contenedor y ejecutarlo como una aplicación de contenedor. Esto se explica en la siguiente sección.

Contenedorización de un microservicio

La contenedorización es un tipo de virtualización del sistema operativo en el que las aplicaciones se ejecutan en su espacio de usuario separado, pero comparten el mismo sistema operativo. Este espacio de usuario separado se denomina **contenedor**. Docker es la plataforma más popular para crear, administrar y ejecutar aplicaciones como contenedores. Docker todavía tiene más del 80 % de la cuota de mercado, pero existen otros tiempos de ejecución de contenedores, como **CoreOS rkt**, **Mesos**, **Ixc** y **containerd**. Antes de usar Docker para contener nuestro microservicio, revisaremos rápidamente los componentes principales de la plataforma Docker:

- **Docker Engine:** esta es la aplicación principal de Docker para crear, empaquetar y ejecutar aplicaciones basadas en contenedores.
- **Imagen de Docker:** una imagen de Docker es un archivo que se utiliza para ejecutar la aplicación en un entorno del contenedor. Las aplicaciones desarrolladas con Docker Engine se almacenan como imágenes de Docker, que son una colección de código de aplicación, bibliotecas, archivos de recursos y cualquier otra dependencia necesaria para la ejecución de la aplicación.
- **Docker Hub:** este es un repositorio en línea de imágenes de Docker para compartir dentro de su equipo y también con la comunidad. **Docker Registry** es otro término usado en el mismo contexto. Docker Hub es un nombre oficial del registro de Docker que administra los repositorios de imágenes de Docker.
- **Docker Compose:** esta es una herramienta para crear y ejecutar aplicaciones de contenedores utilizando un archivo basado en YAML en lugar de usar los comandos CLI de Docker Engine. Docker Compose proporciona una forma sencilla de implementar y ejecutar varios contenedores con dependencias y atributos de configuración. Por lo tanto, le recomendamos que utilice Docker Compose o una tecnología similar para crear y ejecutar sus contenedores.

Para usar Docker Engine y Docker Compose, debe tener una cuenta con el registro de Docker. Además, debe descargar e instalar Docker Engine y Docker Compose en su máquina antes de iniciar los siguientes pasos:

1. Como primer paso, crearemos una lista de las dependencias de nuestro proyecto utilizando el archivo de comando pip freeze de la siguiente manera:

```
pip congelar -> requisitos.txt
```

Este comando creará una lista de dependencias y las exportará al archivo requirements.txt .

Docker Engine usará este archivo para descargar estas bibliotecas dentro del contenedor sobre un intérprete de Python. El contenido de este archivo en nuestro proyecto es el siguiente:

```
asgiref==3.4.1
```

```
Django==3.2.5
```

408 Uso de Python para el desarrollo de microservicios

```
django-rest-framework==0.1.0
djangorestframework==3.12.4
pytz==2021.1
sqlparse==0.4.1
```

2. En el siguiente paso, construiremos Dockerfile. Docker Engine también utilizará este archivo para crear una nueva imagen de un contenedor. En nuestro caso, agregaremos las siguientes líneas a este archivo:

```
DESDE python: 3.8-delgado
ENV PYTHON SIN BÚSQUEDA 1
WORKDIR /aplicación
COPIAR requisitos.txt /app/requisitos.txt
EJECUTAR pip install -r requisitos.txt
COPIAR ./aplicación
CMD python manage.py servidor de ejecución 0.0.0.0:8000
```

La primera línea de este archivo establece la imagen base para este contenedor y la configuramos en Python:3.8-slim, que ya está disponible en el repositorio de Docker. La segunda línea del archivo establece una variable de entorno para un mejor registro. El resto de las líneas se explican por sí mismas, ya que en su mayoría son comandos de Unix.

3. Como siguiente paso, crearemos un archivo Docker Compose (docker-compose.yml) como sigue:

```
versión: '3.7'
servicios:
  calificaciones:
    construir:
      contexto: .
    archivo acoplable: archivo acoplable
    puertos:
      - 8000:8000
    volúmenes:
      - .:/aplicación
```

Este es un archivo YAML y definimos contenedores como servicios en él. Como solo tenemos un contenedor, definimos el servicio gradesms . Tenga en cuenta que la compilación apunta a Dockerfile que acabamos de crear y suponiendo que está en el mismo directorio que este archivo docker-compose.yml . El puerto del contenedor 8000 se asigna al puerto del servidor web 8000. Este es un paso importante para permitir el tráfico desde el contenedor a su aplicación dentro del contenedor.

4. Como último paso, montamos el directorio actual (.) en el directorio /app dentro del contenedor. Esto permitirá que los cambios realizados en nuestro sistema se reflejen en el contenedor y viceversa. Este paso es importante si está creando contenedores durante el ciclo de desarrollo.
5. Podemos iniciar nuestro contenedor usando el siguiente comando Docker Compose:

docker-compose up

Por primera vez, creará una nueva imagen de contenedor y requerirá acceso a Internet para descargar la imagen de contenedor base del registro de Docker. Después de crear una imagen de contenedor, iniciará automáticamente el contenedor.

Los detalles de cómo funcionan Docker Engine y Docker Compose están más allá del alcance de este libro, pero le recomendamos que se familiarice con la tecnología de contenedores como Docker a través de su documentación en línea (<https://docs.docker.com/>).

Reutilización de nuestra aplicación API para estudiantes

Reutilizaremos nuestra aplicación API para estudiantes , que desarrollamos en el capítulo anterior. Esta aplicación se iniciará con su servidor incorporado y la llamaremos Estudiantes microservicio para nuestra aplicación de muestra. No habrá ningún cambio en esta aplicación.

Actualización de nuestra aplicación web Estudiantes

La aplicación webapp , que desarrollamos para el estudio de caso en el capítulo anterior, solo usa apiapp a través de una API REST. En una versión revisada de esta aplicación web, utilizaremos el microservicio Grades y el microservicio Students para obtener la lista de Grade objetos y la lista de objetos de Estudiante . La función de lista en nuestra aplicación web combinará las dos listas de objetos para proporcionar información adicional a los clientes web. la lista actualizada La función en el archivo webapp.py será la siguiente:

```
ESTUDIANTES_MS = http://localhost:8080/estudiantes
GRADOS_MS = "http://localhost:8000/grados"
@aplicación.get('/')
lista definida():
```

410 Uso de Python para el desarrollo de microservicios

```

student_svc_resp = solicitudes.get(ESTUDIANTES_MS)
estudiantes = json.loads(student_svc_resp.text)

grades_svc_resp = solicitudes.get(GRADES_MS)
grades_list = json.loads(grades_svc_resp.text)
grades_dict = {cls_item['grade']:
    cls_item para cls_item en grades_list}

para estudiante en estudiantes:
    estudiante['edificio'] =
        grades_dict[estudiante['grado']][['edificio']]
    estudiante['profesor'] =
        calificaciones_dict[estudiante['calificación']][['profesor']]

volver render_template('main.html', estudiantes=estudiantes)

```

En este código revisado, creamos un diccionario de calificaciones usando una comprensión de diccionario de la lista de objetos de Calificaciones . Este diccionario se usará para insertar los atributos de calificación dentro de los objetos Estudiante antes de enviarlos a la plantilla Jinja para renderizar. En nuestra plantilla principal de Jinja (main.html), agregamos dos columnas adicionales, **Building** y **Teacher**, a la tabla **Students** , como se muestra aquí:

The screenshot shows a two-part interface. The top part is a modal window titled "Add a Student" with fields for First name, Last Name, and Grade, and a "Submit" button. The bottom part is a table titled "Students" with columns: No, Name, Grade, Building, Teacher, and Actions. It contains two rows of data: one for John Lee (Building #10, Miss Hannah) and one for Brian Miles (NA, Miss Fey). Each row has "Update" and "Delete" buttons in the Actions column.

No	Name	Grade	Building	Teacher	Actions
1	John Lee	10	Building #10	Miss Hannah	<button>Update</button> <button>Delete</button>
2	Brian Miles	7	NA	Miss Fey	<button>Update</button> <button>Delete</button>

Figura 11.6 – Página principal actualizada con datos de Edificio y Profesor

En esta sección, cubrimos la creación de un microservicio, su implementación como un contenedor de Docker y una aplicación web en un servidor web, y la combinación de los resultados de los dos microservicios para una aplicación web.

Implementación del microservicio Students en GCP Cloud Run

Hasta ahora, hemos utilizado el microservicio Students como una aplicación web con una API REST alojada en un servidor de desarrollo de matraz. Ahora es el momento de contenerlo e implementarlo en **Google Cloud Platform (GCP)**. GCP tiene un motor de tiempo de ejecución (**Cloud Run**) para implementar contenedores y ejecutarlos como un servicio (microservicio). Aquí están los pasos involucrados:

1. Para empaquetar el código de la aplicación de nuestro microservicio Estudiantes en un contenedor, primero identificaremos una lista de dependencias y las exportaremos a requisitos. archivo de texto Ejecutaremos el siguiente comando desde el entorno virtual del proyecto de microservicio Students :

```
pip congelar -> requisitos.txt
```

2. El siguiente paso es construir un Dockerfile en el directorio raíz del proyecto, como el que preparamos para nuestro microservicio Grades . El contenido del Dockerfile es el siguiente:

```
DESDE python: 3.8-delgado
ENV PYTHONUNBUFERED Verdadero
WORKDIR /aplicación
COPIAR . .
#Instalar dependencias de producción.
EJECUTAR pip install -r requisitos.txt
EJECUTAR pip instalar Flask gunicorn
# Ejecute el servicio web en el inicio del contenedor. usaremos # gunicorn y vincularemos
nuestra api_app como la aplicación principal
CMD exec gunicorn --bind:$PORT --workers 1 --threads 8 api_app:app
```

Para implementar nuestra aplicación en GCP Cloud Run, Dockerfile será suficiente. Pero primero, debemos crear la imagen del contenedor con el SDK de GCP Cloud. Esto requerirá que creamos un proyecto de GCP usando Cloud SDK o GCP Console. Explicamos los pasos para crear un proyecto de GCP y asociarle una cuenta de facturación en los capítulos anteriores. Suponemos que ha creado un proyecto con el nombre dirigido por estudiantes en GCP.

412 Uso de Python para el desarrollo de microservicios

3. Una vez que el proyecto esté listo, podemos usar el siguiente comando para construir un contenedor imagen de nuestra aplicación API de Estudiantes :

```
compilaciones de gcloud enviar --tag gcr.io/students-run/students
```

Tenga en cuenta que gcr significa Google Container Registry.

4. Para crear una imagen, debemos proporcionar el atributo de la etiqueta en el siguiente formato:

```
<nombre de host>/<ID del proyecto>/<nombre de la imagen>
```

5. En nuestro caso, el nombre de host es gcr.io, que tiene su sede en los Estados Unidos. Podemos usar una imagen creada localmente también, pero primero debemos configurar el atributo de la etiqueta según el formato mencionado anteriormente y luego enviarlo al registro de Google. Esto se puede lograr con los siguientes comandos de Docker:

```
etiqueta docker SOURCE_IMAGE <nombre de host>/<ID del proyecto>/<nombre de la imagen>:tagid
```

```
docker push <nombre de host>/<ID del proyecto>/<nombre de la imagen>
```

```
#o si queremos empujar una etiqueta específica
```

```
docker push <nombre de host>/<ID de proyecto>/<nombre de imagen>:etiqueta
```

Como podemos ver, el comando de compilación de gcloud puede lograr dos pasos en un solo comando.

6. El siguiente paso es ejecutar la imagen cargada. Podemos ejecutar nuestra imagen de contenedor por con el siguiente comando del SDK de Cloud:

```
gcloud ejecutar implementar --image gcr.io/students-run/students
```

La ejecución de nuestra imagen también se puede activar desde la consola de GCP. Una vez que el contenedor se haya implementado y ejecutado correctamente, el resultado de este comando (o en una consola de GCP) incluirá la URL de nuestro microservicio.

Para consumir esta nueva versión del microservicio Estudiantes de GCP Cloud Run, actualizaremos nuestra aplicación web para cambiar y usar la URL de este servicio recientemente implementado en GCP Cloud Run. Si probamos nuestra aplicación web con un Grades desplegado localmente microservicio y el microservicio Estudiantes implementado de forma remota , obtendremos los mismos resultados que se muestran anteriormente en la Figura 11.6 y podemos realizar todas las operaciones como lo hicimos cuando el microservicio Estudiantes se implementó localmente.

Esto concluye nuestra discusión sobre la creación de microservicios utilizando diferentes marcos de Python, implementándolos tanto localmente como en la nube y consumiéndolos desde una aplicación web.

Resumen

En este capítulo, presentamos la arquitectura de microservicios y analizamos sus ventajas y desventajas. Cubrimos varias prácticas recomendadas para crear, implementar y poner en funcionamiento microservicios. También analizamos las opciones de desarrollo disponibles en Python para crear microservicios que incluyen Flask, Django, Falcon, Nameko, Bottle y Tornado.

Seleccionamos Flask y Django para crear microservicios de muestra. Para implementar un nuevo microservicio, usamos Django con su marco REST (DRF). Esta implementación de microservicio también le presenta cómo funciona el marco Django en general.

Más adelante, brindamos detalles sobre cómo contener un microservicio recién creado mediante Docker Engine y Docker Compose. Finalmente, convertimos nuestra aplicación API para estudiantes en una imagen de Docker y la implementamos en GCP Cloud Run. Actualizamos la aplicación web de Estudiantes para consumir dos microservicios implementados en diferentes partes del mundo.

Los ejemplos de código incluidos en este capítulo le brindarán experiencia práctica en la creación e implementación de microservicios para diferentes entornos. Este conocimiento es beneficioso para cualquier persona que busque crear microservicios en sus próximos proyectos. En el próximo capítulo, exploraremos cómo usar Python para desarrollar funciones sin servidor, otro nuevo paradigma de desarrollo de software para la nube.

Preguntas

1. ¿Podemos implementar un microservicio sin un contenedor?
2. ¿Es adecuado que dos microservicios comparten una misma base de datos pero con diferente esquema?
3. ¿Qué es Docker Compose y cómo ayuda a implementar microservicios?
4. ¿Es REST el único formato para el intercambio de datos para microservicios?

Otras lecturas

- Hands-On Docker para Microservicios con Python, por Jaime Buelta
- Desarrollo de Microservicios Python, por Tarek Ziade
- Diseño impulsado por el dominio: abordar la complejidad en el corazón del software, por Eric Evans
- Tutoriales de inicio rápido de Google Cloud Run para crear e implementar microservicios, disponibles en <https://cloud.google.com/run/docs/quickstarts/>

respuestas

1. Sí, pero se recomienda desplegarlo en un contenedor.
2. Técnicamente, es factible, pero no es una buena práctica. La falla de la base de datos traerá ambos microservicios caídos.
3. Docker Compose es una herramienta para implementar y ejecutar aplicaciones de contenedores utilizando un archivo YAML. Proporciona un formato fácil para definir diferentes servicios (contenedores) con atributos de implementación y tiempo de ejecución.
4. Una API REST es la interfaz más popular para el intercambio de datos para microservicios, pero no la única. Los microservicios también pueden usar RPC y protocolos basados en eventos para el intercambio de datos.

12

Construcción sin servidor Funciones usando Pitón

La computación sin servidor es un nuevo modelo de computación en la nube que separa la administración de servidores físicos o virtuales y el software a nivel de infraestructura, como los sistemas de bases de datos, de la aplicación misma. Este modelo permite a los desarrolladores centrarse únicamente en el desarrollo de aplicaciones y permite que otra persona administre los recursos de la infraestructura subyacente. Los proveedores de la nube son la mejor opción para adoptar este modelo. Los contenedores no solo son oportunos para implementaciones complejas, sino que también son una tecnología innovadora para la era de la **computación sin servidor**. Además de los contenedores, existe otra forma de computación sin servidor, que se conoce como **Función como servicio (FaaS)**. En este nuevo paradigma, los proveedores de la nube ofrecen una plataforma para desarrollar y ejecutar funciones de aplicaciones o funciones **sin servidor**, generalmente en respuesta a un evento o como una llamada directa a esas funciones. Todos los proveedores de nube pública, como Amazon, Google, Microsoft, IBM y Oracle, ofrecen este servicio. El enfoque de este capítulo será comprender y crear funciones sin servidor usando Python.

416 Creación de funciones sin servidor usando Python

En este capítulo, cubriremos los siguientes temas:

- Introducción de funciones sin servidor
- Comprensión de las opciones de implementación para funciones sin servidor
- Aprender a crear funciones sin servidor con un estudio de caso

Después de completar este capítulo, debería tener una comprensión clara del rol de las funciones sin servidor en la computación en la nube y cómo construirlas usando Python.

Requerimientos técnicos

La siguiente es una lista de los requisitos técnicos para este capítulo:

- Deberá tener Python 3.7 o posterior instalado en su computadora.
- Para implementar una función sin servidor en **Google Cloud Platform (GCP)** Cloud Functions, necesitará una cuenta de GCP (una prueba gratuita funcionará bien).
- Necesitará una cuenta (es decir, una cuenta gratuita) con SendGrid para enviar correos electrónicos.

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter12>.

Comencemos con una introducción a las funciones sin servidor.

Introducción a las funciones sin servidor

Una función sin servidor es un modelo que se puede utilizar para desarrollar y ejecutar componentes o módulos de software sin necesidad de conocer o preocuparse por una plataforma de alojamiento subyacente. Estos módulos o componentes de software se conocen como **funciones Lambda** o **funciones de nube** en las ofertas de productos de los proveedores de nube pública. Amazon fue el primer proveedor que ofreció tales funciones sin servidor en su plataforma AWS como **AWS Lambda**. Le siguieron Google y Microsoft, que ofrecen **Google Cloud Functions** y **Azure Functions**, respectivamente.

Por lo general, una función sin servidor tiene cuatro componentes, de la siguiente manera:

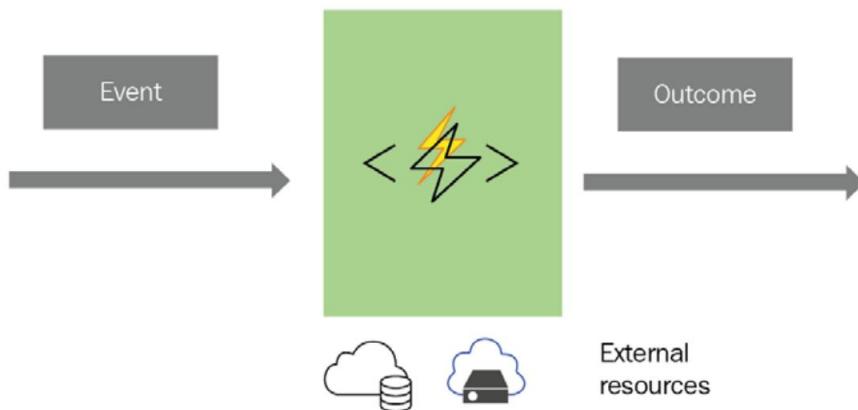


Figura 12.1 – Los componentes de una función sin servidor

Estos cuatro componentes se describen a continuación:

- **Código funcional:** Esta es una unidad de programación que realiza ciertas tareas según el objetivo comercial o funcional de la función. Por ejemplo, podemos escribir una función sin servidor para procesar un flujo de entrada de datos o escribir una actividad programada para verificar ciertos recursos de datos con fines de monitoreo.
- **Eventos:** las funciones sin servidor no están diseñadas para usarse como microservicios. En cambio, están destinados a usarse en función de un disparador que puede ser iniciado por un evento de un sistema pub/sub, o pueden venir como llamadas HTTP basadas en un evento externo en el campo, como eventos de sensores de campo.
- **Resultado:** cuando se activa una función sin servidor para realizar un trabajo, hay una salida de la función, que puede ser una simple respuesta a la persona que llama o desencadenar las siguientes acciones para mitigar el impacto de un evento. Un ejemplo del resultado de una función sin servidor es activar otro servicio en la nube, como un servicio de base de datos, o enviar un correo electrónico a las partes suscritas.
- **Recursos:** a veces, el código funcional tiene que usar un recurso adicional para hacer su trabajo, por ejemplo, un servicio de base de datos o almacenamiento en la nube para acceder o enviar archivos.

Beneficios

Las funciones sin servidor traen consigo todos los beneficios de la informática sin servidor, de la siguiente manera:

- **Facilidad de desarrollo:** las funciones sin servidor eliminan las complejidades de la infraestructura de los desarrolladores para que puedan concentrarse en el aspecto funcional del programa.
- **Escalabilidad integrada:** las funciones sin servidor se ofrecen con escalabilidad integrada para manejar cualquier crecimiento de tráfico en cualquier momento.
- **Rentabilidad:** las funciones sin servidor no solo reducen los costos de desarrollo, sino que también ofrecen una implementación optimizada y un modo operativo. Por lo general, este es un modelo de pago por uso, lo que significa que solo se le cobrará por el tiempo durante el cual se ejecuta su función.
- Independiente de la **tecnología:** las funciones sin servidor son independientes de la tecnología. Esto significa que puede crearlos en muchos lenguajes de programación utilizando una variedad de diferentes recursos en la nube.

Tenga en cuenta que existen algunas limitaciones para las funciones sin servidor; por ejemplo, tendremos menos control a nivel del sistema en la construcción de dichas funciones y la solución de problemas puede ser complicada sin acceso a nivel del sistema.

Casos de uso

Hay varios usos posibles de las funciones sin servidor. Por ejemplo, podemos usar dichas funciones para el procesamiento de datos si recibimos un evento de carga de archivos en el almacenamiento en la nube o si tenemos datos disponibles a través de transmisión en tiempo real. En particular, las funciones sin servidor se pueden integrar con los sensores **de Internet de las cosas (IoT)**. Por lo general, los sensores de IoT son miles en número. Las funciones sin servidor poseen la capacidad de manejar las solicitudes de una gran cantidad de sensores de manera escalable. Una aplicación móvil puede usar funciones como un servicio de back-end para realizar ciertas tareas o procesar datos sin poner en peligro los recursos del dispositivo móvil. Un uso práctico de las funciones sin servidor en la vida real es el producto **Amazon Alexa**. No es posible poner cada habilidad u onza de inteligencia dentro del propio dispositivo Alexa. En su lugar, utiliza funciones de Amazon Lambda para estas habilidades. Otra razón por la que Alexa usa las funciones de Amazon Lambda es la capacidad de escalarlas según la demanda. Por ejemplo, algunas funciones pueden usarse con más frecuencia que otras, como las consultas meteorológicas.

En la siguiente sección, investigaremos las diversas opciones de implementación para implementar y ejecutar funciones sin servidor.

Comprensión de las opciones de implementación para funciones sin servidor

El uso de una máquina virtual u otro recurso de tiempo de ejecución en nubes públicas para aplicaciones a las que se accede esporádicamente podría no ser una solución comercialmente atractiva. En tales situaciones, las funciones sin servidor vienen al rescate. Aquí, un proveedor de la nube ofrece recursos administrados dinámicamente para su aplicación y solo le cobra cuando su aplicación se ejecuta en respuesta a un evento determinado. En otras palabras, una función sin servidor es un método informático de back-end que es un servicio bajo demanda y de pago por uso que solo se ofrece en nubes públicas. Presentaremos algunas opciones para implementar funciones sin servidor en las nubes públicas, de la siguiente manera:

- **AWS Lambda:** se considera que es una de las primeras ofertas de servicios de cualquiera de los proveedores de nube pública. Las funciones de AWS Lambda se pueden escribir en Python, Node.js, Java, PowerShell, Ruby, Java, C# y Go. Las funciones de AWS Lambda se pueden ejecutar en respuesta a eventos, como cargas de archivos a **Amazon S3**, una notificación de **Amazon SNS** o una llamada directa a la API. Las funciones de AWS Lambda no tienen estado.
- **Azure Functions:** Microsoft presentó Azure Functions casi dos años después del lanzamiento de las funciones de AWS Lambda. Estas funciones se pueden adjuntar a eventos dentro de la infraestructura de la nube. Microsoft brinda soporte para compilar y depurar estas funciones mediante Visual Studio, Visual Studio Code, IntelliJ y Eclipse. Azure Functions se puede escribir en C#, F#, Node.js, PowerShell, PHP y Python. Además, Microsoft ofrece **funciones duraderas** que nos permiten escribir funciones con estado en un entorno sin servidor.
- **Google Cloud Functions:** GCP ofrece Google Cloud Functions sin servidor funciones Google Cloud Functions se puede escribir en Python, Node.js, Go, .NET, Ruby y PHP. Al igual que sus competidores, AWS Lambda y Azure Functions, Google Cloud Functions puede activarse mediante solicitudes HTTP o eventos de la infraestructura de Google Cloud. Google le permite usar Cloud Build para la implementación y prueba automática de Cloud Functions.

Además de los tres principales proveedores de nube pública, hay algunas ofertas más de otros proveedores de nube. Por ejemplo, IBM ofrece Cloud Functions que se basan en el proyecto de código abierto **Apache OpenWhisk**. Oracle ofrece su plataforma informática sin servidor basada en el proyecto **Fn** de código abierto. La belleza de usar estos proyectos de código abierto es que puede desarrollar y probar su código localmente. Además, estos proyectos le permiten transferir su código de una nube a otra nube o incluso a una implementación de entorno local sin ningún cambio.

420 Creación de funciones sin servidor usando Python

Vale la pena mencionar otro marco que es bien conocido en la informática sin servidor, llamado **Serverless Framework**. Esta no es una plataforma de implementación, sino una herramienta de software que se puede usar localmente para compilar y empaquetar su código para la implementación sin servidor y luego se puede usar para implementar el paquete en una de sus nubes públicas favoritas. El marco sin servidor admite varios lenguajes de programación, como Python, Java, Node.js, Go, C#, Ruby y PHP.

En la siguiente sección, crearemos un par de funciones sin servidor usando Python.

Aprender a crear funciones sin servidor

En esta sección, investigaremos cómo crear funciones sin servidor para uno de los proveedores de nube pública. Aunque Amazon AWS fue pionero en las funciones sin servidor en 2014 al ofrecer funciones AWS Lambda, utilizaremos la plataforma Google Cloud Functions para nuestras funciones de ejemplo. La razón de esto es que ya presentamos GCP con gran detalle en capítulos anteriores, y puede aprovechar la misma cuenta de GCP para la implementación de estas funciones de ejemplo. Sin embargo, le recomendamos enfáticamente que use las otras plataformas, especialmente si planea usar sus funciones sin servidor en el futuro. Los principios básicos para construir e implementar estas funciones en varias plataformas en la nube son los mismos.

GCP Cloud Functions ofrece varias formas de desarrollar e implementar funciones sin servidor (en el futuro, las llamaremos Cloud Functions en el contexto de GCP). Exploraremos dos tipos de eventos en nuestro ejemplo de funciones en la nube, que se pueden describir de la siguiente manera:

- La primera función en la nube se creará e implementará con GCP Console desde de extremo a extremo. Esta función de la nube se activará en función de una llamada (o evento) HTTP.
- La segunda Cloud Function será parte de un caso de estudio para construir una aplicación que escucha un evento en la infraestructura de la nube y realiza una acción, como enviar un correo electrónico como respuesta a este evento. La función de la nube utilizada en este estudio de caso se creará e implementará mediante el **kit de desarrollo de software (SDK) de la nube**.

Comenzaremos creando una función en la nube con GCP Console.

Creación de una función en la nube basada en HTTP con el Consola de GCP

Comencemos con el proceso de desarrollo de Google Cloud Function. Crearemos una función en la nube muy simple que proporcione la fecha actual y la hora actual para un activador HTTP.

Tenga en cuenta que el disparador HTTP es la forma más fácil de invocar una función de la nube.

Primero, necesitaremos un proyecto GCP. Puede crear un nuevo proyecto de GCP con GCP Console para esta Cloud Function o un proyecto de GCP existente. Los pasos relacionados con cómo crear un proyecto de GCP y asociarle una cuenta de facturación se analizan en el Capítulo 9, Programación de Python para la nube. Una vez que tenga listo un proyecto de GCP, crear una nueva función en la nube es un proceso de tres pasos. Explicaremos estos pasos en las siguientes subsecciones.

Configuración de los atributos de la función de la nube

Cuando iniciamos el flujo de trabajo **Crear función** desde GCP Console, se nos solicita que proporcionemos la definición de la función en la nube, de la siguiente manera:

Cloud Functions | Copy function

Configuration — Code

Basics

Function name * my-datetime

Region asia-southeast1

Trigger

HTTP

Trigger type HTTP

URL https://asia-southeast1-students-run.cloudfunctions.net/my-datetime

Authentication

Allow unauthenticated invocations
Check this if you are creating a public API or website.

Require authentication
Manage authorised users with Cloud IAM.

Require HTTPS

SAVE **CANCEL**

Figura 12.2: los pasos para crear una nueva función en la nube con GCP Console (1/2)

422 Creación de funciones sin servidor usando Python

Un resumen de alto nivel de la definición de la función de la nube aparece de la siguiente manera:

1. Proporcionamos el **nombre de la función** (en nuestro caso, my-datetime) y seleccionamos la **región** de GCP para alojar esta función.
2. Seleccionamos HTTP como **tipo de activador** para nuestra función. Seleccionar un disparador para su función es el paso más importante. También hay otros activadores disponibles, como **Cloud Pub/Sub** y **Cloud Storage**. En el momento de escribir este libro, GCP ha agregado algunos activadores más con fines de evaluación.
3. En aras de la simplicidad, permitiremos el acceso no autenticado para nuestra función.

Después de hacer clic en el botón **Guardar**, se nos pedirá que **ingresemos TIEMPO DE EJECUCIÓN, CONSTRUIR Y AJUSTES DE CONEXIONES**, como se muestra en la siguiente captura de pantalla:

RUNTIME, BUILD AND CONNECTIONS SETTINGS ^

RUNTIME	BUILD	CONNECTIONS
Memory allocated * 128 MiB		
Timeout * 60	seconds	?

Runtime service account ?

Runtime service account App Engine default service account

Auto-scaling ?

Maximum number of instances 1

Runtime environment variables ?

+ ADD VARIABLE

NEXT **CANCEL**

Figura 12.3: los pasos para crear una nueva función en la nube con GCP Console (2/2)

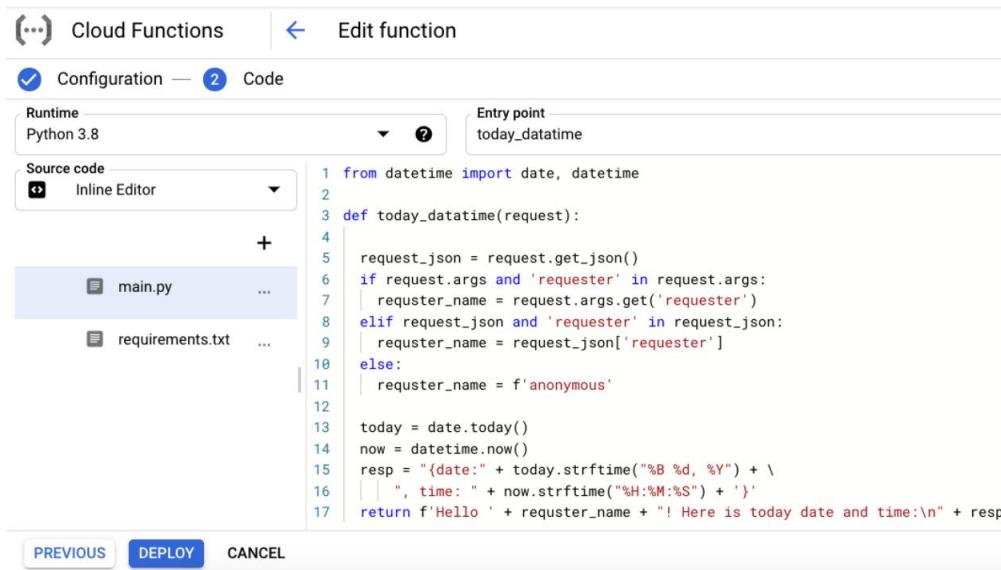
Podemos proporcionar la **CONFIGURACIÓN DE TIEMPO DE EJECUCIÓN, CONSTRUCCIÓN Y CONEXIONES** de la siguiente manera:

1. Podemos dejar los atributos de tiempo de ejecución en su configuración predeterminada, pero reduciremos la **Memoria asignada** a **128 MiB** para nuestra función. Hemos asociado una cuenta de servicio predeterminada como cuenta de **servicio Runtime** a esta función. Dejaremos el ajuste de **escala automática** en su configuración predeterminada, pero esto se puede establecer en un número máximo de instancias para nuestra función.
2. Podemos agregar **variables de entorno de tiempo de ejecución** debajo de la pestaña **TIEMPO DE EJECUCIÓN** si tenemos ese requisito para hacerlo. No agregaremos ninguna variable de entorno para nuestra Cloud Function.
3. Debajo de la pestaña **CONSTRUIR**, hay una opción para agregar **variables de entorno de compilación**. No agregaremos ninguna variable para nuestra Cloud Function.
4. Debajo de la pestaña **CONEXIONES**, podemos dejar la configuración predeterminada como está y permitir que todo el tráfico acceda a nuestra función de nube.

Después de configurar el tiempo de ejecución, la compilación y la conexión de Cloud Function, el siguiente paso será agregar el código de implementación para esta Cloud Function.

Agregar código Python a una función en la nube

Después de hacer clic en el botón **Siguiente**, como se muestra en la Figura 12.3, GCP Console nos ofrecerá una vista para definir o agregar los detalles de implementación de la función, como se muestra en la siguiente captura de pantalla:



The screenshot shows the 'Edit function' interface for a Cloud Function named 'today_datETIME'. The 'Configuration' tab is selected, showing the runtime is set to 'Python 3.8'. The 'Code' tab is also visible. The code editor contains the following Python code:

```

1 from datetime import date, datetime
2
3 def today_datETIME(request):
4
5     request_json = request.get_json()
6     if request.args and 'requester' in request.args:
7         requester_name = request.args.get('requester')
8     elif request_json and 'requester' in request_json:
9         requester_name = request_json['requester']
10    else:
11        requester_name = f'anonymous'
12
13    today = date.today()
14    now = datetime.now()
15    resp = "{date:" + today.strftime("%B %d, %Y") + \
16           ", time: " + now.strftime("%H:%M:%S") + '}'
17    return f'Hello ' + requester_name + '! Here is today date and time:\n' + resp

```

At the bottom of the interface, there are buttons for 'PREVIOUS', 'DEPLOY' (highlighted in blue), and 'CANCEL'.

Figura 12.4: los pasos de implementación de una función en la nube con GCP Console

424 Creación de funciones sin servidor usando Python

Las opciones que están disponibles para agregar nuestro código Python son las siguientes:

- Podemos seleccionar varias opciones de tiempo de ejecución como Java, PHP, Node.js o varias versiones de Python. Seleccionamos **Python 3.8** como el tiempo de ejecución para nuestra función en la nube. • El atributo **Punto de entrada** debe ser el nombre de la función en nuestro código. Google Cloud Function invocará la función en nuestro código en función de este atributo **de punto de entrada** .
- El código fuente de Python se puede agregar en línea usando el **Editor** en línea a la derecha lado; alternativamente, se puede cargar usando un archivo ZIP desde su máquina local o incluso desde el almacenamiento en la nube. También podemos proporcionar la ubicación del repositorio GCP **Cloud Source** para el código fuente. Aquí, seleccionamos implementar nuestra función usando la herramienta **Editor en línea** .
- Para Python, la plataforma GCP Cloud Functions crea automáticamente dos archivos: main.py y requisitos.txt. El archivo main.py tendrá nuestra implementación de código y el archivo requirements.txt debe contener nuestras dependencias de bibliotecas de terceros.

Un código de muestra, que se muestra dentro de la herramienta **Editor en línea** , primero verifica si la persona que llama ha enviado un atributo de solicitante en la solicitud HTTP o no. Según el valor del atributo del solicitante , enviaremos un mensaje de bienvenida con la fecha y la hora de hoy. Implementamos un ejemplo de código similar con dos API web separadas usando una aplicación web de Flask en el Capítulo 9, Programación de Python para la nube, para demostrar las capacidades de GCP App Engine.

Una vez que estemos satisfechos con nuestro código de Python, implementaremos la función en la plataforma Google Cloud Functions.

Implementación de una función en la nube

El siguiente paso es implementar esta función usando el botón **Implementar** en la parte inferior de la pantalla, como se muestra en la Figura 12.4. GCP comenzará a implementar la función de inmediato y puede llevar unos minutos completar esta actividad. Es importante comprender que las funciones de Google Cloud se implementan mediante contenedores, al igual que los microservicios en GCP Cloud Run. Las diferencias clave son que se pueden invocar mediante diferentes tipos de eventos y utilizan el modelo de precios de pago por uso.

Una vez desplegada nuestra función, podemos duplicarla, probarla o eliminarla de la **Lista de Cloud Functions**, como se muestra en la siguiente captura de pantalla:

<input type="checkbox"/>	Name ↑	Region	Trigger	Memory allocated	Actions
<input type="checkbox"/>	handle_storage_delete	us-central1	Bucket: ch12-cfunc-testing	256 MiB	⋮
<input type="checkbox"/>	handle_storage_upload	us-central1	Bucket: ch12-cfunc-testing	256 MiB	⋮
<input type="checkbox"/>	my-datetime	us-east1	HTTP	128 MiB	⋮

Copy function
▶ Test function
☰ View logs

 Delete

Figura 12.5: la vista principal de Google Cloud Functions Ahora,

le mostraremos rápidamente lo conveniente que es probar y solucionar problemas de nuestra función de nube mediante GCP Console. Una vez que hayamos seleccionado la opción **Función de prueba** para nuestra Función de nube recién implementada, GCP Console nos ofrecerá una página de prueba, que es similar a la que se muestra en la Figura 12.6, debajo de la pestaña **PRUEBA**. Para probar nuestra función de nube implementada, podemos pasar el atributo del solicitante en formato JSON, de la siguiente manera:

```
{"solicitante":"Juan"}
```

426 Creación de funciones sin servidor usando Python

Después de hacer clic en [...]PROBAR LA FUNCIÓN, podemos ver los resultados en **Salida** y los detalles del registro en la sección **Registros** en la parte inferior de la pantalla, como se muestra en la Figura 12.6. Debido a que estamos usando un activador HTTP para nuestra función de nube, también podemos probarlo usando un navegador web o la utilidad CURL desde cualquier lugar de Internet. Sin embargo, debemos asegurarnos de que nuestra función en la nube incluya a todos los usuarios como su miembro con el rol de invocador de funciones en la nube. Esto se puede configurar debajo de los **PERMISOS** pestaña. Sin embargo, no recomendamos hacerlo sin configurar un mecanismo de autenticación para su Cloud Function:

The screenshot shows the 'Function details' page for a Cloud Function named 'my-datetime'. At the top, there's a 'Version' dropdown set to 'Version 3, deployed at 10 Jul 2021, 10:04:03 ...'. Below it, tabs include METRICS, DETAILS, SOURCE, VARIABLES, TRIGGER, PERMISSIONS, LOGS, and TESTING (which is selected). Under 'Triggering event', the code is shown as:

```
1  {"requester": "John"}  
2
```

Below this, a 'TEST THE FUNCTION' button is visible. A note says: 'Testing in the Cloud Console has a 60-second timeout. Note that this is different from the limit set in the function configuration.' The 'Output' section shows the command '\$ Hello John! Here is today date and time:' followed by the output: '{date:July 10, 2021, time: 07:24:52}'. The 'Logs' section shows two log entries:

- 2021-07-10T07:24:51.717822528Z my-datetime ke0djohswsq2 Function execution started
- 2021-07-10T07:24:52.033451420Z my-datetime ke0djohswsq2 Function execution took 316 ms, finished with status code: 200

Figura 12.6: prueba de su función en la nube con GCP Console

Crear una función en la nube simple con GCP Console es un proceso sencillo.

A continuación, exploraremos un caso de estudio de una aplicación real de Cloud Functions.

Estudio de caso: creación de una aplicación de notificación para eventos de almacenamiento en la nube

En este estudio de caso, desarrollaremos una función en la nube que se active para eventos en un depósito de **Google Storage**. Al recibir tal evento, nuestra función de nube enviará un correo electrónico a una lista predefinida de direcciones de correo electrónico como notificación. El flujo de esta aplicación de notificación con una función en la nube aparece de la siguiente manera:

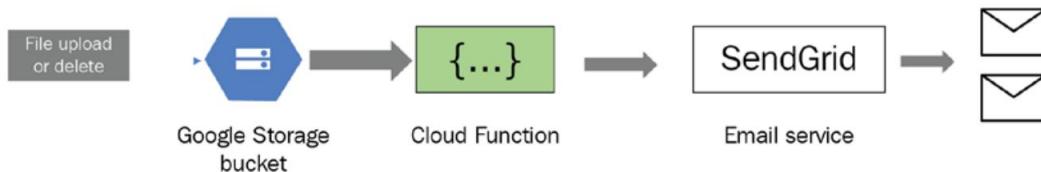


Figura 12.7: una función de la nube que escucha los eventos del depósito de almacenamiento de Google

Tenga en cuenta que podemos configurar nuestra función de nube para escuchar uno o más eventos de almacenamiento de Google.

Google Cloud Functions admite los siguientes eventos de Google Storage:

- **finalizar:** este evento se crea cuando se agrega o reemplaza un nuevo archivo dentro de un depósito de almacenamiento.
- **eliminar:** este evento representa la eliminación de un archivo de un depósito de almacenamiento. Esto se aplica a depósitos sin control de versiones. Tenga en cuenta que un archivo no se elimina en realidad, pero se archiva si el depósito está configurado para usar el control de versiones.
- **archivo:** este evento se genera cuando se archiva un archivo. La operación de archivado se activa cuando se elimina o sobrescribe un archivo para depósitos con control de versiones.
- **actualización de metadatos:** Si hay alguna actualización en los metadatos de un archivo, se crea este evento.

Después de recibir un evento de un depósito de Google Storage, Cloud Function extraerá los atributos del contexto y los objetos de evento pasados como argumentos a nuestra función de nube. Luego, la función de la nube utilizará un servicio de correo electrónico de terceros (como SendGrid de Twilio) para enviar la notificación.

Como requisito previo, debe crear una cuenta gratuita con SendGrid (<https://sendgrid.com/>). Después de crear una cuenta, deberá crear al menos un usuario remitente dentro de su cuenta SendGrid. Además, deberá configurar una clave API secreta dentro de SendGrid cuenta que se puede usar con Cloud Function para enviar correos electrónicos. Twilio SendGrid ofrece dentro del rango de 100 correos electrónicos por día de forma gratuita, lo que es lo suficientemente bueno para fines de prueba.

Para este estudio de caso, escribiremos nuestro código Python para Cloud Function localmente y luego lo implementaremos en la plataforma de Google Cloud Functions usando el SDK de Cloud. Implementaremos esta aplicación de notificación paso a paso, de la siguiente manera:

1. Crearemos un depósito de almacenamiento para adjuntarlo a nuestra función en la nube y cargaremos o eliminaremos archivos de este depósito para simular los eventos de nuestra función en la nube. Podemos usar el siguiente comando de Cloud SDK para crear un nuevo depósito:

```

gsutil mb gs://<nombre del depósito>
gsutil mb gs://muasif-testcloudfn #Cubo de ejemplo
creado

```

428 Creación de funciones sin servidor usando Python

Para simplificar la generación de estos eventos, desactivaremos el control de versiones en nuestro depósito de almacenamiento mediante el siguiente comando del SDK de Cloud:

```
El control de versiones de gsutil desencadenó gs://muasif-testcloudfn
```

2. Una vez que el cubo de almacenamiento esté listo, crearemos un directorio de proyecto local y configuraremos un entorno virtual usando los siguientes comandos:

```
python -m venv mienv
```

```
frente myenv/bin/activar
```

3. A continuación, instalaremos el paquete Python de sendgrid usando la utilidad pip , de la siguiente manera:

```
pip instalar sendgrid
```

4. Una vez que nuestras bibliotecas de terceros hayan sido instaladas, tendremos que crear el archivo de dependencias requirements.txt , de la siguiente manera:

```
pip congelar -> requisitos.txt
```

5. A continuación, crearemos un nuevo archivo de Python (main.py) con handle_storage_ función de evento dentro de él. Esta función será el punto de entrada para nuestra función en la nube. El código de muestra para esta función de punto de entrada es el siguiente:

```
#principal.py
desde sendgrid importar SendGridAPIClient
from sendgrid.helpers.mail import Mail, Email, To,
Contenido

def handle_storage_event(evento, contexto):
    from_email = Correo electrónico ("abc@domain1.com")
    to_emails = Para("xyz@domain2.com")
    subject = "Su notificación de depósito de almacenamiento"
    content = f"Cubo afectado:{evento['cubo']} \n" +
              f"Archivo afectado: {evento['nombre']} \n" f"Hora del + \
              evento: {evento['tiempo de creación']} \n" + \
              f"ID del evento: {context.event_id} \n" + \
              f"Tipo de evento: {context.event_type}"
```

```
mail = Mail(from_email, to_emails, asunto, contenido)
sg = SendGridAPIClient()
respuesta = sg.send(correo)
print(response.status_code) # para fines de registro
imprimir (respuesta. encabezados)
```

En el ejemplo de código de Python anterior, se espera que nuestra función handle_storage_event reciba objetos de contexto y eventos como argumentos de entrada. un evento objeto es un diccionario que contiene los datos del evento. Podemos acceder al evento. datos de este objeto usando claves como depósito (es decir, el nombre del depósito), nombre (es decir, el nombre del archivo) y timeCreated (es decir, la hora de creación). el contexto object proporciona el contexto del evento, como event_id y event_type. Además, usamos la biblioteca sendgrid para preparar el contenido del correo electrónico y luego enviamos el correo electrónico con la información del evento a la lista de correo electrónico de destino.

6. Una vez que tenemos nuestro archivo de código de Python (en nuestro caso, este es main.py) y Los requisitos.txt están listos, podemos desencadenar la operación de implementación con el siguiente comando de Cloud SDK:

```
Las funciones de gcloud implementan handle_storage_create \
--punto de entrada handle_storage_event --runtime python38 \
--trigger-resource gs://muasif-testcloudfn/
--trigger-event google.storage.object.finalize
--set-env-vars SENDGRID_API_KEY=<Su CLAVE SEND-GRID>
```

Deberíamos ejecutar este comando en un proyecto de GCP con la facturación habilitada, como se explicó en la sección anterior. Hemos proporcionado el nombre de nuestra función en la nube como handle_storage_create, y el atributo de punto de entrada se establece en la función handle_storage_event en el código de Python. Configuramos el evento desencadenante al evento de finalización . Al usar set-env-vars, configuramos SENDGRID_API_KEY para el servicio SendGrid.

El comando de implementación empaquetará el código Python del directorio actual, preparará la plataforma de destino según el archivo requirements.txt y luego implementará nuestro código Python en la plataforma GCP Cloud Functions. En nuestro caso, podemos crear un archivo .gcloudignore para excluir los archivos y directorios para que el comando de implementación de Cloud SDK pueda ignorarlos .

430 Creación de funciones sin servidor usando Python

7. Una vez que implementamos nuestra función en la nube, podemos probarla cargando un archivo local en nuestro depósito de almacenamiento mediante el comando SDK de la nube, de la siguiente manera:

```
gsutil cp test1.txt gs://muasif-testcloudfn
```

Tan pronto como se haya completado la operación de copia de archivo (carga), el finalize evento activará la ejecución de nuestra función de nube. Como resultado, recibiremos un correo electrónico con los detalles del evento. También podemos verificar los registros de las funciones de la nube usando el siguiente comando:

```
lectura de registros de funciones de gcloud --limit 50
```

Para esta aplicación de notificación, adjuntamos nuestra función de nube solo al evento Finalizar .

Sin embargo, ¿qué sucede si también desea adjuntar otro tipo de evento, como un evento Eliminar ?

Bueno, solo se puede adjuntar una función de nube a un evento desencadenante. Pero espera, una función de la nube es una entidad de implementación y no el código del programa real. Esto significa que no necesitamos escribir o duplicar nuestro código de Python para manejar otro tipo de evento. Podemos crear una nueva función en la nube usando el mismo código de Python pero para el evento Eliminar , de la siguiente manera:

```
Las funciones de gcloud implementan handle_storage_delete \
--punto de entrada handle_storage_event --runtime python38 \
--trigger-resource gs://muasif-testcloudfn/ \
--trigger-event google.storage.object.delete
--set-env-vars SENDGRID_API_KEY=<Su CLAVE SEND-GRID>
```

Si nota esta versión del comando de implementación , los únicos cambios que hicimos fueron con el nombre de la función de la nube y el tipo de evento desencadenante. Este comando de implementación creará una nueva función en la nube y funcionará en paralelo con una función en la nube anterior, pero se activará en función de un evento diferente (en este caso, se trata de una eliminación).

Para probar el evento de eliminación con nuestra función de nube recién agregada, podemos eliminar el archivo ya cargado (o cualquier archivo) de nuestro depósito de almacenamiento usando la siguiente función de nube Comando SDK:

```
gsutil rm gs://muasif-testcloudfn/test1.txt
```

Podemos crear más funciones en la nube usando el mismo código de Python para otros eventos de almacenamiento.

Esto concluye nuestra discusión sobre cómo compilar Cloud Functions para eventos de almacenamiento mediante el SDK de Cloud. Todos los pasos descritos con el SDK de Cloud también se pueden implementar con GCP Console.

Resumen

En este capítulo, presentamos la computación sin servidor y FaaS, seguido de un análisis de los ingredientes principales de las funciones sin servidor. A continuación, discutimos los beneficios clave de las funciones sin servidor y sus peligros. Además, analizamos varias opciones de implementación que están disponibles para crear e implementar funciones sin servidor, y estas opciones incluyen AWS Lambda, Azure Functions, Google Cloud Functions, Oracle Fn e IBM Cloud Functions. En la parte final de este capítulo, creamos una función de Google Cloud simple basada en un activador HTTP usando GCP Console. Luego, creamos una aplicación de notificación basada en eventos de almacenamiento de Google y una función de nube de Google usando el SDK de la nube. Estas funciones sin servidor se implementaron utilizando la plataforma Google Cloud Functions.

Los ejemplos de código incluidos en este capítulo deberían brindarle cierta experiencia sobre cómo usar GCP Console y Cloud SDK para compilar e implementar Cloud Functions. Este conocimiento práctico es beneficioso para cualquier persona que esté buscando desarrollar una carrera en la informática sin servidor.

En el próximo capítulo, exploraremos cómo usar Python con el aprendizaje automático.

Preguntas

1. ¿En qué se diferencian las funciones sin servidor de los microservicios?
2. ¿Cuál es el uso práctico de las funciones sin servidor en ejemplos del mundo real?
3. ¿Qué son las funciones duraderas y quién las ofrece?
4. Una función en la nube se puede adjuntar a varios activadores. ¿Es esto cierto o falso?

Otras lecturas

- Computación sin servidor con Google Cloud por Richard Rose
- Dominar AWS Lambda por Yohan Wadia
- Dominar la informática sin servidor de Azure por Lorenzo Barbieri y Massimo Bonanni
- Tutoriales de inicio rápido de Google Cloud Functions para crear e implementar Cloud Funciones, que está disponible en <https://cloud.google.com/functions/documents/inicio-rapido>

respuestas

1. Ambos son dos ofertas diferentes de computación sin servidor. Por lo general, sin servidor
Las funciones se desencadenan por un evento y se basan en el modelo de pago por uso. En comparación, los microservicios generalmente se consumen a través de llamadas API y no se basan en el modelo de pago por uso.
2. Amazon Alexa utiliza las funciones de AWS Lambda para proporcionar inteligencia y otras habilidades para sus usuarios.
3. Las funciones duraderas son una extensión de Microsoft Azure Functions, que ofrece funcionalidad con estado en un entorno sin servidor.
4. Falso. Una función en la nube solo se puede adjuntar a un solo disparador.

13

pitón y Aprendizaje automático

El aprendizaje automático (ML) es una rama de la **inteligencia artificial (IA)** que se basa en la creación de modelos mediante el aprendizaje de patrones a partir de datos y luego el uso de esos modelos para hacer predicciones. Es una de las técnicas de IA más populares para ayudar a los humanos y a las empresas de muchas maneras. Por ejemplo, se está utilizando para diagnóstico médico, procesamiento de imágenes, reconocimiento de voz, predicción de amenazas, minería de datos, clasificación y muchos más escenarios. Todos entendemos la importancia y la utilidad del aprendizaje automático en nuestras vidas. Python, al ser un lenguaje conciso pero poderoso, se usa ampliamente para implementar modelos de aprendizaje automático. La capacidad de Python para procesar y preparar datos mediante bibliotecas como NumPy, pandas y PySpark lo convierte en la opción preferida de los desarrolladores para crear y entrenar modelos de aprendizaje automático.

En este capítulo, analizaremos el uso de Python para tareas de aprendizaje automático de forma optimizada. Esto es especialmente importante porque el entrenamiento de un modelo de ML es una tarea de computación intensiva y la optimización del código es fundamental cuando se usa Python para el aprendizaje automático.

434 Python y aprendizaje automático

Cubriremos los siguientes temas en este capítulo:

- Introducción al aprendizaje automático
- Usar Python para el aprendizaje automático
- Probar y evaluar modelos de aprendizaje automático
- Implementación de modelos de aprendizaje automático en la nube

Después de completar este capítulo, comprenderá cómo usar Python para crear, entrenar y evaluar modelos de aprendizaje automático y cómo implementarlos en la nube y usarlos para hacer predicciones.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- Debe instalar bibliotecas adicionales para el aprendizaje automático, como SciPy, NumPy, pandas y scikit-learn.
- Para implementar un modelo ML en la plataforma AI de GCP, necesitará una cuenta de GCP (una prueba gratuita funcionará bien).

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter13>.

Comenzaremos nuestra discusión con una introducción al aprendizaje automático.

Introducción al aprendizaje automático

En la programación tradicional, proporcionamos datos y algunas reglas como entrada a nuestro programa para obtener el resultado deseado. El aprendizaje automático es un enfoque de programación fundamentalmente diferente, en el que los datos y el resultado esperado se proporcionan como entrada para producir un conjunto de reglas. Esto se denomina **modelo** en la nomenclatura de aprendizaje automático. Este concepto se ilustra en el siguiente diagrama:

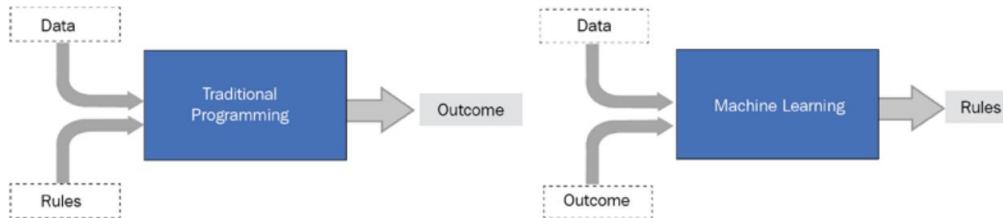


Figura 13.1 – Programación tradicional versus programación de aprendizaje automático

Para comprender cómo funciona el aprendizaje automático, debemos familiarizarnos con sus componentes o elementos principales:

- **Conjunto de datos:** sin un buen conjunto de datos, el aprendizaje automático no es nada. Los buenos datos son el verdadero poder del aprendizaje automático. Tiene que recopilarse de diferentes entornos y cubrir varias situaciones para representar un modelo cercano a un proceso o sistema del mundo real. Otro requisito para los datos es que debe ser grande, y por grande nos referimos a miles de registros. Además, los datos deben ser lo más precisos posible y contener información significativa. Los datos se utilizan para entrenar el sistema y también para evaluar su precisión. Podemos recopilar datos de muchas fuentes, pero la mayoría de las veces está en un formato sin formato. Podemos usar técnicas de procesamiento de datos utilizando bibliotecas como pandas, como discutimos en los capítulos anteriores.

 - **Extracción de características:** antes de usar cualquier dato para construir un modelo, necesitamos entender qué tipo de datos tenemos y cómo están estructurados. Una vez que hayamos entendido eso, podemos seleccionar qué características de los datos pueden ser utilizadas por un algoritmo ML para construir un modelo. También podemos calcular características adicionales basadas en el conjunto de características original. Por ejemplo, si tenemos datos de imagen sin procesar en forma de píxeles, que en sí mismos pueden no ser útiles para entrenar un modelo, podemos usar la longitud o el ancho de la forma dentro de una imagen como características para crear reglas para nuestro modelo.

 - **Algoritmo:** Este es un programa que se utiliza para construir un modelo ML a partir de los datos disponibles. En términos matemáticos, un algoritmo de aprendizaje automático intenta aprender una función de destino $f(X)$ que puede asignar los datos de entrada, X , a una salida, y , así:

$$y = f(X)$$
- Hay varios algoritmos disponibles para diferentes tipos de problemas y situaciones porque no hay un solo algoritmo que pueda resolver todos los problemas. Algunos algoritmos populares son **la regresión lineal**, **los árboles de clasificación y regresión** y **el clasificador de vectores de soporte (SVC)**. Los detalles matemáticos de cómo funcionan estos algoritmos están más allá del alcance de este libro. Recomendamos consultar los enlaces adicionales proporcionados en la sección Lecturas adicionales para obtener detalles sobre estos algoritmos.

436 Python y aprendizaje automático

- **Modelos:** a menudo escuchamos el término modelo en el aprendizaje automático. Un modelo es una representación matemática o computacional de un proceso que está ocurriendo en nuestra vida cotidiana. Desde una perspectiva de aprendizaje automático, es el resultado de un algoritmo de aprendizaje automático cuando lo aplicamos a nuestro conjunto de datos. Esta salida (modelo) puede ser un conjunto de reglas o alguna estructura de datos específica que se puede usar para hacer predicciones cuando se usa para cualquier dato del mundo real.
- **Capacitación:** este no es un componente o paso nuevo en el aprendizaje automático. Cuando decimos entrenar un modelo, esto significa aplicar un algoritmo de ML a un conjunto de datos para producir un modelo de ML. Se dice que el modelo que obtenemos como salida está entrenado en un determinado conjunto de datos. Hay tres formas diferentes de entrenar un modelo:
 - a) **Aprendizaje supervisado:** esto incluye proporcionar el resultado deseado, junto con nuestros registros de datos. El objetivo aquí es aprender cómo se puede asignar la entrada (X) a la salida (Y) utilizando los datos disponibles. Este enfoque de aprendizaje se utiliza para problemas de clasificación y regresión. La clasificación de imágenes y la predicción de los precios de la vivienda (regresión) son un par de ejemplos reales de aprendizaje supervisado. En el caso del procesamiento de imágenes, podemos entrenar a un modelo para que identifique el tipo de animal en una imagen, como un gato o un perro, según la forma, el largo y el ancho de la imagen. Para entrenar nuestro modelo de clasificación de imágenes, etiquetaremos cada imagen en el conjunto de datos de entrenamiento con el nombre del animal. Para predecir el precio de las casas, debemos proporcionar datos sobre las casas en la ubicación que estamos buscando, como el área en la que se encuentran, la cantidad de habitaciones y baños, etc.
 - b) **Aprendizaje no supervisado:** En este caso, entrenamos un modelo sin conocer el resultado deseado. El aprendizaje no supervisado generalmente se aplica a casos de uso de agrupación y asociación. Este tipo de aprendizaje se basa principalmente en la observación y la búsqueda de grupos o conglomerados de puntos de datos para que los puntos de datos de un grupo o conglomerado tengan características similares. Las tiendas minoristas en línea como Amazon utilizan ampliamente este tipo de enfoque de aprendizaje para encontrar diferentes grupos de clientes (agrupación) en función de su comportamiento de compra y ofrecerles los artículos que les interesan. Las tiendas en línea también intentan encontrar una asociación entre diferentes compras., como la probabilidad de que una persona que compra el artículo A también quiera comprar el artículo B.
 - c) **Aprendizaje por refuerzo:** En el caso del aprendizaje por refuerzo, el modelo es recompensado por tomar una decisión adecuada en una situación particular. En este caso, no hay datos de entrenamiento disponibles, pero el modelo tiene que aprender de la experiencia.
Los coches autónomos son un ejemplo popular de aprendizaje por refuerzo.

- **Pruebas:** Necesitamos probar nuestro modelo en un conjunto de datos que no se usa para entrenar el modelo. Un enfoque tradicional es entrenar nuestro modelo usando dos tercios del conjunto de datos y probar el modelo usando el tercio restante.

Además de los tres enfoques de aprendizaje que discutimos, también tenemos el aprendizaje profundo.

Este es un tipo avanzado de aprendizaje automático basado en el enfoque de cómo el cerebro humano logra cierto tipo de conocimiento utilizando algoritmos de redes neuronales. En este capítulo, utilizaremos el aprendizaje supervisado para construir nuestros modelos de muestra.

En la siguiente sección, exploraremos las opciones disponibles en Python para el aprendizaje automático.

Uso de Python para el aprendizaje automático

Python es un lenguaje popular en la comunidad de científicos de datos debido a su simplicidad, compatibilidad multiplataforma y gran soporte para el análisis y procesamiento de datos a través de sus bibliotecas. Uno de los pasos clave en el aprendizaje automático es preparar datos para construir los modelos ML, y Python es un ganador natural al hacerlo. El único desafío al usar Python es que es un lenguaje interpretado, por lo que la velocidad de ejecución del código es lenta en comparación con lenguajes como C. Pero esto no es un problema importante ya que hay bibliotecas disponibles para maximizar la velocidad de Python mediante el uso de múltiples núcleos, de **las unidades centrales de procesamiento (CPU)** o **unidades de procesamiento gráfico (GPU)** en paralelo.

En la siguiente subsección, presentaremos algunas bibliotecas de Python para el aprendizaje automático.

Presentación de bibliotecas de aprendizaje automático en Python

viene con varias bibliotecas de aprendizaje automático. Ya mencionamos bibliotecas compatibles como NumPy, SciPy y pandas, que son fundamentales para el refinamiento, el análisis y la manipulación de datos. En esta sección, analizaremos brevemente las bibliotecas de Python más populares para crear modelos de aprendizaje automático.

scikit-aprender

Esta biblioteca es una opción popular porque tiene una gran variedad de algoritmos y herramientas de ML integrados para evaluar el rendimiento de esos algoritmos de ML. Estos algoritmos incluyen algoritmos de clasificación y regresión para el aprendizaje supervisado y algoritmos de agrupación y asociación para el aprendizaje no supervisado. scikit-learn está escrito principalmente en Python y se basa en la biblioteca NumPy para muchas operaciones. Para principiantes, recomendamos comenzar con la biblioteca scikit-learn y luego pasar al siguiente nivel de bibliotecas, como TensorFlow. Usaremos scikit-learn para ilustrar los conceptos de creación, capacitación y evaluación de los modelos ML.

scikit-learn también ofrece **algoritmos de aumento de gradiente**. Estos algoritmos se basan en el concepto matemático de **Gradiente**, que es la pendiente de una función. Mide el cambio en un error en el contexto ML. La idea de los algoritmos basados en gradientes es ajustar los parámetros de forma iterativa para encontrar el mínimo local de una función (minimizando errores para los modelos ML). Los algoritmos de aumento de gradiente utilizan la misma estrategia para mejorar un modelo de forma iterativa teniendo en cuenta el rendimiento del modelo anterior, ajustando los parámetros para el nuevo modelo y estableciendo el objetivo para aceptar el nuevo modelo si minimiza más los errores. que el modelo anterior.

XGBoost

XGBoost, o **eXtreme Gradient Boosting**, es una biblioteca de algoritmos que se basa en árboles de decisión potenciados por gradientes. Esta biblioteca es popular porque es extremadamente rápida y ofrece el mejor rendimiento en comparación con otras implementaciones de algoritmos de aumento de gradiente, así como algoritmos tradicionales de aprendizaje automático. scikit-learn también ofrece algoritmos de aumento de gradiente, que son fundamentalmente los mismos que XGBoost, aunque XGBoost es significativamente rápido. La razón principal es la máxima utilización del paralelismo entre diferentes núcleos de una sola máquina o en un grupo distribuido de nodos. XGBoost también puede regularizar los árboles de decisión para evitar el ajuste excesivo del modelo a los datos. XGBoost no es un marco completo para el aprendizaje automático, pero ofrece principalmente algoritmos (modelos). Para usar XGBoost, debemos usar scikit-learn para el resto de las funciones y herramientas de la utilidad, como el análisis de datos y la preparación de datos.

TensorFlow

TensorFlow es otra biblioteca de código abierto muy popular para el aprendizaje automático, desarrollada por el equipo de Google Brain para computación de alto rendimiento. TensorFlow es particularmente útil para entrenar y ejecutar redes neuronales profundas y es una opción popular en el área del aprendizaje profundo.

Keras

Esta es una API de código abierto para el aprendizaje profundo de redes neuronales en Python. Keras es más una API de alto nivel además de TensorFlow. Para los desarrolladores, usar Keras es más conveniente que usar TensorFlow directamente, por lo que se recomienda usar Keras si está comenzando a desarrollar modelos de aprendizaje profundo con Python. Keras puede funcionar tanto con CPU como con GPU.

PyTorch

PyTorch es otra biblioteca de aprendizaje automático de código abierto que es una implementación de Python de la popular biblioteca **Torch** en C.

En la siguiente sección, analizaremos brevemente las mejores prácticas para usar Python para el aprendizaje automático.

Mejores prácticas de entrenamiento de datos con Python

Ya hemos destacado la importancia de los datos a la hora de entrenar un modelo de aprendizaje automático. En esta sección, destacaremos algunas mejores prácticas y recomendaciones al preparar y usar datos para entrenar su modelo ML. Estos son los siguientes:

- Como mencionamos anteriormente, la recopilación de un gran conjunto de datos es de vital importancia (unos pocos miles de registros de datos o al menos varios cientos). Cuanto mayor sea el tamaño de los datos, más preciso será el modelo ML.
- Limpie y refine sus datos antes de comenzar cualquier entrenamiento. Esto significa que hay no debe haber campos faltantes o campos engañosos en los datos. Las bibliotecas de Python como pandas son muy útiles para tales tareas.
- Es importante usar un conjunto de datos sin comprometer la privacidad y la seguridad de los datos. Debe asegurarse de que no está utilizando los datos de alguna otra organización sin la aprobación correspondiente.
- Las GPU funcionan bien con aplicaciones de uso intensivo de datos. Le recomendamos que utilice GPU para entrenar sus algoritmos y obtener resultados más rápidos. Las bibliotecas como XGBoost, TensorFlow y Keras son bien conocidas por usar GPU con fines de capacitación.
- Cuando se trata de un gran conjunto de datos para el entrenamiento, es importante utilizar el la memoria del sistema de manera eficiente. Deberíamos cargar los datos en la memoria en fragmentos o utilizar clústeres distribuidos para procesar los datos. Le recomendamos que utilice la función de generador tanto como pueda.
- También es una buena práctica observar el uso de la memoria durante las tareas de uso intensivo de datos (por ejemplo, mientras entrena un modelo) y liberar memoria periódicamente forzando la recolección de elementos no utilizados para liberar objetos sin referencia.

Ahora que hemos cubierto las bibliotecas de Python disponibles y las mejores prácticas de uso de Python para el aprendizaje automático, es hora de comenzar a trabajar con ejemplos de código reales.

Creación y evaluación de un modelo de aprendizaje automático

Antes de comenzar a escribir un programa en Python, evaluaremos el proceso de creación de un modelo de aprendizaje automático.

Aprender sobre un proceso de creación de modelos de ML

Discutimos los diferentes componentes del aprendizaje automático en la sección Introducción al aprendizaje automático. El proceso de aprendizaje automático utiliza esos elementos como entrada para entrenar un modelo. Este proceso sigue un procedimiento con tres fases principales, y cada fase tiene varios pasos. Estas fases se muestran aquí:

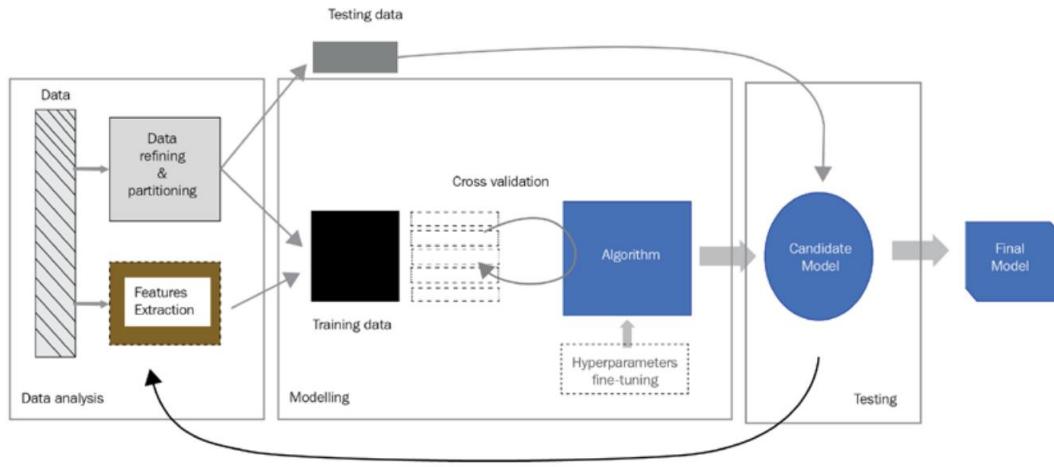


Figura 13.2 – Pasos para construir un modelo ML utilizando un enfoque de aprendizaje clásico

Cada fase, junto con los pasos detallados de la misma, se describe aquí:

- **Análisis de datos:** en esta fase, recopilamos datos sin procesar y los transformamos en un formulario que se puede analizar y luego usar para entrenar y probar un modelo. Podemos descartar algunos datos, como registros con valores vacíos. A través del análisis de datos, tratamos de seleccionar las características (atributos) que se pueden usar para identificar patrones en nuestros datos. La extracción de características es un paso muy importante, y mucho depende de estas características al construir un modelo exitoso. En muchos casos, tenemos que ajustar las funciones después de la fase de prueba para asegurarnos de que tenemos el conjunto correcto de funciones para los datos. Por lo general, dividimos los datos en dos conjuntos; una parte se usa para entrenar el modelo en la fase de modelado, mientras que la otra parte se usa para probar la precisión del modelo entrenado en la fase de prueba. Podemos omitir la fase de prueba si estamos evaluando el modelo utilizando otros enfoques, como la **validación cruzada**. Recomendamos tener una fase de prueba en su proceso de creación de ML y reservar algunos datos (que no se ven en el modelo) para la fase de prueba, como se muestra en el diagrama anterior.

• **Modelado:** esta fase se trata de entrenar nuestro modelo en función de los datos de entrenamiento y las características que extrajimos en la fase anterior. En un enfoque de ML tradicional, podemos usar los datos de entrenamiento tal como están para entrenar nuestro modelo. Pero para garantizar que nuestro modelo tenga una mayor precisión, podemos usar las siguientes técnicas adicionales:

- a) Podemos dividir nuestros datos de entrenamiento en segmentos y usar un segmento para evaluar nuestro modelo y usar los segmentos restantes para entrenar el modelo. Repetimos esto para una combinación diferente de segmentos de entrenamiento y el segmento de evaluación. Este enfoque de evaluación se denomina validación cruzada.
- b) Los algoritmos de ML vienen con varios parámetros que se pueden usar para ajustar el modelo para que se ajuste mejor a los datos. El ajuste fino de estos parámetros, también conocidos como **hiperparámetros**, generalmente se realiza junto con la validación cruzada durante la fase de modelado.

Los valores de características en los datos pueden usar diferentes escalas de medición, lo que dificulta la creación de reglas con una combinación de dichas características. En tales casos, podemos transformar los datos (valores característicos) en una escala común o en una escala normalizada (por ejemplo, 0 a 1). Este paso se llama escalar los datos o normalización. Todos estos pasos de escalado y evaluación (o algunos de ellos) se pueden agregar a una tubería (como una tubería de Apache Beam) y se pueden ejecutar juntos para evaluar diferentes combinaciones para seleccionar el mejor modelo. El resultado de esta fase es un modelo de ML candidato, como se muestra en el diagrama anterior.

• **Prueba:** en la fase de prueba, usamos los datos que reservamos para probar la precisión del modelo de ML candidato que construimos en la fase anterior. El resultado de esta fase se puede usar para agregar o eliminar algunas características y ajustar el modelo hasta que obtengamos uno con una precisión aceptable.

Una vez que estemos satisfechos con la precisión de nuestro modelo, podemos implementarlo para predecir en función de los datos del mundo real.

Creación de un modelo de ML de muestra

En esta sección, construiremos un modelo de ML de muestra usando Python, que identificará tres tipos de plantas Iris. Para construir este modelo, utilizaremos un conjunto de datos comúnmente disponible que contiene cuatro características (largo y ancho de sépalos y pétalos) y tres tipos de plantas de iris.

Para este ejercicio de código, usaremos los siguientes componentes:

- Usaremos el conjunto de datos Iris proporcionado por el repositorio de aprendizaje automático de UC Irvine (<http://archive.ics.uci.edu/ml/>). Este conjunto de datos contiene 150 registros y tres patrones esperados para identificar. Este es un conjunto de datos refinado que viene con las características necesarias ya identificadas.

- Usaremos varias bibliotecas de Python, de la siguiente manera:

- a) Las bibliotecas pandas y matplotlib, para el análisis de datos
- b) La biblioteca scikit-learn, para entrenar y probar nuestro modelo ML

Primero, escribirímos un programa en Python para analizar el conjunto de datos de Iris.

Análisis del conjunto de datos de Iris

Para facilitar la programación, descargamos los dos archivos para el conjunto de datos de Iris (iris. data e iris.names) de <https://archive.ics.uci.edu/ml/machine-learning-databases/iris/>.

Podemos acceder directamente al archivo de datos de este repositorio a través de Python. Pero en nuestro programa de muestra, usaremos una copia local de los archivos. La biblioteca scikit-learn también proporciona varios conjuntos de datos como parte de la biblioteca y se puede usar directamente con fines de evaluación.

Decidimos usar los archivos reales, ya que estarán cerca de los escenarios del mundo real, donde usted mismo recopila datos y luego los usa en su programa.

El archivo de datos de Iris contiene 150 registros que se ordenan según el resultado esperado.

En el archivo de datos, se proporcionan los valores de cuatro características diferentes. Estas cuatro características se describen en el archivo iris.names como longitud del sépalo, anchura del sépalo, longitud del pétalo y anchura del pétalo. Los tipos de salida esperados de la planta Iris, según el archivo de datos, son Iris-setosa, Iris-versicolor e Iris-virginica. Cargaremos los datos en un DataFrame de pandas y luego los analizaremos en busca de diferentes atributos de interés. Algunos ejemplos de código para analizar los datos de Iris son los siguientes:

```
#iris_data_analysis.py
de pandas importar read_csv
desde matplotlib importar pyplot

archivo_datos = "iris/iris.datos"
iris_names = ['longitud del sépalo', 'ancho del sépalo', 'pétalo-longitud', 'ancho de pétalo', 'clase']
df = read_csv(archivo_datos, nombres=nombres_iris)

imprimir (df.forma)
imprimir (df. cabeza (20))
imprimir(df.describir())
imprimir(df.groupby('clase').tamaño())

# diagramas de caja y bigotes
```

```
df.plot(kind='box', subplots=True, layout=(3,3), sharex=False, sharey=False)

pyplot.mostrar()

# comprobar los histogramas
df.hist()
pyplot.mostrar()
```

En la primera parte del análisis de datos, verificamos algunas métricas sobre los datos utilizando las funciones de la biblioteca de pandas, de la siguiente manera:

- Usamos el método de la forma para obtener la dimensión del DataFrame. Debería ser [150, 5] para el conjunto de datos de Iris, ya que tenemos 150 registros y cinco columnas (cuatro para características y una para el resultado esperado). Este paso asegura que todos los datos se carguen correctamente en nuestro DataFrame.
- Verificamos los datos reales usando el método de cara o cruz . Esto es solo para ver los datos de forma visual, especialmente si no hemos visto lo que hay dentro del archivo de datos.
- El método describe nos dio los diferentes KPI estadísticos disponibles para los datos.
El resultado de este método es el siguiente:

```
sépalo-longitud sépalo-ancho pétalo-longitud pétalo-ancho
cuenta 150.000000 150.000000 150.000000 150.000000
media 5.843333 3.054000 3.758667 1.198667
estándar 0,828066 0,433594 1,764420 0,763161
min 4.300000 2.000000 1.000000 0.100000
25% 5,100000 2,800000 1,600000 0,300000
50% 5,800000 3,000000 4,350000 1,300000
75% 6,400000 3,300000 5,100000 1,800000
máx. 7,900000 4,400000 6,900000 2,500000
```

Estos KPI pueden ayudarnos a seleccionar el algoritmo adecuado para el conjunto de datos.

- Se utilizó el método groupby para identificar el número de registros de cada clase (nombre de la columna para el resultado esperado). La salida indicará que hay 50 registros para cada tipo de planta Iris:

```
Iris-setosa 50
Iris-versicolor 50
Iris-virginica 50
```

En la segunda parte del análisis, intentamos usar un **diagrama de caja** (también conocido como **diagrama de caja y bigotes**) y diagramas de **histograma**. Los diagramas de caja son una forma visual de mostrar los KPI que recibimos mediante el método de descripción (valor mínimo, primer cuartil, segundo cuartil (mediana), tercer cuartil y valor máximo). Esta gráfica le dirá si sus datos están distribuidos simétricamente o agrupados en un cierto rango, o cuánto de sus datos están sesgados hacia un lado de la distribución. Para nuestro conjunto de datos Iris, obtendremos los diagramas de caja para nuestras cuatro características de la siguiente manera:

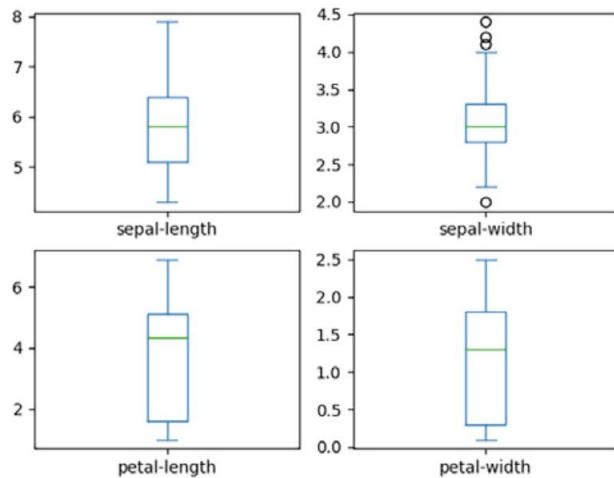


Figura 13.3: diagramas de caja y bigotes de las características del conjunto de datos de Iris

A partir de estos gráficos, podemos ver que los datos **de longitud de pétalo** y **ancho de pétalo** tienen la mayor agrupación entre el primer cuartil y el tercer cuartil. Podemos confirmar esto analizando la distribución de datos utilizando los gráficos de histograma, de la siguiente manera:

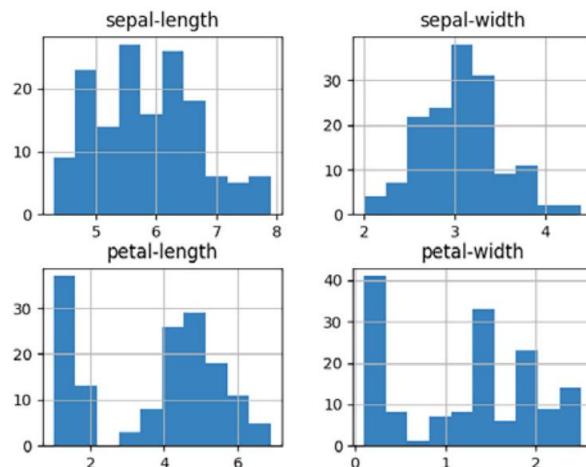


Figura 13.4 – Histograma de las características del conjunto de datos de Iris

Después de analizar los datos y seleccionar el tipo correcto de algoritmo (modelo) para usar, pasaremos al siguiente paso, que es entrenar nuestro modelo.

Entrenamiento y prueba de un modelo de ML de muestra

Para entrenar y probar un algoritmo ML (modelo), debemos seguir estos pasos:

1. Como primer paso, dividiremos nuestro conjunto de datos original en dos grupos: datos de entrenamiento y datos de prueba. Este enfoque de dividir los datos se denomina método de **retención**. La biblioteca scikit-learn proporciona la función `train_test_split` para que esta división sea conveniente:

```
#iris_build_svm_model.py (#1)
# Dividir el conjunto de datos
X = df.drop('clase', eje = 1)
y = df['clase']
X_tren, X_prueba, y_tren, y_prueba = tren_prueba_dividir(X,
y, test_size=0.20, random_state=1, shuffle=True)
```

Antes de llamar a la función `train_test_split`, dividimos el conjunto de datos completo en un conjunto de datos de características (generalmente llamado `X`, que debe estar en mayúsculas en la nomenclatura de aprendizaje automático) y el conjunto de datos de salida esperado (llamado `y`, que debe estar en minúsculas en la nomenclatura de aprendizaje automático). Estos dos conjuntos de datos (`X` e `y`) están divididos por la función `train_test_split` según nuestro `test_size` (20%, en nuestro caso). También permitimos que los datos se mezclen antes de dividirlos. El resultado de esta operación nos dará cuatro conjuntos de datos (`X_train`, `y_train`, `X_test` y `y_test`) para propósitos de entrenamiento y prueba.

2. En el siguiente paso, crearemos un modelo y proporcionaremos los datos de entrenamiento (`X_train` e `y_train`) para entrenar este modelo. La elección del algoritmo ML no es tan importante para este ejercicio. Para el conjunto de datos de Iris, usaremos el algoritmo SVC con parámetros predeterminados. Algunos ejemplos de código de Python son los siguientes:

```
#iris_build_svm_model.py (#2)
# hacer predicciones
modelo = SVC(gamma='auto')
modelo.fit(X_tren, y_tren)
predicciones = modelo.predecir(X_test)
```

446 Python y aprendizaje automático

Para entrenar nuestro modelo, usamos el método de ajuste . En la siguiente declaración, hicimos predicciones basadas en los datos de prueba (`X_test`). Estas predicciones se utilizarán para evaluar el rendimiento de nuestro modelo entrenado.

3. Finalmente, las predicciones se evaluarán con los resultados esperados, según los datos de la prueba (`y_test`), utilizando la `precisión_puntuación` y `clasificación_informe` funciones de la biblioteca scikit-learn, de la siguiente manera:

```
#iris_build_svm_model.py (#3)
# evaluación de predicciones
imprimir (puntuación de precisión (y_test, predicciones))
print(classification_report(y_test, predicciones))
```

La salida de la consola de este programa es la siguiente:

```
0.9666666
Iris-setosa 1,00 1,00 1,00 11
Iris-versicolor 1,00 0,92 0,96 13
Iris-virginica 0,86 1,00 0,92 6

precisión 0,97 30
promedio macro 0,95 0,97 0,96 30
promedio ponderado 0,97 0,97 0,97 30
```

El alcance de precisión es muy alto (0,966), lo que indica que el modelo puede predecir la planta de Iris con una precisión de casi el 96 % para los datos de prueba. El modelo está haciendo un excelente trabajo para **Iris-setosa** e **Iris-versicolor**, pero solo un trabajo decente (86 % de precisión) en el caso de **Iris-virginica**. Hay varias formas de mejorar el rendimiento de nuestro modelo, todas las cuales discutiremos en la siguiente sección.

Evaluación de un modelo mediante validación cruzada y ajuste fino de hiperparámetros

Para el modelo de muestra anterior, mantuvimos el proceso de capacitación simple con el fin de aprender los pasos principales para crear un modelo de ML. Para implementaciones de producción, no podemos confiar en un conjunto de datos que solo contiene 150 registros. Además, debemos evaluar el modelo para obtener las mejores predicciones utilizando técnicas como las siguientes:

- **Validación cruzada de k-fold:** en el modelo anterior, mezclamos los datos antes de dividirlos en conjuntos de datos de entrenamiento y prueba mediante el método Holdout. Debido a esto, el modelo puede darnos resultados diferentes cada vez que lo entrenamos, dando como resultado un modelo inestable. No es trivial seleccionar datos de entrenamiento de un pequeño conjunto de datos que contiene 150 registros ya que, en nuestro caso, eso puede representar verdaderamente los datos de un sistema o entorno del mundo real. Para hacer que nuestro modelo anterior sea más estable con un conjunto de datos pequeño, el enfoque recomendado es la validación cruzada k-fold. Este enfoque se basa en dividir nuestro conjunto de datos en k pliegues o rebanadas. La idea es usar segmentos $k-1$ para entrenamiento y utilizar el segmento k -ésimo para evaluar o probar. Este proceso se repite hasta que usamos cada segmento de datos para fines de prueba. Esto es equivalente a repetir el método de exclusión k veces usando los diferentes segmentos de datos para la prueba.

Para profundizar más, debemos dividir todo nuestro conjunto de datos o conjunto de datos de entrenamiento en cinco segmentos, digamos $k=5$, para una validación cruzada de 5 veces. En la primera iteración, podemos usar el primer segmento (20 %) para probar y los cuatro segmentos restantes (80 %) para entrenar. En la segunda iteración, podemos usar el segundo segmento para probar y los cuatro segmentos restantes para entrenar, y así sucesivamente. Podemos evaluar el modelo para los cinco conjuntos de datos de entrenamiento posibles y seleccionar el mejor modelo al final. El esquema de selección de datos para entrenamiento y prueba se muestra aquí:

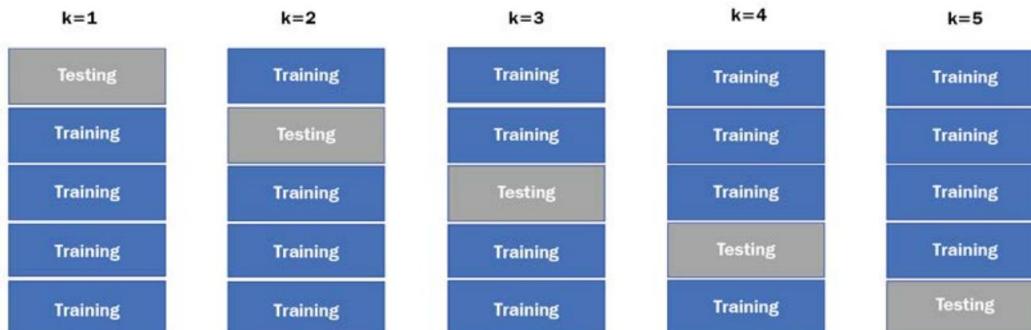


Figura 13.5 – Esquema de validación cruzada para cinco segmentos de datos

448 Python y aprendizaje automático

La precisión de la validación cruzada se calcula tomando la precisión promedio de cada modelo que construimos en cada iteración, según el valor de k.

- **Optimización de hiperparámetros:** en el ejemplo de código anterior, usamos el algoritmo de aprendizaje automático con parámetros predeterminados. Cada algoritmo de aprendizaje automático viene con muchos hiperparámetros que se pueden ajustar para personalizar el modelo, según el conjunto de datos. Es posible que los estadísticos establezcan algunos parámetros manualmente mediante el análisis de la distribución de datos, pero es tedioso analizar el impacto de una combinación de estos parámetros. Existe la necesidad de evaluar nuestro modelo utilizando diferentes valores de estos hiperparámetros, lo que puede ayudarnos a seleccionar la mejor combinación de hiperparámetros al final. Esta técnica se denomina ajuste fino u optimización de los hiperparámetros.

Los hiperparámetros de validación cruzada y ajuste fino son tediosos de implementar, incluso a través de la programación. La buena noticia es que la biblioteca scikit-learn viene con herramientas para lograr estas evaluaciones en un par de líneas de código Python. La biblioteca scikit-learn ofrece dos tipos de herramientas para esta evaluación: GridSearchCV y RandomizedSearchCV. A continuación, hablaremos de cada una de estas herramientas.

GridSearchCV

La herramienta GridSearchCV evalúa cualquier modelo dado utilizando el enfoque de validación cruzada para todas las combinaciones posibles de valores proporcionados para los hiperparámetros. Cada combinación de valores de hiperparámetros se evaluará mediante la validación cruzada en segmentos de conjuntos de datos.

En el siguiente ejemplo de código, usaremos la clase GridSearchCV de la biblioteca scikit-learn para evaluar el modelo SVC para una combinación de parámetros C y gamma . la c El parámetro es un parámetro de regularización que gestiona el equilibrio entre tener un error de entrenamiento bajo y un error de prueba bajo. Un valor más alto de C significa que podemos aceptar una mayor cantidad de errores. Usaremos 0.001, 0.01, 1, 5, 10 y 100 como valores para C. El parámetro gamma se usa para definir los hiperplanos no lineales o las líneas no lineales para la clasificación. Cuanto mayor sea el valor de gamma, el modelo puede intentar ajustar más datos agregando más curvatura o curva al hiperplano o la línea. Usaremos valores como 0.001, 0.01, 1, 5, 10 y 100 para gamma también. El código completo de GridSearchCV es el siguiente:

#iris_eval_svc_model.py (parte 1 de 2)

```
de sklearn.model_selection import train_test_split  
de sklearn.model_selection importar  
GridSearchCV,RandomizedSearchCV  
desde sklearn.datasets importar load_iris
```

```
desde sklearn.svm importar SVC
iris = cargar_iris()
X = iris.datos
y = iris.objetivo
X_tren, X_prueba, y_tren, y_prueba=tren_prueba_dividir
(X,y,tamaño_prueba=0.2)
parámetros = {"C": [0.001, 0.01, 1, 5, 10, 100],
"gamma": [0.001, 0.01, 0.1, 1, 10, 100]}
modelo=SVC()
grid_cv=GridSearchCV(modelo, parámetros, cv=5)
grid_cv.fit(tren_X,tren_y)
print(f"GridSearch- mejor parámetro:{grid_cv.best_params_}")
print(f"GridSearch- precisión: {grid_cv.best_score_}")
print(classification_report(y_test,
grid_cv.best_estimator_.predict(X_test)))
```

En este ejemplo de código, es necesario resaltar los siguientes puntos:

- Cargamos los datos directamente desde la biblioteca scikit-learn con fines ilustrativos.
También puede usar el código anterior para cargar los datos desde un archivo local.
- Es importante definir el diccionario de parámetros para ajustar hiperparámetros como primer paso.
Establecemos los valores para los parámetros C y gamma en este diccionario.
- Ponemos cv=5. Esto evaluará cada combinación de parámetros mediante el uso de validación cruzada en los cinco sectores.

La salida de este programa nos dará la mejor combinación de C y gamma y la precisión del modelo con validación cruzada. La salida de la consola para los mejores parámetros y la precisión del mejor modelo es la siguiente:

GridSearch- mejor parámetro: {'C': 5, 'gamma': 0.1}

GridSearch- precisión: 0.9833333333333334

Mediante la evaluación de diferentes combinaciones de parámetros y el uso de la validación cruzada con GridSearchCV, la precisión general del modelo mejoró del 96 % al 98 %, en comparación con los resultados que observamos sin la validación cruzada y el ajuste fino de los hiperparámetros. El informe de clasificación (que no se muestra en el resultado del programa) muestra que la precisión para los tres tipos de plantas es del 100 % para nuestros datos de prueba. Sin embargo, esta herramienta no es factible de usar cuando tenemos una gran cantidad de valores de parámetros con un gran conjunto de datos.

RandomizedSearchCV

En el caso de la herramienta RandomizedSearchCV , solo evaluamos un modelo para valores de hiperparámetros seleccionados al azar en lugar de todas las diferentes combinaciones. Podemos proporcionar los valores de los parámetros y el número de iteraciones aleatorias para realizar como entrada. La herramienta RandomizedSearchCV seleccionará aleatoriamente la combinación de parámetros según el número de iteraciones proporcionadas. Esta herramienta es útil cuando se trata de un gran conjunto de datos y cuando son posibles muchas combinaciones de parámetros/valores. Evaluar todas las combinaciones posibles para un gran conjunto de datos puede ser un proceso muy largo que requiere muchos recursos informáticos.

El código de Python para usar RandomizedSearchCV es el mismo que para el Herramienta GridSearchCV , excepto por las siguientes líneas adicionales de códigos:

```
#iris_eval_svc_model.py (parte 2 de 2)
rand_cv=RandomizedSearchCV(modelo, parámetros, n_iter = 5, cv=5)
rand_cv.fit(tren_x,tren_y)
print(f" RandomizedSearch - mejor parámetro:
{rand_cv.best_params_}")
print(f" Búsqueda aleatoria - precisión:
{rand_cv.best_score_}")
```

Dado que definimos n_iter=5, RandomizedSearchCV seleccionará solo cinco combinaciones de los parámetros C y gamma y evaluará el modelo en consecuencia.

Cuando se ejecute esta parte del programa, obtendremos una salida similar a la siguiente:

```
RandomizedSearch- mejor parámetro: {'gamma': 10, 'C': 5}
Búsqueda aleatoria: precisión: 0,9333333333333333
```

Tenga en cuenta que puede obtener un resultado diferente porque esta herramienta puede seleccionar diferentes valores de parámetros para la evaluación. Si aumentamos el número de iteraciones (n_iter) para el objeto RandomizedSearchCV , observaremos una mayor precisión en la salida. Si no configuramos n_iter, ejecutaremos la evaluación para todas las combinaciones, lo que significa que obtendremos el mismo resultado que proporciona GridSearchCV .

Como podemos ver, la mejor combinación de parámetros seleccionada por la herramienta GridSearchCV es diferente a la seleccionada por la herramienta RandomizedSearchCV . Esto era de esperar porque ejecutamos las dos herramientas para un número diferente de iteraciones.

Esto concluye nuestra discusión sobre la creación de un modelo de ML de muestra con la biblioteca scikit-learn. Cubrimos los pasos y conceptos básicos que se requieren para construir y evaluar dichos modelos. En la práctica, también escalamos los datos para la normalización. Este escalado se puede lograr mediante el uso de clases de escalador integradas en la biblioteca scikit-learn, como StandardScaler, o mediante la creación de nuestra propia clase de escalador. La operación de escalado es una operación de transformación de datos y se puede combinar con la tarea de entrenamiento del modelo en una sola canalización. scikit-learn admite la combinación de varias operaciones o tareas como una canalización mediante la clase Pipeline . La clase Pipeline también se puede usar directamente con las herramientas RandomizedSearchCV o GridSearchCV . Puede obtener más información sobre cómo usar escaladores y canalizaciones con la biblioteca scikit-learn leyendo la documentación en línea de la biblioteca scikit-learn (https://scikit-learn.org/stable/guia_usuario.html).

En la siguiente sección, discutiremos cómo guardar un modelo en un archivo y restaurar un modelo desde un archivo.

Guardar un modelo de ML en un archivo

Cuando hemos evaluado un modelo y seleccionado el mejor según nuestro conjunto de datos, el siguiente paso es implementar este modelo para futuras predicciones. Este modelo se puede implementar como parte de cualquier aplicación de Python, como aplicaciones web, Flask o Django, o se puede usar como un microservicio o incluso como una función en la nube. La verdadera pregunta es cómo transferir el objeto modelo de un programa a otro. Hay un par de bibliotecas, como pickle y joblib , que se pueden usar para serializar un modelo en un archivo. Luego, el archivo se puede usar en cualquier aplicación para cargar el modelo nuevamente en Python y hacer predicciones usando el método de predicción del objeto del modelo .

Para ilustrar este concepto, guardaremos el modelo ML que creamos en uno de los ejemplos de código anteriores (por ejemplo, el objeto modelo en iris_build_svm_model.py program) a un archivo llamado model.pkl. En el siguiente paso, cargaremos el modelo desde este archivo usando la biblioteca pickle y haremos una predicción usando nuevos datos para emular el uso de un modelo en cualquier aplicación. El código de ejemplo completo es el siguiente:

```
#iris_save_load_predict_model.py #model está  
creando usando el código en #iris_build_svm_model.py  
#guardar el modelo en un archivo  
con open("model.pkl", 'wb') como archivo:  
pickle.dump(modelo, archivo)  
  
#cargar el modelo desde un archivo (en otra aplicación)  
con open("model.pkl", 'rb') como archivo:
```

452 Python y aprendizaje automático

```
modelo_cargado = pickle.load(archivo)
x_nuevo = [[5.6, 2.6, 3.9, 1.2]]
y_nuevo = modelo_cargado.predecir(x_nuevo)
imprimir("X=%s, Predicho=%s" % (x_nuevo[0], y_nuevo[0]))
```

El uso de la biblioteca joblib es más simple que la biblioteca pickle, pero puede requerir que instale esta biblioteca si no se instaló como una dependencia de scikit-learn. El siguiente código de ejemplo muestra el uso de la biblioteca joblib para guardar nuestro mejor modelo, según la evaluación de la herramienta GridSearchCV que hicimos en la sección anterior, y luego cargar el modelo desde el archivo:

```
#iris_save_load_predict_gridmodel.py #grid_cv se
crea y entrena usando el código en #iris_eval_svm_model.py

joblib.dump(grid_cv.best_estimator_, "modelo.joblib")
modelo_cargado = joblib.load("modelo.joblib")
x_nuevo = [[5.6, 2.5, 3.9, 1.1]]
y_nuevo = modelo_cargado.predecir(x_nuevo)
imprimir("X=%s, Predicho=%s" % (x_nuevo[0], y_nuevo[0]))
```

El código de la biblioteca joblib es conciso y simple. La parte de predicción del código de ejemplo es la misma que en el ejemplo de código anterior para la biblioteca pickle.

Ahora que hemos aprendido cómo se puede guardar el modelo en un archivo, podemos llevar el modelo a cualquier aplicación para su implementación e incluso a una plataforma en la nube, como GCP AI Platform. Discutiremos cómo implementar nuestro modelo ML en una plataforma GCP en la siguiente sección.

Implementar y predecir un modelo de ML en GCP Nube

Los proveedores de nube pública ofrecen varias plataformas de IA para entrenar modelos integrados, así como sus modelos personalizados, para implementar modelos para predicciones. Google ofrece la plataforma **Vertex AI** para casos de uso de ML, mientras que Amazon y Azure ofrecen **Amazon SageMaker** y servicios **Azure ML**, respectivamente. Seleccionamos Google porque asumimos que ha configurado una cuenta con GCP y que ya está familiarizado con los conceptos básicos de GCP. GCP ofrece su AI Platform, que es parte de Vertex AI Platform, para entrenar e implementar sus modelos ML a escala. GCP AI Platform admite bibliotecas como scikit-learn, TensorFlow y XGBoost. En esta sección, exploraremos cómo implementar nuestro modelo ya entrenado en GCP y luego predeciremos el resultado en función de ese modelo.

Google AI Platform ofrece su servidor de predicción (nodo de cómputo) a través de un punto final global (ml.googleapis.com) o a través de un punto final regional (<región>-ml.googleapis.com). Se recomienda el extremo de la API global para las predicciones por lotes, que están disponibles para TensorFlow en Google AI Platform. El extremo regional ofrece protección adicional contra interrupciones en otras regiones. Usaremos un punto final regional para implementar nuestro modelo de ML de muestra.

Antes de comenzar a implementar un modelo en GCP, necesitaremos tener un proyecto de GCP. Podemos crear un nuevo proyecto de GCP o usar un proyecto de GCP existente que hayamos creado para ejercicios anteriores. Los pasos para crear un proyecto de GCP y asociarle una cuenta de facturación se analizaron en el Capítulo 9, Programación en Python para la nube. Una vez que tengamos un proyecto de GCP listo, podemos implementar el modelo `model.joblib`, que creamos en la sección anterior. Los pasos para desplegar nuestro modelo son los siguientes:

1. Como primer paso, crearemos un depósito de almacenamiento donde almacenaremos nuestro archivo de modelo. Podemos usar el siguiente comando de Cloud SDK para crear un nuevo depósito:

```
gsutil mb gs://<nombre del depósito>
gsutil mb gs://muasif-svc-model #Ejemplo de cubeta creada
```

2. Una vez que nuestro cubo esté listo, podemos cargar nuestro archivo modelo (`model.joblib`) a este depósito de almacenamiento con el siguiente comando del SDK de Cloud:

```
gsutil cp model.joblib gs://muasif-svc-model
```

Tenga en cuenta que el nombre de archivo del modelo debe ser `modelo.*`. Esto significa que el nombre del archivo debe ser `modelo` con una extensión como `pkl`, `joblib` o `bst`, según la biblioteca que usamos para empaquetar el modelo.

Ahora podemos iniciar un flujo de trabajo para crear un objeto modelo en AI Platform usando el siguiente comando. Tenga en cuenta que el nombre del modelo debe incluir solo caracteres alfanuméricos y guiones bajos:

```
Los modelos de gcloud ai-platform crean my_iris_model –
region=us-central1
```

3. Ahora, podemos crear una versión para nuestro modelo usando el siguiente comando:

```
Las versiones de gcloud ai-platform crean v1 \
--model=mi_modelo_de_iris\
--origin=gs://muasif-svc-modelo \
--framework=scikit-aprender \
--runtime-version=2.4 \
--python-versión=3.7 \
```

454 Python y aprendizaje automático

```
--region=us-central1 \
--máquina-tipo=n1-estándar-2
```

Los diferentes atributos de este comando son los siguientes:

- a) El atributo del modelo apuntará al nombre del modelo que creamos en el paso anterior.
- b) El atributo de origen apuntará a la ubicación del depósito de almacenamiento donde reside el archivo del modelo. Solo proporcionaremos la ubicación del directorio, no la ruta al archivo.
- c) El atributo del marco se usa para seleccionar qué biblioteca ML usar. GCP ofrece scikit-learn, TensorFlow y XGBoost.
- d) la versión de tiempo de ejecución es para la biblioteca scikit-learn en nuestro caso.
- e) la versión python se seleccionó como 3.7, que es la versión más alta ofrecida por GCP AI Platform al momento de escribir este libro.
- f) El atributo de la región se establece según la región que se seleccionó para el modelo.
- g) El atributo de tipo de máquina es opcional y se usa para indicar qué tipo de nodo de cómputo usar para la implementación del modelo. Si no se proporciona, el n1-estándar-2 se utiliza el tipo de máquina.

El comando de creación de versiones puede tardar unos minutos en implementar una nueva versión.

Una vez hecho esto, obtendremos una salida similar a la siguiente:

Usando el punto final [https://us-central1-ml.googleapis.com/]

**Creando la versión (esto puede tardar unos minutos)...
listo.**

4. Para verificar si nuestro modelo y versión se implementaron correctamente, podemos usar el comando describe en el contexto de las versiones , como se muestra aquí:

**Las versiones de gcloud ai-platform describen v1:
modelo=mi_modelo_de_iris**

5. Una vez que nuestro modelo se implementó con su versión, podemos usar nuevos datos para predecir el resultado utilizando el modelo que implementamos en Google AI Platform. Para realizar pruebas, agregamos un par de registros de datos, diferentes del conjunto de datos original, en un archivo JSON (input.json) , de la siguiente manera:

[5.6, 2.5, 3.9, 1.1]

[3.2, 1.4, 3.0, 1.8]

Podemos usar el siguiente comando de Cloud SDK para predecir el resultado en función de los registros dentro del archivo input.json , de la siguiente manera:

```
gcloud ai-platform predict --model my_iris_model --version v1 --json-  
instances input.json
```

La salida de la consola mostrará la clase prevista para cada registro, así como lo siguiente:

**Usando el punto final [https://us-central1-ml.googleapis.com/]
['Iris-versicolor', 'Iris-virginica']**

Para usar el modelo implementado en nuestra aplicación (local o en la nube), podemos usar Cloud SDK o Cloud Shell, pero el enfoque recomendado es usar la API de Google AI para hacer cualquier predicción.

Con eso, hemos cubierto la implementación en la nube y las opciones de predicción para nuestro modelo ML usando Google AI Platform. Sin embargo, también puede llevar su modelo de ML a otras plataformas, como Amazon SageMaker y Azure ML para la implementación y la predicción. Puede encontrar más detalles sobre la plataforma Amazon SageMaker en <https://docs.aws.amazon.com/sagemaker/> y más detalles sobre Azure ML en <https://docs.microsoft.com/en-us/azure/machine-learning/>.

Resumen

En este capítulo, presentamos el aprendizaje automático y sus componentes principales, como conjuntos de datos, algoritmos y modelos, así como el entrenamiento y la prueba de un modelo. Esta introducción fue seguida por una discusión sobre los marcos de trabajo y las bibliotecas de aprendizaje automático populares disponibles para Python. Estos incluyen scikit-learn, TensorFlow, PyTorch y BGBoost. También discutimos las mejores prácticas para refinar y administrar los datos para entrenar modelos ML. Para familiarizarse con la biblioteca scikit-learn, construimos un modelo de ML de muestra usando el algoritmo SVC. Entrenamos el modelo y lo evaluamos utilizando técnicas como la validación cruzada de k-fold y el ajuste de hiperparámetros. También aprendimos cómo almacenar un modelo entrenado en un archivo y luego cargar ese modelo en cualquier programa con fines de predicción. Al final, demostramos cómo podemos implementar nuestro modelo ML y predecir resultados usando Google AI Platform con algunos comandos de GCP Cloud SDK.

456 Python y aprendizaje automático

Los conceptos y los ejercicios prácticos incluidos en este capítulo son adecuados para ayudar a construir la base para usar Python en proyectos de aprendizaje automático. Este conocimiento teórico y práctico es beneficioso para aquellos que buscan comenzar a usar Python para el aprendizaje automático.

En el próximo capítulo, exploraremos cómo usar Python para la automatización de redes.

Preguntas

1. ¿Qué son el aprendizaje supervisado y el aprendizaje no supervisado?
2. ¿Qué es la validación cruzada k-fold y cómo se usa para evaluar un modelo?
3. ¿Qué es RandomizedSearchCV y en qué se diferencia de GridSearchCV?
4. ¿Qué bibliotecas podemos usar para guardar un modelo en un archivo?
5. ¿Por qué los puntos finales regionales son la opción preferida sobre los puntos finales globales para Google AI?
¿Plataforma?

Otras lecturas

- Algoritmos de aprendizaje automático, por Giuseppe Bonacorso
- 40 algoritmos que todo programador debería conocer, por Imran Ahmad
- Dominar el aprendizaje automático con scikit-learn, por Gavin Hackeling
- Aprendizaje automático de Python: aprendizaje automático y aprendizaje profundo con Python, scikit-learn y TensorFlow 2, por Sebastian Raschka y Vahid Mirjalili
- Guía del usuario de scikit-learn, disponible en https://scikit-learn.org/stable/guia_usuario.html
- Las guías de Google AI Platform para entrenar e implementar modelos ML están disponibles en <https://cloud.google.com/ai-platform/docs>

respuestas

1. En el aprendizaje supervisado, proporcionamos el resultado deseado con los datos de entrenamiento. Él resultado deseado no se incluye como parte de los datos de entrenamiento para el aprendizaje no supervisado.
2. La validación cruzada es una técnica estadística que se utiliza para medir el rendimiento de un modelo de ML. En la validación cruzada k-fold, dividimos los datos en k pliegues o rebanadas. Entrenamos nuestro modelo usando los segmentos k-1 del conjunto de datos y probamos la precisión del modelo usando el segmento k-ésimo. Repetimos este proceso hasta que cada segmento k-ésimo se utilice como datos de prueba.
La precisión de validación cruzada del modelo se calcula tomando el promedio de la precisión de todos los modelos que construimos a través de k iteraciones.
3. RandomizedSearchCV es una herramienta que está disponible con scikit-learn para aplicar la funcionalidad de validación cruzada a un modelo de ML para hiperparámetros seleccionados al azar. GridSearchCV proporciona una funcionalidad similar a RandomizedSearchCV, excepto que valida el modelo para todas las combinaciones de valores de hiperparámetros que se le proporcionan como entrada.
4. Pickle y Joblib.
5. Los terminales regionales ofrecen protección adicional contra cualquier interrupción en otras regiones, y la disponibilidad de recursos informáticos es mayor para los terminales regionales que para los terminales globales.

14

Usando Python para la red Automatización

Tradicionalmente, las redes son construidas y operadas por expertos en redes, y esta sigue siendo una tendencia en la industria de las telecomunicaciones. Sin embargo, este enfoque manual de administrar y operar una red es lento y, a veces, genera costosas interrupciones de la red debido a errores humanos. Además, para obtener un nuevo servicio (como un servicio de Internet), los clientes tienen que esperar días después de realizar una solicitud de un nuevo servicio antes de que esté listo. Según la experiencia de los teléfonos inteligentes y las aplicaciones móviles, donde puede habilitar nuevos servicios y aplicaciones con solo hacer clic en un botón, los clientes esperan que el servicio de red esté listo en minutos, si no segundos. Esto no es posible con el enfoque actual de gestión de red. Los enfoques tradicionales también son a veces un obstáculo para la introducción de nuevos productos y servicios por parte de los proveedores de servicios de telecomunicaciones.

La automatización de redes puede mejorar estas situaciones al ofrecer software para automatizar la gestión y los aspectos operativos de una red. La automatización de redes ayuda a eliminar los errores humanos en la configuración de dispositivos de red y reduce significativamente los costos operativos mediante la automatización de tareas repetitivas. La automatización de la red ayuda a acelerar la prestación de servicios y permite a los proveedores de servicios de telecomunicaciones introducir nuevos servicios.

460 Uso de Python para la automatización de redes

Python es una opción popular para la automatización de redes. En este capítulo, descubriremos las capacidades de Python para la automatización de redes. Python proporciona bibliotecas como **Paramiko**, **Netmiko** y **NAPALM** que se pueden usar para interactuar con dispositivos de red. Si los dispositivos de red son administrados por un **Sistema** de administración de red (**NMS**) o un controlador de red/orquestador, Python puede interactuar con estas plataformas usando **REST** o **RESTCONF** protocolos. La automatización de la red de extremo a extremo no es posible sin escuchar los eventos en tiempo real que suceden en la red. Estos eventos de red en tiempo real o datos de transmisión en tiempo real generalmente están disponibles a través de sistemas como **Apache Kafka**. También exploraremos la interacción con un sistema controlado por eventos usando Python.

Cubriremos los siguientes temas en este capítulo:

- Introducción a la automatización de redes
- Interactuar con dispositivos de red
- Integración con sistemas de administración de red
- Trabajar con sistemas basados en eventos

Después de completar este capítulo, comprenderá cómo usar las bibliotecas de Python para obtener datos de un dispositivo de red y enviar datos de configuración a estos dispositivos. Estos son pasos fundamentales para cualquier proceso de automatización de red.

Requerimientos técnicos

Los siguientes son los requisitos técnicos para este capítulo:

- Debe tener Python 3.7 o posterior instalado en su computadora.
- Debe instalar Paramiko, Netmiko, NAPALM, ncclient y las solicitudes bibliotecas sobre Python.
- Debe tener acceso a uno o más dispositivos de red con el protocolo SSH.
- Debe tener acceso a un laboratorio de desarrolladores de Nokia para poder acceder al NMS de Nokia (conocido como **Plataforma de servicios de red (NSP)**).

El código de muestra para este capítulo se puede encontrar en <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter14>.

Nota IMPORTANTE

En este capítulo, necesitará acceso a dispositivos de red físicos o virtuales y sistemas de administración de red para ejecutar los ejemplos de código. Esto puede no ser posible para todos. Puede utilizar cualquier dispositivo de red con capacidades similares.

Nos centraremos más en el lado Python de la implementación y haremos que sea conveniente reutilizar el código para cualquier otro dispositivo o sistema de gestión.

Comenzaremos nuestra discusión brindando una introducción a la automatización de redes.

Introducción a la automatización de la red

La automatización de redes es el uso de tecnología y software para automatizar los procesos de gestión y puesta en funcionamiento de las redes. La palabra clave para la automatización de la red es la automatización de un proceso, lo que significa que no se trata solo de implementar y configurar una red, sino también de los pasos que se deben seguir para lograr la automatización de la red.

Por ejemplo, a veces, los pasos de automatización implican obtener la aprobación de diferentes partes interesadas antes de enviar una configuración a una red. La automatización de este paso de aprobación es parte de la automatización de la red. Por lo tanto, el proceso de automatización de la red puede variar de una organización a otra en función de los procesos internos que sigue cada organización. Esto hace que sea un desafío construir una plataforma única que pueda realizar la automatización lista para usar para muchos clientes.

Hay una cantidad significativa de esfuerzos en curso para proporcionar las plataformas necesarias de los proveedores de dispositivos de red que pueden ayudar a crear una automatización personalizada con un esfuerzo mínimo. Algunos ejemplos de dichas plataformas son Cisco **Network Services Orchestrator (NSO)**, la plataforma **Paragon Automation** de Juniper Networks y **NSP** de Nokia.

Uno de los desafíos con estas plataformas de automatización es que, por lo general, están bloqueadas por el proveedor. Esto significa que los proveedores afirman que su plataforma también puede administrar y automatizar los dispositivos de red de otros proveedores, pero el proceso para lograr la automatización de múltiples proveedores es tedioso y costoso. Por lo tanto, los proveedores de servicios de telecomunicaciones buscan automatización más allá de las plataformas del proveedor. **Python** y **Ansible** son dos lenguajes de programación populares que se utilizan para la automatización en la industria de las telecomunicaciones. Antes de pasar a la forma en que Python logra la automatización de la red, exploremos algunos méritos y desafíos de la automatización de la red.

Méritos y desafíos de la automatización de redes

Ya hemos destacado algunos méritos de la automatización de redes. Podemos resumir los méritos clave de la siguiente manera:

- **Entrega de servicios más rápida:** una entrega de servicios más rápida a nuevos clientes le permite comenzar la facturación del servicio temprano y tener más clientes satisfechos.
- **Reducción de costos operativos:** los costos operativos de una red se pueden reducir mediante la automatización de tareas repetitivas y el monitoreo de la red a través de herramientas y plataformas de automatización de ciclo cerrado.
- **Elimine los errores de humor:** la mayoría de las interrupciones de la red se deben a errores humanos. La automatización de la red puede eliminar esta causa configurando las redes utilizando plantillas estándar. Estas plantillas se evalúan y prueban en profundidad antes de ponerlas en producción.
- **Configuración de red consistente:** cuando los humanos están configurando una red, es imposible seguir plantillas y convenciones de nomenclatura consistentes, que son importantes para que el equipo de operaciones administre la red. La automatización de la red brinda consistencia en la configuración de la red, ya que configuramos la red cada vez que usamos el mismo script o plantilla.
- **Visibilidad de la red:** con herramientas y plataformas de automatización de redes, podemos tener acceso a las capacidades de monitoreo del rendimiento y puede visualizar nuestra red de punta a punta. La administración proactiva de la red es posible mediante la detección de picos de tráfico y la utilización intensiva de recursos antes de que provoquen cuellos de botella en el tráfico de la red.

La automatización de la red es imprescindible para la transformación digital, pero existen algunos costos y desafíos para lograrlo. Estos desafíos son los siguientes:

- **Costo:** siempre hay un costo cuando se trata de construir o personalizar el software para la automatización de redes. La automatización de la red es un viaje y se debe establecer un presupuesto de costos anualmente.
- **Resistencia humana:** en muchas organizaciones, los recursos humanos consideran la automatización de redes como una amenaza para sus trabajos, por lo que se resisten a adoptar la automatización de redes, especialmente dentro de los equipos de operaciones.
- **Estructura organizativa:** la automatización de la red aporta un **retorno real de la inversión (ROI)** cuando se utiliza en diferentes capas de red y dominios de red, como TI y dominios de red. El desafío en muchas organizaciones es que estos dominios pertenecen a diferentes departamentos y cada uno tiene sus propias estrategias y preferencias de automatización con respecto a las plataformas de automatización.

- **Seleccionar una plataforma/herramienta de automatización:** seleccionar una plataforma de automatización de proveedores de equipos de red como Cisco o Nokia, o trabajar con plataformas de automatización de terceros como HP o Accenture, no es una decisión fácil. En muchos casos, los proveedores de servicios de telecomunicaciones terminan con múltiples proveedores para construir la automatización de su red, y esto trae un nuevo conjunto de desafíos para que estos proveedores trabajen juntos.
- **Mantenimiento:** Mantener las herramientas y scripts de automatización es tan esencial como construyéndolos. Esto requiere comprar contratos de mantenimiento esenciales de proveedores de automatización o establecer un equipo interno para brindar mantenimiento a dichas plataformas de automatización.

A continuación, analizamos los casos de uso.

casos de uso

Varias tareas monótonas relacionadas con la administración de la red se pueden automatizar utilizando Python u otras herramientas. Pero los beneficios reales son automatizar aquellas tareas que son repetitivas, propensas a errores o tediosas si se realizan manualmente. Desde el punto de vista de un proveedor de servicios de telecomunicaciones, las siguientes son las principales aplicaciones de la automatización de redes:

- Podemos automatizar la configuración diaria de los dispositivos de red, como la creación de nuevas interfaces IP y servicios de conectividad de red. Lleva mucho tiempo hacer estas tareas manualmente.
- Podemos configurar reglas y políticas de firewall para ahorrar tiempo. Creación de regla de cortafuegos Las configuraciones son una actividad tediosa y cualquier error puede resultar en una pérdida de tiempo para solucionar los problemas de comunicación.
- Cuando tenemos miles de dispositivos en una red, actualizar su software es un gran desafío y, a veces, lleva de 1 a 2 años lograrlo. La automatización de la red puede acelerar esta actividad y hacer cumplir las comprobaciones previas y posteriores a la actualización convenientemente para actualizaciones sin inconvenientes.
- Podemos utilizar la automatización de la red para incorporar nuevos dispositivos de red en la red. Si el dispositivo se va a instalar en las instalaciones de un cliente, podemos ahorrar un camión rodando por automatizando el proceso de incorporación del dispositivo. Este proceso de incorporación también se conoce como **aprovisionamiento sin contacto (ZTP)**.

Ahora que hemos introducido la automatización de la red, exploremos cómo interactuar con dispositivos de red usando diferentes protocolos.

Interactuar con dispositivos de red

Python es una opción popular para la automatización de redes porque es fácil de aprender y se puede usar para integrarse con dispositivos de red directamente, así como a través de NMS. De hecho, muchos proveedores, como Nokia y Cisco, admiten tiempos de ejecución de Python en sus dispositivos de red. La opción de tiempos de ejecución de Python en el dispositivo es útil para automatizar tareas y actividades en el contexto de un solo dispositivo. En esta sección, nos centraremos en la opción de tiempo de ejecución de Python fuera del dispositivo. Esta opción nos dará la flexibilidad de trabajar con varios dispositivos a la vez.

Nota IMPORTANTE

Para todos los ejemplos de código proporcionados en esta sección, utilizaremos un dispositivo de red virtual de Cisco (IOS XR con versión 7.1.2). Para la integración con el NMS utilizaremos el sistema Nokia NSP.

Antes de trabajar con Python para que podamos interactuar con los dispositivos de red, analizaremos los protocolos que están disponibles para comunicarse con los dispositivos de red.

Protocolos para interactuar con dispositivos de red

Cuando se trata de hablar directamente con los dispositivos de red, existen varios protocolos que podemos usar, como el **Protocolo de shell seguro (SSH)**, el **Protocolo simple de administración de red (SNMP)** y la **Configuración de red (NETCONF)**. Algunos de estos protocolos funcionan uno encima del otro. A continuación se describirán los protocolos más utilizados.

SSH

SSH es un protocolo de red para la comunicación entre dos dispositivos u ordenadores de forma segura. Toda la información entre las dos entidades se cifrará antes de enviarse a un canal de transporte. Normalmente usamos un cliente SSH para conectarnos a un dispositivo de red usando el comando ssh . El cliente SSH usa el nombre de usuario del usuario del sistema operativo que inició sesión con el comando ssh :

```
ssh <ip del servidor o nombre de host>
```

Para usar un usuario diferente al usuario que inició sesión, podemos especificar el nombre de usuario, de la siguiente manera:

```
ssh nombredeusuario@<IP del servidor o nombre de host>
```

Una vez que se ha establecido una conexión SSH, podemos enviar comandos CLI para recuperar información de configuración o operativa de un dispositivo o para configurar el dispositivo.

SSH versión 2 (SSHv2) es una opción popular para interactuar con dispositivos para la administración de redes e incluso para fines de automatización.

Discutiremos cómo usar el protocolo SSH con bibliotecas de Python como Paramiko, Netmiko y NAPALM en Interacción con dispositivos de red usando protocolos basados en SSH.

sección. SSH también es un protocolo de transporte fundamental para muchos protocolos de administración de red avanzados, como NETCONF.

SNMP

Este protocolo ha sido un estándar de facto para la gestión de redes durante más de 30 años y todavía se usa mucho para la gestión de redes. Sin embargo, está siendo reemplazado por protocolos más avanzados y escalables como NETCONF y gNMI. SNMP se puede usar tanto para la configuración de la red como para el monitoreo de la red, pero es más popular para el monitoreo de la red. En el mundo actual, se considera un protocolo heredado que se introdujo a fines de la década de 1980, únicamente para la administración de redes.

El protocolo SNMP se basa en la **base de información de gestión (MIB)**, que es un modelo de dispositivo. Este modelo fue construido utilizando un lenguaje de modelado de datos llamado **Estructura de información de gestión (SMI)**.

NETCONF

El protocolo NETCONF, que fue presentado por el **Grupo de Trabajo de Ingeniería de Internet (IETF)**, se considera un sucesor de SNMP. NETCONF se usa principalmente para configurar dispositivos de red y se espera que sea compatible con todos los dispositivos de red nuevos. NETCONF se basa en cuatro capas:

- **Contenido:** esta es una capa de datos que se basa en el modelado YANG. Cada dispositivo ofrece varios modelos de YANG para los distintos módulos que ofrece. Estos modelos se pueden explorar en <https://github.com/YangModels/yang>.
- **Operaciones:** Las operaciones NETCONF son acciones o instrucciones que se envían desde un cliente NETCONF a un servidor NETCONF (también llamado **Agente NETCONF**). Estas operaciones están envueltas en los mensajes de solicitud y respuesta. Ejemplos de operaciones NETCONF son get, get-config, edit-config y delete-config.
- **Mensajes:** estos son mensajes de llamada a **procedimiento remoto (RPC)** que se intercambian entre los clientes NETCONF y el agente NETCONF. Las operaciones NETCONF y los datos que están codificados como XML se envuelven dentro de los mensajes RPC.

- **Transporte:** esta capa proporciona una ruta de comunicación entre un cliente y un servidor.

Los mensajes NETCONF pueden usar NETCONF sobre SSH o NETCONF sobre TLS con la opción de certificado SSL.

El protocolo NETCONF se basa en mensajes XML que se han intercambiado a través del protocolo SSH utilizando el puerto 830 como puerto predeterminado. Por lo general, hay dos tipos de bases de datos de configuración que son administradas por dispositivos de red. El primer tipo se llama el **funcionamiento** base de datos, que representa la configuración activa en un dispositivo, incluidos los datos de operación. Esta es una base de datos obligatoria para cada dispositivo. El segundo tipo se conoce como el **candidato**. base de datos, que representa la configuración candidata antes de que pueda enviarse a la base de datos en ejecución. Cuando existe una base de datos candidata, no se permite realizar cambios de configuración directamente en la base de datos en ejecución.

Discutiremos cómo trabajar con NETCONF usando Python en la sección Interacción con dispositivos de red usando NETCONF.

RESTCONF

RESTCONF es otro estándar IETF que ofrece un subconjunto de la funcionalidad NETCONF utilizando la interfaz RESTful. En lugar de usar llamadas RPC de NETCONF con codificación XML, RESTCONF ofrece llamadas REST basadas en HTTP/HTTPS, con la opción de usar mensajes XML o JSON. Si los dispositivos de red ofrecen la interfaz RESTCONF, podemos usar métodos HTTP (GET, PATCH, PUT, POST y DELETE) para la administración de la red. Cuando RESTCONF se usa para la automatización de redes, debemos entender que proporciona una funcionalidad NETCONF limitada sobre HTTP/HTTPS. Las operaciones de NETCONF, como confirmaciones, reverisiones y bloqueo de configuración, no se admiten a través de RESTCONF.

gRPC/gNMI

gNMI es una **interfaz de administración de red (NMI)** de gRPC. gRPC es una llamada de procedimiento remoto desarrollada por Google para la recuperación de datos de baja latencia y altamente escalable. El protocolo gRPC se desarrolló originalmente para clientes móviles que querían comunicarse con servidores en la nube con estrictos requisitos de latencia. El protocolo gRPC es muy eficiente para transportar datos estructurados a través de **búferes de protocolo (Protobufs)**, que es un componente clave de este protocolo. Al usar Protobufs, los datos se empaquetan en un formato binario en lugar de un formato textual como JSON o XML. Este formato no solo reduce el tamaño de los datos, sino que es muy eficiente para serializar y deserializar datos en comparación con JSON o XML. Además, los datos se transportan utilizando HTTP/2 en lugar de HTTP 1.1.

HTTP/2 ofrece tanto el modelo de solicitud-respuesta como el modelo de comunicación bidireccional. Este modelo de comunicación bidireccional hace posible que los clientes abran conexiones de larga duración que aceleran significativamente el proceso de transferencia de datos. Estas dos tecnologías hacen que el protocolo gRPC sea de 7 a 10 veces más rápido que la API REST.

gNMI es una implementación específica del protocolo gRPC para fines de administración de redes y aplicaciones de telemetría. También es un protocolo basado en modelos YANG como NETCONF y ofrece muy pocas operaciones en comparación con NETCONF. Estas operaciones incluyen Get, Set y Subscribe. gNMI se está volviendo más popular para la recopilación de datos de telemetría que para la administración de redes. La razón principal de esto es que no brinda tanta flexibilidad como NETCONF para la configuración de la red, pero es un protocolo optimizado cuando se trata de recopilar datos de un sistema remoto, especialmente en tiempo real o casi en tiempo real.

A continuación, analizaremos las bibliotecas de Python para interactuar con dispositivos de red.

Interactuar con dispositivos de red utilizando bibliotecas de Python basadas en SSH

Hay varias bibliotecas de Python disponibles para interactuar con dispositivos de red mediante SSH.

Paramiko, Netmiko y NAPALM son tres bibliotecas populares que están disponibles y las exploraremos en las siguientes subsecciones. Comenzaremos con Paramiko.

Paramiko

La biblioteca Paramiko es una abstracción del protocolo SSH v2 en Python e incluye tanto Funcionalidad del lado del servidor y del lado del cliente. Aquí solo nos centraremos en las capacidades del lado del cliente de la biblioteca Paramiko.

Cuando interactuamos con un dispositivo de red, intentamos obtener datos de configuración o presionamos una nueva configuración para ciertos objetos. Lo primero se logra con show tipos de comandos CLI, según el sistema operativo del dispositivo, mientras que este último puede requerir un modo especial para ejecutar los comandos CLI de configuración. Estos dos tipos de comandos se manejan de manera diferente cuando se trabaja con bibliotecas de Python.

Obtener la configuración del dispositivo

Para conectarse a un dispositivo de red (escuchando como un servidor SSH) usando la biblioteca Paramiko, debemos usar una instancia de la clase paramiko.SSHClient o usar directamente una clase paramiko.Transport de bajo nivel . La clase Transport ofrece métodos de bajo nivel que brindan control directo sobre la comunicación basada en sockets. La clase SSHClient es una clase contenedora y utiliza la clase Transport bajo el capó para administrar una sesión, con un servidor SSH implementado en un dispositivo de red.

468 Uso de Python para la automatización de redes

Podemos usar la biblioteca Paramiko para establecer una conexión con un dispositivo de red (Cisco IOS XR, en nuestro caso) y ejecutar un comando show (show ip int brief, en nuestro caso) así:

```
#show_cisco_int_pmk.py importar
paramiko

anfitrión='HOST_ID'
puerto=22
nombre de usuario = 'xxx'
contraseña = 'xxxxxx'

#comando cisco ios para obtener una lista de interfaces IP cmd= 'show ip int brief
\n'

def principal():

prueba: ssh
= paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.
AutoAddPolicy())
ssh.connect(host, puerto, nombre de usuario, contraseña)

stdin, stdout, stderr = ssh.exec_command(cmd) output_lines = stdout.readlines()
respuesta = ".join(output_lines) print(response)

por fin:
ssh.cerrar()

si __nombre__ == '__principal__': principal()
```

Los puntos clave de este ejemplo de código son los siguientes:

- Creamos una instancia SSHClient y abrimos una conexión con el servidor SSH. • Dado que no estamos usando la clave de host para nuestra conexión SSH, aplicamos el set_ faltando el método host_key_policy para evitar advertencias o errores.
- Una vez que se estableció la conexión SSH, enviamos nuestro comando show, show ip int brief, a la máquina host mediante el transporte SSH y recibimos el resultado del comando como una respuesta SSH.
- La salida de este programa es una tupla de objetos stdin, stdout y stderr . Si nuestro comando se ejecuta con éxito, recuperaremos la salida del objeto stdout .

El resultado de este programa, cuando se ejecuta en un dispositivo Cisco IOS XR, es el siguiente:

```
Iun 19 de julio 12:03:41.631 UTC
Protocolo de estado de la dirección IP de la interfaz
Loopback0 10.180.180.10 Arriba Arriba
GigabitEthernet0/0/0/0 10.1.10.2 Arriba Arriba
GigabitEthernet0/0/0/0.100 sin asignar Arriba Abajo
GigabitEthernet0/0/0/1 sin asignar Arriba Arriba
GigabitEthernet0/0/0/1.100 150.150.150.1 Arriba Arriba
GigabitEthernet0/0/0/2 apagado no asignado
```

Si está ejecutando este programa en otro tipo de dispositivo, debe cambiar el comando que se ha configurado como la variable cmd , según el tipo de su dispositivo.

La biblioteca Paramiko proporciona control de bajo nivel sobre la comunicación de la red, pero a veces puede ser peculiar debido a la implementación no estándar o incompleta del protocolo SSH por parte de muchos dispositivos de red. Si enfrenta desafíos al usar Paramiko con algunos dispositivos de red, no es usted o Paramiko, sino la forma en que el dispositivo espera que se comunique con él. Un canal de transporte de bajo nivel puede resolver estos problemas, pero esto requiere un poco de programación compleja. Netmiko viene al rescate aquí.

Netmiko

Netmiko es una biblioteca abstracta para la gestión de redes que se basa en la biblioteca Paramiko. Elimina los desafíos de Paramiko al tratar cada red dispositivo de manera diferente. Netmiko usa Paramiko debajo del capó y oculta muchos detalles de comunicación a nivel de dispositivo. Netmiko admite varios dispositivos de diferentes proveedores, como Cisco, Arista, Juniper y Nokia.

Obtener la configuración del dispositivo

Para conectarse a un dispositivo de red usando el tipo show de los comandos CLI, debemos establecer una definición de tipo de dispositivo que se usa para conectarse con el dispositivo de red de destino. Esta definición de tipo de dispositivo es un diccionario que debe incluir el tipo de dispositivo, la IP del host o el **nombre de dominio completo (FQDN) del dispositivo**, el nombre de usuario y la contraseña para conectarse con el dispositivo. Podemos establecer el número de puerto para la conexión SSH si la máquina de destino está escuchando en un puerto que no sea el 22. El siguiente código se puede usar para ejecutar el mismo comando show que ejecutamos con la biblioteca Paramiko:

```
#show_cisco_int_nmk.py
de netmiko importar ConnectHandler

cisco_rtr = {
    "tipo_dispositivo": "cisco_ios",
    "anfitrión": "HOST_ID",
    "nombre de usuario": "xxx",
    "contraseña": "xxxxxxxx",
    "#"factor_de_retraso_global": 2,
}

def principal():
    comando = "mostrar ip int breve"
    con ConnectHandler(**cisco_rtr) como net_connect:
        imprimir (net_connect.find_prompt())
        imprimir (net_connect. habilitar ())
        salida = net_connect.send_command(comando)

    imprimir (salida)
```

Los puntos clave de este código de ejemplo son los siguientes:

- Creamos una conexión de red usando la clase ConnectHandler usando un administrador de contexto. El administrador de contexto administrará el ciclo de vida de la conexión.
- Netmiko ofrece un método simple llamado `find_prompt` para tomar el indicador del dispositivo de destino, que es útil para analizar la salida de muchos dispositivos de red. Esto no es necesario para el dispositivo de red Cisco IOS XR, pero lo usamos como práctica recomendada.
- Netmiko también nos permite ingresar al modo Habilitar (es un indicador de línea de comando, #) para dispositivos Cisco IOS mediante el método `enable`. Nuevamente, esto no es necesario para este ejemplo, pero es una buena práctica usarlo, especialmente en los casos en los que estamos enviando comandos CLI para la configuración como parte del mismo script de programación.
- Ejecutamos el comando `show ip int brief` usando el comando `send_command` y obtuve el mismo resultado que obtuvimos para `show_cisco_int_pmk.py` programa.

Según el ejemplo de código que compartimos para el mismo comando `show`, podemos concluir que trabajar con Netmiko es mucho más conveniente en comparación con Paramiko.

Nota IMPORTANTE

Es muy importante configurar el tipo de dispositivo correcto para obtener resultados consistentes, incluso si está utilizando dispositivos del mismo proveedor. Esto es especialmente importante cuando se usan comandos para configurar un dispositivo. Un tipo de dispositivo incorrecto puede generar errores inconsistentes.

A veces, ejecutamos comandos que requieren más tiempo para completarse que el `show` normal comandos. Por ejemplo, es posible que deseemos copiar un archivo de una ubicación a otra en un dispositivo y sabemos que esto puede demorar unos cientos de segundos para un archivo grande. De manera predeterminada, Netmiko espera casi 100 segundos para que se complete un comando. Podemos agregar un factor de retraso global como parte de la definición del dispositivo agregando una línea como la siguiente:

```
"global_delay_factor": 2
```

Esto aumentará el tiempo de espera para todos los comandos de este dispositivo en un factor de 2. Alternativamente, podemos establecer el factor de retraso para un comando individual con el comando `send_command` pasando el siguiente argumento:

```
factor_retraso=2
```

472 Uso de Python para la automatización de redes

Deberíamos agregar un factor de retraso cuando esperamos un tiempo de ejecución significativo. Cuando tengamos que agregar un factor de retraso, también deberíamos agregar otro atributo como argumento con el método `send_command`, que romperá el ciclo de espera antes si vemos un símbolo del sistema (por ejemplo, # en el caso de los dispositivos Cisco IOS). Esto se puede configurar usando el siguiente atributo:

```
expect_string=r'#'
```

Configuración de un dispositivo de red

En el siguiente ejemplo de código, proporcionaremos un código de muestra para propósitos de configuración. Configurar un dispositivo usando Netmiko es similar a ejecutar comandos `show`, ya que Netmiko se encargará de habilitar el terminal de configuración (si es necesario, según el tipo de dispositivo) y salir del terminal de configuración correctamente.

Para nuestro ejemplo de código, estableceremos una descripción de una interfaz usando Netmiko con el siguiente programa:

```
#config_cisco_int_nmk.py
de netmiko importar ConnectHandler

cisco_rtr = {
    "tipo_dispositivo": "cisco_ios",
    "anfitrión": "HOST_ID",
    "nombre de usuario": "xxx",
    "contraseña": "xxxxxx",
}

def principal():
    comandos = ["int Lo0 \"descripción custom_description\"", "commit"]

    con ConnectHandler(**cisco_rtr) como net_connect:
        salida = net_connect.send_config_set(comandos)

    imprimir (salida)
    imprimir()
```

Los puntos clave de este ejemplo de código son los siguientes:

- Para este programa, creamos una lista de tres comandos (`int <id de interfaz>`, descripción `<nueva descripción>` y confirmación). Los dos primeros comandos también se pueden enviar como un solo comando, pero los hemos mantenido separados con fines ilustrativos. El comando de confirmación se utiliza para guardar los cambios.
- Cuando enviamos un comando al dispositivo para su configuración, usamos el método `send_config_set` de la biblioteca de Netmiko para establecer una conexión con fines de configuración. La ejecución exitosa de este paso depende de la configuración correcta del tipo de dispositivo. Esto se debe a que el comportamiento del dispositivo varía de un dispositivo a otro para los comandos de configuración.
- El conjunto de tres comandos agregará o actualizará el atributo de descripción para la interfaz especificada.

No se esperará ningún resultado especial de este programa, excepto que la configuración del dispositivo solicite nuestros comandos. La salida de la consola se verá de la siguiente manera:

```
Iun 19 de julio 13:21:16.904 UTC
RP/0/RP0/CPU0:cisco(config)#int Lo0
RP/0/RP0/CPU0:cisco(config-if)#descripción descripción_personalizada
RP/0/RP0/CPU0:cisco(config-if)#commit
Iun 19 de julio 13:21:17.332 UTC
RP/0/RP0/CPU0:cisco(config-if)#
```

Netmiko ofrece muchas más funciones, pero dejaremos esto para que lo explore leyendo su documentación oficial (<https://pypi.org/project/netmiko/>). Los ejemplos de código que analizamos en esta sección se probaron con un dispositivo de red de Cisco, pero se puede usar el mismo programa cambiando el tipo de dispositivo y los comandos para cualquier otro dispositivo si el dispositivo es compatible con Netmiko.

Netmiko simplifica nuestro código para la interacción de dispositivos de red, pero seguimos ejecutando los comandos CLI para obtener la configuración del dispositivo o enviar la configuración hacia el dispositivo. La programabilidad no es fácil de hacer con Netmiko, pero otra biblioteca llamada NAPALM está ahí para ayudar.

NAPALM

NAPALM es un acrónimo de **Network Automation and Programmability Abstraction Layer with Multivendor**. Esta biblioteca proporciona el siguiente nivel de abstracción además de Netmiko al ofrecer un conjunto de funciones como una API unificada para interactuar con varios dispositivos de red. No es compatible con tantos dispositivos como Netmiko. Para la versión 3 de NAPALM, los controladores principales están disponibles para los dispositivos de red **Arista EOS**, **Cisco IOS**, **Cisco IOS-XR**, **Cisco NX-OS** y **Juniper JunOS**. Sin embargo, hay varios controladores creados por la comunidad disponibles para comunicarse con muchos otros dispositivos, como **Nokia SROS**, **Aruba AOS-CX** y **Ciena SAOS**.

Como hicimos con Netmiko, crearemos ejemplos de NAPALM para interactuar con un dispositivo de red. En el primer ejemplo, obtendremos una lista de interfaces IP, mientras que para el segundo ejemplo agregaremos o actualizaremos el atributo de descripción para una interfaz IP. Estos dos ejemplos de código realizarán las mismas operaciones que realizamos con las bibliotecas Paramiko y Netmiko.

Obtener la configuración del dispositivo

Para obtener la configuración de un dispositivo, debemos configurar la conexión a nuestro dispositivo de red.

Haremos esto en ambos ejemplos de código. Configurar una conexión es un proceso de tres pasos, como se explica aquí:

1. Para configurar una conexión, debemos obtener una clase de controlador de dispositivo basada en el tipo de dispositivo compatible. Esto se puede lograr utilizando la función `get_network_driver` de la biblioteca NAPALM.
2. Una vez que tenemos una clase de controlador de dispositivo, podemos crear un objeto de dispositivo proporcionando argumentos como la identificación del host, el nombre de usuario y la contraseña para el constructor de la clase de controlador.
3. El siguiente paso es conectarse al dispositivo utilizando el método abierto del objeto del dispositivo. Todos estos pasos se pueden implementar como código Python, como se muestra aquí:

```
desde napalm import get_network_driver driver =
get_network_driver('iosxr')
dispositivo = controlador('HOST_ID', 'xxxx', 'xxxx')
dispositivo.abierto()
```

Una vez que la conexión del dispositivo está disponible, podemos llamar a métodos como get_interfaces_ip (equivalente al comando CLI show interfaces) o get_facts (equivalente al comando CLI show version). El código completo para usar estos dos métodos es el siguiente:

```
#show_cisco_int_npm.py
desde napalm import get_network_driver import json

def principal():
    controlador = get_network_driver('iosxr') dispositivo =
    controlador('HOST_ID', 'root', 'rootroot')
    tratar:
        dispositivo.open()
        print(json.dumps(device.get_interfaces_ip(), sangría=2)) #print(json.dumps(device.get_facts(), sangría=2))

    por fin:
        dispositivo.cerrar()
```

El hecho más interesante es que la salida de este programa está en formato JSON por defecto. NAPALM convierte la salida del comando CLI en un diccionario de forma predeterminada que es fácil de consumir en Python. Aquí se muestra un extracto del resultado del ejemplo de código anterior:

```
{ "Loopback0": { "ipv4":
{ "10.180.180.180":
{ "prefix_length": 32 } } }, "MgmtEth0/
RPO/CPU0/0": { "ipv4": { "172.16.2.12":
{ "prefijo_longitud": 24 }}
```

476 Uso de Python para la automatización de redes

```
}
```

```
}
```

```
}
```

Configuración de un dispositivo de red

En el siguiente ejemplo de código, estamos usando la biblioteca NAPALM para agregar o actualizar el atributo de descripción para una interfaz IP existente:

```
#config_cisco_int_npm.py
de la importación de napalm get_network_driver
importar json

def principal():

    controlador = get_network_driver('iosxr')
    dispositivo = controlador('HOST_ID', 'xxx', 'xxxx')
    tratar:
        dispositivo.abierto()
        dispositivo.load_merge_candidate(config='interfaz Lo0 \n
descripción napalm_desc \n final\n')
        imprimir (dispositivo.compare_config())
        dispositivo.commit_config()
    por fin:
        dispositivo.cerrar()
```

Los puntos clave de este ejemplo de código son los siguientes:

- Para configurar la interfaz IP, debemos usar el método `load_merge_candidate` y pasar el mismo conjunto de comandos CLI a este método que hicimos para la configuración de la interfaz con Netmiko.
- Luego, comparamos las configuraciones antes de nuestros comandos y después de los comandos usando el método `compare_config`. Esto indica qué nueva configuración se ha agregado y qué se ha eliminado.
- Aplicamos un compromiso a todos los cambios usando el método `commit_config`.

Para este código de ejemplo, la salida mostrará el delta de cambios, así:

```
---  
+++  
@@ -47,7 +47,7 @@  
!  
!  
interfaz Loopback0  
- descripción mi descripción personalizada  
+ descripción napalm añadido nuevo desc  
dirección ipv4 10.180.180.180 255.255.255.255  
!  
interfaz MgmtEth0/RP0/CPU0/0
```

Aquí, la línea que comienza con - es una configuración que debe eliminarse; cualquier línea con + al comienzo es una nueva configuración que se agregará.

Con estos dos ejemplos de código, le mostramos un conjunto básico de características de NAPALM para un tipo de dispositivo. La biblioteca se puede utilizar para configurar varios dispositivos a la vez y puede funcionar con diferentes conjuntos de configuraciones.

En la siguiente sección, analizaremos la interacción con dispositivos de red mediante el protocolo NETCONF.

Interactuar con dispositivos de red usando NETCONF

NETCONF se creó para la gestión de redes basadas en modelos (objetos), especialmente para la configuración de redes. Cuando se trabaja con un dispositivo de red mediante NETCONF, es importante comprender dos capacidades del dispositivo, a saber:

- Puede comprender los modelos YANG de los dispositivos que tiene. tener esto el conocimiento es importante si desea enviar los mensajes en el formato correcto. Aquí hay una excelente fuente de modelos YANG de varios proveedores: <https://github.com/YangModels/yang>.
- Puede habilitar los puertos NETCONF y SSH para el protocolo NETCONF en el dispositivo de red para su dispositivo de red. En nuestro caso, usaremos un dispositivo virtual de Cisco IOS XR, como hicimos en nuestros ejemplos de código anteriores.

478 Uso de Python para la automatización de redes

Antes de iniciar cualquier actividad relacionada con la administración de la red, debemos verificar las capacidades de NETCONF del dispositivo y los detalles de la configuración de la fuente de datos de NETCONF. Para todos los ejemplos de código de esta sección, utilizaremos una biblioteca de cliente NETCONF para Python conocida como ncclient. Esta biblioteca proporciona métodos convenientes para enviar solicitudes NETCONF RPC. Podemos escribir un programa Python de muestra usando el ncclient biblioteca para obtener las capacidades del dispositivo y la configuración completa del dispositivo, de la siguiente manera:

#check_cisco_device.py

del administrador de importación ncclient

```
con manager.connect(host='device_ip', username=xxxx, password=xxxxxx,  
hostkey_verify=False) como conexión:
```

```
capacidades = []
```

```
para la capacidad en conn.server_capabilities:
```

```
capacidades.append(capacidad)
```

```
capacidades = ordenado(capacidades)
```

```
para el límite de capacidades:
```

```
imprimir (tapa)
```

```
resultado = conn.get_config(fuente="en ejecución")
```

```
imprimir (resultado)
```

El objeto administrador de la biblioteca ncclient se usa para conectarse al dispositivo mediante SSH pero para el puerto NETCONF 830 (predeterminado). Primero, obtenemos una lista de capacidades del servidor a través de la instancia de conexión y luego las imprimimos en un formato ordenado para facilitar la lectura. En la siguiente parte de este ejemplo de código, iniciamos un get-config

Operación NETCONF utilizando el método get_config de la biblioteca de clases del administrador .

La salida de este programa es muy larga y muestra todas las capacidades y la configuración del dispositivo.

Le dejamos a usted explorar la salida y familiarizarse con las capacidades de su dispositivo.

Es importante entender que el alcance de esta sección no es explicar NETCONF sino aprender a usar Python y ncclient para trabajar con NETCONF. Para lograr este objetivo, escribiremos dos ejemplos de código: uno para obtener la configuración de las interfaces del dispositivo y otro sobre cómo actualizar la descripción de una interfaz, que es lo mismo que hicimos con las bibliotecas de Python anteriores.

Obtener interfaces a través de NETCONF

En la sección anterior, aprendimos que nuestro dispositivo (Cisco IOS XR) admite interfaces mediante el uso de la implementación de **OpenConfig**, que está disponible en <http://openconfig.net/yang/interfaces?module=openconfig-interfaces>.

También podemos comprobar el formato XML de la configuración de nuestra interfaz, que recibimos como salida del método `get_config`. En este ejemplo de código, simplemente pasaremos un filtro XML con la configuración de la interfaz como argumento al método `get_config`, de la siguiente manera:

```
#show_all_interfaces.py
del administrador de importación ncclient
con manager.connect(host='device_ip', username=xxx, password='xxxx',
hostkey_verify=False) como conn:
resultado = conn.get_config("ejecutando", filter=('subárbol', '<interfaces xmlns='
"http://openconfig.net.yang/
interfaces"/>'))
imprimir (resultado)
```

La salida de este programa es una lista de interfaces. Solo mostraremos un extracto de la salida aquí con fines ilustrativos:

```
<rpc-reply mensaje-id="urn:uuid:f4553429-
ede6-4c79-aeea-5739993cacf4"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<datos>
<interfaces xmlns="http://openconfig.net/yang/interfaces">
<interfaz>
<nombre>Bucle invertido0</nombre>
<configuración>
<nombre>Bucle invertido0</nombre>
<description>Configurado por NETCONF</description>
</config>
<!--el resto de la salida se salta -->
```

480 Uso de Python para la automatización de redes

Para obtener un conjunto selectivo de interfaces, utilizaremos una versión extendida del filtro XML basado en el modelo YANG de la interfaz. Para el siguiente ejemplo de código, definiremos un filtro XML con las propiedades de nombre de las interfaces como nuestro criterio de filtrado. Dado que este filtro XML tiene más de una línea, lo definiremos por separado como un objeto de cadena. Aquí está el código de muestra con el filtro XML:

```
#show_int_config.py
del administrador de importación ncclient
# Crear plantilla de filtro para una interfaz
        """
filter_temp = <filtro>

<interfaces xmlns="http://openconfig.net/yang/interfaces">
<interfaz>
<nombre>{int_name}</nombre>
</interfaz>
</interfaces>
</filtro>"""

con manager.connect(host='device_ip', username=xxx, password='xxxx',
hostkey_verify=False) como conn:
filtro = filter_temp.format(int_name = "MgmtEth0/RP0/
CPU0/0")
resultado = m.get_config("ejecutando", filtro)
imprimir (resultado)
```

La salida de este programa será una sola interfaz (según la configuración en nuestro dispositivo) y se verá de la siguiente manera:

```
<?xml versión="1.0"?>
<rpc-reply mensaje-id="urn:uuid:c61588b3-
1fbf-4aa4-a9de-2a98727e1e15"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<datos>
<interfaces xmlns="http://openconfig.net/yang/interfaces">
<interfaz>
<nombre>MgmtEth0/RP0/CPU0/0</nombre>
<configuración>
<nombre>MgmtEth0/RP0/CPU0/0</nombre>
```

```
</config>
<ethernet xmlns="http://openconfig.net/yang/interfaces/
    ethernet">
<configuración>
<negociación automática>falso</negociación automática>
</config>
</ethernet>
<subinterfaces>
<!-- detalles de subinterfaces omitidos para ahorrar espacio -->
</subinterfaces>
</interfaz>
</interfaces>
</datos>
</rpc-respuesta>
```

También podemos definir filtros XML en un archivo XML y luego leer el contenido del archivo en un objeto de cadena en el programa Python. Otra opción es usar plantillas Jinja si planeamos usar filtros de forma extensiva.

A continuación, discutiremos cómo actualizar la descripción de una interfaz.

Actualización de la descripción de la interfaz

Para configurar un atributo de interfaz como la descripción, debemos usar el modelo YANG disponible en <http://cisco.com/ns.yang/Cisco-IOS-XR-ifmgr-cfg>.

Además, el bloque XML para configurar una interfaz es diferente al bloque XML que usamos para obtener la configuración de la interfaz. Para actualizar una interfaz, debemos usar la siguiente plantilla, que hemos definido en un archivo separado:

```
<!--config-template.xml-->
<config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
<configuraciones de interfaz xmlns="http://cisco.com/ns.yang/
    Cisco-IOS-XR-ifmgr-cfg">
<configuración-interfaz>
<activo>actuar</activo>
<nombre-interfaz>{int_name}</nombre-interfaz>
<descripción>{int_desc}</descripción>
</interface-configuration>
```

482 Uso de Python para la automatización de redes

```
</interfaz-configuraciones>
</config>
```

En esta plantilla, establecemos los marcadores de posición para las propiedades de nombre y descripción de la interfaz. A continuación, escribiremos un programa en Python que leerá esta plantilla y llamará a la operación NETCONF edit-config usando el método edit_config de la biblioteca ncclient . Esto enviará la plantilla a la base de datos de candidatos del dispositivo:

```
#config_cisco_int_desc.py
del administrador de importación ncclient

nc_template = open("config-template.xml").read()
nc_payload = nc_template.format(int_name='Loopback0',
int_desc="Configurado por NETCONF")

con manager.connect(host='device_ip', nombre de usuario=xxxx,
contraseña = xxx, hostkey_verify = Falso) como nc:
    netconf_reply = nc.edit_config(nc_payload, target="candidato")

imprimir (netconf_respuesta)
respuesta = nc.commit()
imprimir (respuesta)
```

Es importante destacar aquí dos cosas. Primero, el dispositivo Cisco IOS XR se configuró para aceptar solo la nueva configuración a través de la base de datos de candidatos. Si intentamos configurar el atributo de destino para que se ejecute, fallará. En segundo lugar, debemos llamar al método de confirmación después de la operación de edición de configuración en la misma sesión para que la nueva configuración sea operativa. El resultado de este programa serán dos respuestas OK del servidor NETCONF, de la siguiente manera:

```
<?xml versión="1.0"?>
<rpc-reply mensaje-id="urna:uuid:6d70d758-
6a8e-407d-8cb8-10f500e9f297"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <bien/>
</rpc-response>

<?xml versión="1.0"?>
<rpc-reply mensaje-id="urna:uuid:2a97916b db5f-427d-9553-
de1b56417d89"
```

```
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<bien/>
</rpc-respuesta>
```

Esto concluye nuestra discusión sobre el uso de Python para operaciones NETCONF. Cubrimos dos operaciones principales (get-config y edit-config) de NETCONF con la biblioteca ncclient .

En la siguiente sección, veremos la integración con los sistemas de administración de redes usando Python.

Integración con sistemas de gestión de red

Los sistemas de gestión de red o controladores de red son sistemas que ofrecen aplicaciones de gestión de red con **interfaces gráficas de usuario (GUI)**. Estos sistemas incluyen aplicaciones como inventario de red, aprovisionamiento de red, gestión de fallas y mediación con dispositivos de red. Estos sistemas se comunican con dispositivos de red mediante una combinación de protocolos de comunicación como SSH/NETCONF para el aprovisionamiento de redes, SNMP para alarmas y monitoreo de dispositivos, y gRPC para recopilación de datos de telemetría. Estos sistemas también ofrecen capacidades de automatización a través de sus motores de secuencias de comandos y flujo de trabajo.

El aspecto de mayor valor agregado de estos sistemas es que agregan la funcionalidad del dispositivo de red en un solo sistema (en sí mismo) y luego lo ofrecen a través de sus **North Bound Interfaces (NBI)**, que generalmente son interfaces REST o RESTCONF. Estos sistemas también ofrecen notificaciones de eventos en tiempo real como alarmas a través de un sistema basado en eventos como Apache Kafka. En esta sección, discutiremos un par de ejemplos del uso de la API REST de un NMS. Exploraremos cómo integrarse con Apache Kafka usando Python en la sección Integración con sistemas controlados por eventos.

Para trabajar con un NMS, utilizaremos un laboratorio compartido ofrecido por el portal para desarrolladores en línea de Nokia (<https://network.developer.nokia.com/>). Este laboratorio tiene algunos enruteadores IP de Nokia y un NSP. Este laboratorio compartido se ofrece de forma gratuita por un tiempo limitado (3 horas por día en el momento de escribir este libro). Se le pedirá que cree una cuenta en el portal para desarrolladores de forma gratuita. Cuando reserve un laboratorio para su uso, recibirá un correo electrónico con instrucciones sobre cómo conectarse al laboratorio, junto con los detalles de VPN necesarios. Si es ingeniero de redes y tiene acceso a cualquier otro NMS o controlador, puede usar ese sistema para los ejercicios de esta sección haciendo los ajustes apropiados.

484 Uso de Python para la automatización de redes

Para consumir una API REST de Nokia NSP, debemos interactuar con REST API Gateway, que administra varios puntos finales de API para Nokia NSP. Podemos comenzar a trabajar con REST API Gateway utilizando los servicios de ubicación, como se explica a continuación.

Uso de puntos finales de servicios de ubicación

Para comprender qué puntos finales de API están disponibles, Nokia NSP ofrece un punto final de servicios de ubicación que proporciona una lista de todos los puntos finales de API. Para consumir cualquier API REST en esta sección, usaremos la biblioteca de solicitudes de Python. La biblioteca de solicitudes es conocida por enviar solicitudes HTML a un servidor usando el protocolo HTTP, y la hemos usado en capítulos anteriores. Para obtener una lista de puntos finales de API del sistema Nokia NSP, usaremos el siguiente código de Python para invocar una API de servicios de ubicación:

```
#ubicación_servicios1.py
solicitudes de importación
carga útil = {}
encabezados = {}
url = "https://<URL NSP>/rest-gateway/rest/api/v1/ubicación/
      servicios"
resp = solicitudes.solicitud("OBTENER", url, encabezados=encabezados,
                               datos = carga útil)
imprimir(resp.texto)
```

Esta respuesta de API le proporcionará unas pocas docenas de puntos finales de API en formato JSON.

Puede consultar la documentación en línea de Nokia NSP en <https://network.developer.nokia.com/api-documentation/> para comprender cómo funciona cada API.

Si estamos buscando un punto final de API específico, podemos cambiar el valor de la URL variable en el ejemplo de código antes mencionado, de la siguiente manera:

```
url = "https://<URL NSP>/rest-gateway/rest/api/v1/ubicación/
      servicios/puntos finales?endPoint=/v1/auth/token"
```

Al usar esta nueva URL de API, intentamos encontrar un punto final de API para el token de autorización (/v1/auth/token). El resultado del ejemplo de código con esta nueva URL es el siguiente:

```
{ "respuesta": {
    "estado": 0,
    "Filalinicio": 0,
    "fila final": 0,
```

```
"filastotales": 1,  
"datos":  
{ "puntos finales": [  
  
    { "docUrl": "https://<NSP_URL>/rest-gateway/api-docs#/!  
    auténtico..",  
    "efectiveUrl": "https://<NSP_URL>/rest-gateway/rest/api", "operación": "[POST]" }  
  
]  
},  
"errores": nulo  
  
}}
```

Tenga en cuenta que no se requiere autenticación para usar la API de servicios de ubicación. Sin embargo, necesitaremos un token de autenticación para llamar a cualquier otra API. En la siguiente sección, aprenderemos cómo obtener un token de autenticación.

Obtener un token de autenticación

Como siguiente paso, usaremos la URL efectiva del resultado del ejemplo de código anterior para obtener el token de autenticación. Esta API requiere que pasemos la codificación base64 del nombre de usuario y la contraseña como el atributo de Autorización del encabezado HTTP. El código de Python para llamar a esta API de autenticación es el siguiente:

```
#get_token.py  
solicitudes de importación  
desde base64 importar b64encode  
importar json  
  
#obteniendo la codificación base64  
mensaje = 'nombre de usuario' + ':' + 'contraseña'  
mensaje_bytes = mensaje.encode('UTF-8')  
basic_token = b64encode(mensaje_bytes)  
carga útil = json.dumps({  
    "grant_type": "client_credentials"  
})
```

486 Uso de Python para la automatización de redes

```
encabezados = {
    'Tipo de contenido': 'aplicación/json',
    'Autorización': 'Básico {}'.format(str(basic_token,'UTF-8'))
}
url = "https://< URL DEL SERVIDOR NSP >/rest-gateway/rest/api/v1/auth/
simbólico"
resp = solicitudes.solicitud ("POST", url, encabezados = encabezados,
    datos = carga útil)
token = resp.json()["access_token"]
imprimir (respetar)

Al ejecutar este código de Python, obtendremos un token para uno
hora que se utilizará para cualquier API de NSP.

{
    "token_acceso": "VEtOLVNBTXFhZDQ3MzE5ZjQtNWUxZjQ0YjNI",
    "refresh_token": "UkVUS04tU0FnCWF5ZIMTmQ0ZTA5MDNIOTY=",
    "token_type": "Portador",
    "expires_in": 3600
}
```

También hay un token de actualización disponible que se puede usar para actualizar el token antes de que caduque. Una buena práctica es actualizar su token cada 30 minutos. Podemos actualizar nuestro token usando la misma API de token de autenticación, pero enviar los siguientes atributos en el cuerpo de la solicitud HTTP:

```
carga útil = json.dumps({
    "grant_type": "refresh_token",
    "refresh_token": "UkVUS04tU0FnCWF5ZIMTmQ0ZTA5MDNIOTY="
})
```

Otra buena práctica es revocar el token cuando ya no sea necesario. Esto se puede lograr utilizando el siguiente punto final de la API:

```
url = "https://<URL NSP>/rest-gateway/rest/api/v1/auth/
revocación"
```

Obtener dispositivos de red y un inventario de interfaz

Una vez que hayamos recibido el token de autenticación, podemos usar la API REST para obtener datos configurados, así como para agregar una nueva configuración. Comenzaremos con un ejemplo de código simple que obtendrá una lista de todos los dispositivos de red en una red administrada por NSP.

En este ejemplo de código, usaremos el token que ya hemos recuperado usando la API de token:

```
#get_network_devices.py
solicitudes de importación
cargar={}
encabezados = {
    'Autorización': 'Portador {token}'.format(token)
}
url = "https://{{NSP_URL}}:8544/NetworkSupervision/rest/api/v1/
      elementos de red"
respuesta = solicitudes.solicitud("OBTENER", url, encabezados=encabezados,
                                    datos=cargar)
imprimir (respuesta.texto)
```

El resultado de este programa será una lista de dispositivos de red con atributos de dispositivos de red.

Omitimos mostrar la salida debido a que se trata de un gran conjunto de datos.

En el siguiente ejemplo de código, mostraremos cómo obtener una lista de puertos de dispositivos (interfaces) en función de un filtro. Tenga en cuenta que también podemos aplicar filtros a los dispositivos de red. Para este ejemplo de código, le pediremos a la API de NSP que nos proporcione una lista de puertos según el nombre del puerto (Puerto 1/1/1, en nuestro caso):

```
#get_ports_filter.py
solicitudes de importación
carga útil={}
encabezados = {
    'Autorización': 'Portador {token}'.format(token)
}
url = "https://{{servidor}}:8544/NetworkSupervision/rest/api/v1/
      puertos?filtro=(nombre='Puerto 1/1/1')"
respuesta = solicitudes.solicitud ("OBTENER", url, encabezados = encabezados, datos
                                    = carga útil)
imprimir (respuesta.texto)
```

El resultado de este programa será una lista de puertos de dispositivos llamada Puerto 1/1/1 de todos los dispositivos de red. Obtener puertos a través de múltiples dispositivos de red con una sola API es el valor real de trabajar con un NMS

A continuación, analizaremos cómo actualizar un recurso de red mediante la API de NMS.

Actualización del puerto del dispositivo de red

La creación de nuevos objetos o la actualización de objetos existentes también es conveniente cuando se utiliza la API de NMS. Implementaremos un caso de actualización de la descripción del puerto, como hicimos en ejemplos de código anteriores, usando Netmiko, NAPALM y ncclient. Para actualizar un puerto o una interfaz, usaremos un extremo de API diferente que está disponible en el módulo **Network Function Manager for Packet (NFMP)**. NFMP es un módulo NMS para dispositivos de red Nokia bajo la plataforma Nokia NSP. Veamos los pasos para actualizar la descripción de un puerto o realizar cualquier cambio en un recurso de red:

1. Para actualizar un objeto o crear un nuevo objeto bajo un objeto existente, necesitaremos el **Nombre completo del objeto (OFN)**, también conocido como **Nombre completamente distingible (FDN)**, de cualquier objeto existente (para actualizar el objeto) o un objeto principal (para crear un nuevo objeto). Este OFN o FDN actúa como clave principal para identificar un objeto de forma única. Para los objetos de red de Nokia disponibles en los módulos NSP, cada objeto tiene un atributo OFN o FDN. Para obtener un OFN para que un puerto se actualice, usaremos el v1/Managedobjects/searchWithFilter API con los siguientes criterios de filtro:

```
#update_port_desc.py (parte 1)
solicitudes de importación
importar json

token = <token obtenido antes>
encabezados = {
    'Tipo de contenido': 'aplicación/json',
    'Autorización': 'Portador {}'.format(token)
}

url1 = "https://NFMP_URL:8443/nfm-p/rest/api/v1/
        objetos gestionados/buscarConFiltro"

payload1 = json.dumps({
    "fullClassName": "equipo.PhysicalPort",
    "filterExpression": "siteId ='<site id>'AND portName='123",
```

```
"filtroresultado": [
    "objetoNombreCompleto",
    "descripción"
]
})
))

respuesta = solicitudes.solicitud ("POST", url1, encabezados
= encabezados, datos = carga útil1, verificar = Falso)
port_ofn = respuesta.json()[0]['objectFullName']
```

En este ejemplo de código, establecemos el nombre del objeto en fullClassNames.

Los nombres de clase completos del objeto están disponibles en la documentación del modelo de objeto NFMP de Nokia. Configuramos filterExpression para buscar un puerto único en función de la identificación del sitio del dispositivo y el nombre del puerto. El atributo resultFilter se utiliza para limitar los atributos que devuelve la API en la respuesta. Nos interesa el atributo objectFullName en la respuesta de esta API.

2. A continuación, usaremos un punto final de API diferente llamado v1/managedobjects/ofn para actualizar un atributo del objeto de red. En nuestro caso, solo estamos actualizando el atributo de descripción. Para la operación de actualización, debemos establecer el fullClassName atributo en la carga útil y un nuevo valor para el atributo de descripción. Para la URL del extremo de la API, concatenaremos la variable port_ofn que calculamos en el paso anterior.
El código de muestra para esta parte del programa es el siguiente:

```
#update_port_desc.py (parte 2)
payload2 = json.dumps({
    "fullClassName": "equipo.PhysicalPort",
    "propiedades": {
        "description": "descripción agregada por un programa de Python"
    }
})

url2 = "https://NFMP_URL:8443/nfm-p/rest/api/v1/
    objetos gestionados/" + port_ofn

respuesta = solicitudes.solicitud ("PUT", url2, encabezados = encabezados,
    datos=carga útil2, verificar=False)

imprimir (respuesta.texto)
```

La automatización de red es el proceso de crear y actualizar muchos objetos de red en un orden específico. Por ejemplo, podemos actualizar un puerto antes de crear un servicio de conectividad IP para conectar dos o más redes de área local. Este tipo de casos de uso requiere que realicemos una serie de tareas para actualizar todos los puertos involucrados, así como muchos otros objetos. Con la API de NMS, podemos orquestar todas estas tareas en un programa para implementar un proceso automatizado.

En la siguiente sección, exploraremos cómo integrarse con Nokia NSP o sistemas similares para la comunicación basada en eventos.

Integración con sistemas controlados por eventos

En las secciones anteriores, discutimos cómo interactuar con los dispositivos de red y los sistemas de administración de red usando el modelo de solicitud-respuesta. En este modelo, un cliente envía una solicitud a un servidor y el servidor envía una respuesta como respuesta a la solicitud. Los protocolos HTTP (REST API) y SSH se basan en un modelo basado en solicitudes y respuestas. Este modelo funciona bien para configurar un sistema u obtener el estado operativo de la red de forma ad hoc o periódica. Pero, ¿qué pasa si sucede algo en la red que requiere la atención del equipo de operaciones? Por ejemplo, supongamos que se ha producido un fallo de hardware en un dispositivo o se ha cortado un cable de línea. Los dispositivos de red suelen generar alarmas en tales situaciones, y estas alarmas deben llegar al operador (a través de un correo electrónico, un SMS o un tablero) de inmediato.

Podemos usar el modelo de solicitud-respuesta para sondear el dispositivo de red cada segundo (o cada pocos segundos) para verificar si ha habido algún cambio en el estado de un dispositivo de red o si hay una nueva alarma. Sin embargo, esto no es un uso eficiente de los recursos del dispositivo de red y contribuirá al tráfico innecesario en la red. ¿Qué pasa si el dispositivo de red o el propio NMS se comunica con los clientes interesados cada vez que hay un cambio en el estado de los recursos críticos o se activa una alarma? Este tipo de modelo se denomina modelo basado en eventos y es un enfoque de comunicación popular para enviar eventos en tiempo real.

Los sistemas basados en eventos se pueden implementar mediante **webhooks/WebSockets** o utilizando el enfoque **de transmisión**. WebSockets ofrece un canal de transporte bidireccional sobre HTTP 1.1 a través de un socket TCP/IP. Dado que esta conexión bidireccional no usa el modelo tradicional de solicitud y respuesta, WebSockets es un enfoque eficiente cuando queremos establecer una conexión uno a uno entre los dos sistemas. Esta es una de las mejores opciones cuando necesitamos comunicación en tiempo real entre los dos programas. WebSockets es compatible con todos los navegadores estándar, incluido el que está disponible con dispositivos iPhone y Android. También es una opción popular para muchas plataformas de redes sociales, aplicaciones de transmisión y juegos en línea.

WebSockets es una solución ligera para obtener eventos en tiempo real. Pero cuando muchos clientes buscan recibir eventos de un sistema, el uso de un enfoque de transmisión es escalable y eficiente. El modelo basado en eventos de transmisión normalmente sigue un patrón de diseño de editor-suscriptor y tiene tres componentes principales, como se describe aquí:

- **Tema:** Todos los mensajes de transmisión o notificaciones de eventos se almacenan bajo un tema.
Podemos pensar en un tema como un directorio. Este tema nos ayuda a suscribirnos a temas de interés para ayudarnos a evitar recibir todos los eventos.
- **Productor:** Este es un programa o pieza de software que empuja los eventos o mensajes a un tema. Esto también se llama el **editor**. En nuestro caso, será una aplicación NSP.
- **Consumidor:** Este es un programa que obtiene los eventos o mensajes de un tema. Esto también se llama un **suscriptor**. En nuestro caso, este será un programa de Python que escribiremos.

Los sistemas controlados por eventos están disponibles para dispositivos de red, así como sistemas de administración de red. Las plataformas NMS usan sistemas de eventos como gRPC o SNMP para recibir eventos en tiempo real desde dispositivos de red y ofrecen interfaces agregadas para la capa de orquestación o las aplicaciones operativas o de monitoreo. Para nuestro ejemplo, interactuaremos con un sistema de eventos de la plataforma Nokia NSP. El sistema Nokia NSP ofrece un sistema de eventos basado en Apache Kafka. Apache Kafka es una pieza de software de código abierto que se desarrolló en Scala y Java, y proporciona la implementación de un bus de mensajería de software que se basa en el patrón de diseño **Publisher-Subscriber**. Antes de interactuar con Apache Kafka, enumeraremos una lista de **categorías** clave (un término utilizado para los temas en Apache Kafka) que se ofrecen a través de Nokia NSP, de la siguiente manera:

- **NSP-FAULT:** Esta categoría cubre eventos relacionados con fallas o alarmas.
- **NSP-PACKET-ALL:** Esta categoría se utiliza para todos los eventos de gestión de red, incluyendo eventos de mantenimiento.
- **NSP-REAL-TIME-KPI:** esta categoría representa eventos para tiempo real transmisión de notificaciones.
- **NSP-PACKET-STATS:** Esta categoría se utiliza para eventos de estadísticas.

Una lista completa de categorías está disponible en la documentación de Nokia NSP. Todas estas categorías ofrecen filtros adicionales para suscribirse a un determinado tipo de evento. En el contexto de Nokia NSP, interactuaremos con Apache Kafka para crear una nueva suscripción y luego procesar los eventos del sistema Apache Kafka. Comenzaremos con la gestión de suscripciones.

Creación de suscripciones para Apache Kafka

Antes de recibir cualquier evento o mensaje de Apache Kafka, debemos suscribirnos a un tema o una categoría. Tenga en cuenta que una suscripción solo es válida para una categoría. Por lo general, una suscripción caduca después de 1 hora, por lo que se recomienda renovar una suscripción 30 minutos antes de la hora de caducidad.

Para crear una nueva suscripción, usaremos v1/notifications/subscriptions API y el siguiente código de muestra para obtener una nueva suscripción:

```
#subscribe.py
solicitudes de importación

token = <token obtenido antes>

url = "https://NSP_URL:8544/nbi-notificación/api/v1/
      notificaciones/suscripciones"

def create_subscription(category): headers =
{'Authorization': 'Bearer {}'.format(token)} payload = { "categories": [ { "name": "}.
      "{}".format(category) } ] } respuesta = solicitudes.solicitud("POST", url, json=carga útil,
      encabezados=encabezados, verificar=False)

imprimir (respuesta.texto)

if __name__ == '__main__':
    create_subscription("NSP-PACKET-ALL")
```

El resultado de este programa incluirá atributos importantes como subscribeId , topicId y expiresAt , entre otros, como se muestra aquí:

```
{ "respuesta":{
  "estado":0,
  "filalnicio":0,
```

```
"FinFila":0,  
"filastotales":1,  
"datos": {  
    "subscriptionId":"440e4924-d236-4fba-b590-  
        a491661aae14",  
    "IdCliente": nulo,  
    "topicId":"ns-eg-440e4924-d236-4fba-b590-  
        a491661aae14",  
    "tiempoDeSuscripción":1627023845731,  
    "expiresAt":1627027445731,  
    "etapa":"ACTIVO",  
    "persistió": verdadero  
},  
"errores": nulo  
}  
}
```

El atributo `subscribeId` se usa para renovar o eliminar una suscripción más tarde .

Apache Kafka creará un tema específicamente para esta suscripción. Se nos proporciona como un atributo `topicId` . Usaremos este atributo `topicId` para conectarnos a Apache Kafka para recibir eventos. Esto explica por qué llamamos categorías de temas generales en Apache Kafka. El atributo `expiresAt` indica la hora en que caducará esta suscripción.

Una vez que una suscripción está lista, podemos conectarnos a Apache Kafka para recibir eventos, como se explica en la siguiente subsección.

Procesando eventos de Apache Kafka

Escribir un consumidor básico de Kafka no requiere más que unas pocas líneas de código Python con la biblioteca `kafka-python` . Para crear un cliente Kafka, usaremos la clase `KafkaConsumer` de la biblioteca `kafka-python` . Podemos usar el siguiente código de muestra para consumir eventos para nuestro tema de suscripción:

```
#basic_consumer.py  
topicid = 'ns-eg-ff15a252-f927-48c7-a98f-2965ab6c187d'  
consumidor = KafkaConsumer(topic_id,  
                           grupo_id='120',  
                           bootstrap_servers=[host_id],  
                           value_deserializer=lambda m: json.loads  
                               (m.decode('ascii')),
```

494 Uso de Python para la automatización de redes

```
versión_api=(0, 10, 1))

tratar:
    para el mensaje en el consumidor:
        si el mensaje es Ninguno:
            Seguir
        demás:
            print(json.dumps(mensaje.valor, sangría=4, sort_
                llaves=True))
    excepto KeyboardInterrupt:
        sys.stderr.write('++++++ Anulado por el usuario ++++++\n')

por fin:
    consumidor.cerrar()
```

Es importante tener en cuenta que debe usar la biblioteca kafka-python si está usando Python 3.7 o posterior. Si usa una versión de Python anterior a la 3.7, puede usar la biblioteca kafka . Hay problemas conocidos con la biblioteca kafka si la usamos con Python 3.7 o posterior. Por ejemplo, existe un problema conocido de que `async` se ha convertido en una palabra clave en Python 3.7 o versiones posteriores, pero se ha utilizado como variable en la biblioteca kafka . También hay problemas con la versión de la API cuando se usa la biblioteca kafka-python con Python 3.7 o posterior. Esto se puede evitar configurando una versión de API correcta como argumento (la versión 0.10.0 , en este caso).

En esta sección, le mostramos un consumidor básico de Kafka, pero puede explorar un ejemplo más sofisticado en el código fuente proporcionado con este libro en https://github.com/nokia/NSP-Integration-Bootstrap/tree/master/kafka/kafka_cmd_consumer.

Renovación y eliminación de una suscripción

Podemos renovar una suscripción con el sistema Nokia NSP Kafka usando el mismo punto final de API que usamos para crear una suscripción. Añadiremos el id de suscripción atributo al final de la URL, junto con el recurso de renovaciones , de la siguiente manera:

```
https://{{servidor}}:8544/nbi-notification/api/v1/notifications/
suscripciones/<subscriptionId>/renovaciones
```

Podemos eliminar una suscripción usando el mismo punto final de la API con el ID de suscripción atributo al final de la URL pero utilizando el método HTTP Delete . Este punto final de la API tendrá el siguiente aspecto para una solicitud de eliminación:

```
https://{{servidor}}:8544/nbi-notification/api/v1/notifications/  
suscripciones/<subscriptionId>
```

En ambos casos, no enviaremos ningún argumento en el cuerpo de la solicitud.

Esto concluye nuestra discusión sobre la integración con NMS y controladores de red utilizando tanto el modelo de solicitud-respuesta como el modelo basado en eventos. Ambos enfoques le brindarán un buen punto de partida cuando se trata de integrarse con otros sistemas de gestión.

Resumen

En este capítulo, presentamos la automatización de redes, junto con sus beneficios y los desafíos que presenta para los proveedores de servicios de telecomunicaciones. También discutimos los casos de uso clave de la automatización de redes. Después de esta introducción, discutimos los protocolos de transporte que están disponibles para la automatización de redes para interactuar con dispositivos de red. La automatización de la red se puede adoptar de muchas maneras. Comenzamos observando cómo interactuar directamente con dispositivos de red utilizando el protocolo SSH en Python. Utilizamos las bibliotecas de Paramiko, Netmiko y NAPALM Python para obtener la configuración de un dispositivo y explicamos cómo enviar esta configuración a un dispositivo de red. A continuación, discutimos cómo usar NETCONF con Python para interactuar con un dispositivo de red. Brindamos ejemplos de código para trabajar con NETCONF y usamos la biblioteca ncclient para obtener una configuración de interfaz IP.

También usamos la misma biblioteca para actualizar una interfaz IP en un dispositivo de red.

En la última parte de este capítulo, exploramos cómo interactuar con los sistemas de gestión de red como Nokia NSP. Interactuamos con el sistema NSP de Nokia utilizando Python como cliente de API REST y como consumidor de Kafka. Brindamos algunos ejemplos de código en términos de cómo obtener un token de autenticación y luego enviamos una API REST a un NMS para recuperar datos de configuración y actualizar la configuración de red en los dispositivos.

Este capítulo incluye varios ejemplos de código para que se familiarice con el uso de Python para interactuar con dispositivos que usan SSH, protocolos NETCONF y el uso de una API REST de nivel NMS. Este conocimiento práctico es fundamental si es un ingeniero de automatización y busca sobresalir en su área mediante el uso de las capacidades de Python.

Este capítulo concluye este libro. No solo cubrimos los conceptos avanzados de Python, sino que también brindamos información sobre el uso de Python en muchas áreas avanzadas, como el procesamiento de datos, la computación sin servidor, el desarrollo web, el aprendizaje automático y la automatización de redes.

Preguntas

1. ¿Cuál es el nombre de la clase de uso común de la biblioteca Paramiko?
para hacer una conexión a un dispositivo?
2. ¿Cuáles son las cuatro capas de NETCONF?
3. ¿Puede enviar la configuración directamente a una base de datos en ejecución en NETCONF?
4. ¿Por qué gNMI es mejor para la recopilación de datos que la configuración de red?
5. ¿RESTCONF proporciona las mismas características que NETCONF pero a través de
¿Interfaces REST?
6. ¿Qué son un editor y un consumidor en Apache Kafka?

Otras lecturas

- Mastering Python Networking, por Eric Chou. •
- Automatización práctica de redes, segunda edición, por Abhishek Ratan. •
- Programabilidad y Automatización de Redes, por Jason Edelman. • La documentación oficial de Paramiko está disponible en <http://docs.paramiko.org/>. • La documentación oficial de Netmiko está disponible en <https://ktbyers.github.io/>. • La documentación oficial de NAPALM está disponible en <https://napalm.readthedocs.io/>.
- La documentación oficial de ncclient está disponible en <https://ncclient.readthedocs.io/>.
- Los modelos NETCONF YANG se pueden encontrar en <https://github.com/YangModelos/yang>.
- La documentación de la API de Nokia NSP está disponible en <https://network.developer.nokia.com/api-documentación/>.

respuestas

1. La clase paramiko.SSHClient .
2. Contenido, Operaciones, Mensajes y Transporte.
3. Si un dispositivo de red no admite una base de datos candidata , normalmente permite realizar actualizaciones directas para la base de datos en ejecución . 4. gNMI se basa en gRPC, que es un protocolo introducido por Google para llamadas RPC entre clientes móviles y aplicaciones en la nube. El protocolo se ha optimizado para la transferencia de datos, lo que lo hace más eficiente en términos de recopilación de datos de dispositivos de red en comparación con su configuración.
5. RESTCONF proporciona la mayor parte de la funcionalidad de NETCONF a través de interfaces REST pero no expone todas las operaciones de NETCONF.
6. El publicador es un programa cliente que envía mensajes a un tema Kafka (categoría) como eventos, mientras que el consumidor es una aplicación cliente que lee y procesa los mensajes de un tema Kafka.



Packt.com

Suscríbase a nuestra biblioteca digital en línea para obtener acceso completo a más de 7000 libros y videos, así como herramientas líderes en la industria para ayudarlo a planificar su desarrollo personal y avanzar en su carrera. Para obtener más información, por favor visite nuestro sitio web.

¿Por qué suscribirse?

- Pase menos tiempo aprendiendo y más tiempo codificando con libros electrónicos y videos prácticos de más de 4000 profesionales de la industria
- Mejore su aprendizaje con planes de habilidades creados especialmente para usted
- Obtenga un libro electrónico o un video gratis todos los meses
- Capacidad de búsqueda completa para acceder fácilmente a información vital
- Copiar y pegar, imprimir y marcar contenido

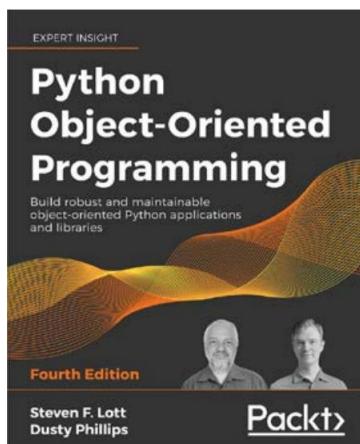
¿Sabía que Packt ofrece versiones de libros electrónicos de cada libro publicado, con archivos PDF y ePub disponibles? Puede actualizar a la versión de libro electrónico en packt.com y, como cliente del libro impreso, tiene derecho a un descuento en la copia del libro electrónico. Póngase en contacto con nosotros en customercare@packtpub.com para obtener más detalles.

En www.packt.com, también puede leer una colección de artículos técnicos gratuitos, suscribirse a una variedad de boletines gratuitos y recibir descuentos y ofertas exclusivas en libros y libros electrónicos de Packt.

500 otros libros que puede disfrutar

Otros libros que usted puede disfrutar

Si disfrutó de este libro, es posible que le interesen estos otros libros de Packt:

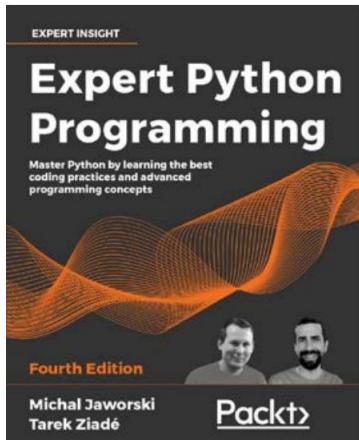


Programación orientada a objetos de Python

Steven F. Lott, Dusty Phillips

ISBN: 978-1-80107-726-2

- Implementar objetos en Python creando clases y definiendo métodos
- Ampliar la funcionalidad de la clase mediante la herencia
- Use excepciones para manejar situaciones inusuales de manera limpia
- Comprender cuándo usar funciones orientadas a objetos y, lo que es más importante, cuándo no para usarlos
- Descubra varios patrones de diseño ampliamente utilizados y cómo se implementan en Pitón
- Descubra la simplicidad de las pruebas unitarias y de integración y comprenda por qué son muy importantes
- Aprende a escribir estáticamente revisa tu código dinámico
- Comprender la concurrencia con asyncio y cómo acelera los programas



Programación Experta en Python – Cuarta Edición

Michaïy Jaworski, Tarek Ziadé

ISBN: 978-1-80107-110-9

- Explorar formas modernas de configurar el desarrollo de Python consistente y repetible ambientes
- Empaque de manera efectiva el código de Python para uso comunitario y de producción. • Aprenda los elementos de sintaxis modernos de la programación de Python, como f-strings, enums, y funciones lambda
- Desmitificar la metaprogramación en Python con metaclasses • Escribir código concurrente en Python • Extender e integrar Python con código escrito en C y C++

Packt está buscando autores como tú

Si está interesado en convertirse en autor de Packt, visite la página de autores. packtpub.com y presente su solicitud hoy. Hemos trabajado con miles de desarrolladores y profesionales de la tecnología, como usted, para ayudarlos a compartir sus conocimientos con la comunidad tecnológica global. Puede hacer una solicitud general, postularse para un tema candente específico para el que estamos reclutando un autor o enviar su propia idea.

Comparte tus pensamientos

Ahora que ha terminado Python for Geeks, ¡nos encantaría escuchar sus opiniones! Si compró el libro en Amazon, haga clic aquí para ir directamente a la página de reseñas de Amazon de este libro y compartir sus comentarios o dejar una reseña en el sitio donde lo compró.

Su revisión es importante para nosotros y la comunidad tecnológica y nos ayudará a asegurarnos de que ofrecemos contenido de excelente calidad.

Índice

simbolos

declaración `__import__`
usando 40

UN

importación absoluta
alrededor de 43
versus importación relativa 42
Clases base abstractas (ABC) 102
clases abstractas en Python 101, 103
abstracción 85
método abstracto 101
pruebas de aceptación 155
modificadores de acceso
captadores 90
setters 90
funciones de
acción recopilar 286
contar 286
reducir 286
información adicional
atributos relacionados 159
información adicional
métodos relacionados 159

pruebas alfa 155
Amazon Alexa 418
Amazon AWS, opciones de tiempo de ejecución en la nube
Corredor de aplicaciones de AWS 317
AWS Lote 317
AWS habichuelas mágicas 317
AWS Kinesis 318
AWS Lambda 317
Amazon S3 332, 419
Enlace de referencia de Amazon SageMaker 455
servicio 452
Amazon SNS 419
Servicios web de Amazon (AWS) 12, 315
Anaconda 288
funciones anónimas 126
Ansible 461
Apache Beam
sobre 332
fundamentos 333, 334
PColección 333
canalización 333
Transformar 333
corredor 333
SDK 333
funciones definidas por el usuario (UDF) 333

Índice 504

- Pipelines de Apache Beam
sobre 334 creación, con
argumentos 338-341 creación, con datos
del archivo de texto 337 creación, con datos
de cadena en memoria 335, 336 creación, con
datos de tuplas en memoria 336, 337
procesamiento, con datos de tuplas en
memoria 336, 337
- Apache Kafka
sobre 460, 491
eventos, procesamiento de 493
suscripciones, creación de 492, 493
- Apache Meso 285
- Proyecto Apache OpenWhisk 419
- Apache Spark
sobre 282
estudio de caso 302-307
administración de clústeres
284 componentes principales
283 idiomas de soporte 283
URL 288
- Configuración de clúster
de Apache Spark 302
IU web, para nodo maestro 304 IU
web, para nodos de trabajo 304
- Apache Spark versión 3.1, administradores de clúster
Apache Mesos 285 Hadoop YARN 285 Kubernetes
285 independiente 284 App Dynamics 396 Interfaz
de programación de aplicaciones (API) 101, 358,
376 servidor de aplicaciones 357 esquema de
nomenclatura de argumentos 22
- Arista EOS 474
- Métodos de afirmación
Aruba AOS-CX 474 158
- matrices asociativas 117
- programación asíncrona, para
sistemas receptivos sobre
266, 267 distribución de tareas del
módulo asyncio 267, usando colas
270, 271
- Puerta de enlace de servicio asíncrono
Interfaz (ASGI) 397
- Asyncio 397
- módulo asyncio
sobre 267 objetos
disponibles, utilizando 269 rutinas
267-269
- tareas 267-269
tareas, ejecutándose simultáneamente 270
- Token de
autenticación Atom 27
obteniendo 485, 486
- Servicio de autorización 393 CI
automatizado 181, 182 Retos de
plataformas de automatización
461
- Corredor de aplicaciones de AWS 317
- AWS habichuelas mágicas 317
- Nube de AWS9 315
- AWS CodeBuild 315
- AWS CodeStar 315
- AWS Lambda 416, 419
- Funciones de Azure 416, 419
- Servicio Azure ML
sobre 452 enlace
de referencia 455
- Tuberías de Azure 315

B

clase base 94
clase TestCase base
 usado, para construir casos de prueba 159, 160
desarrollo por lotes alrededor de 24

CI/CD, empleando 24-26 casos
beneficiosos, subprocessamiento múltiple
 Tareas enlazadas de E/S
 236 aplicaciones multiusuario 237
 aplicación GUI receptiva 236
prueba beta 155
prueba de caja negra 154
bloqueo de bloqueo 243
bootstrap 356
Botella 398
diagrama de caja y bigotes 444
diagrama de caja 444
Aplicaciones Business to Business
 (B2B) 358 bytes conversión
de tipos de datos 115

atributos de clase
 versus atributos de instancia 76-79 clases
sobre 76 esquema de nombres 21

árboles de clasificación y regresión 435 documentación
a nivel de clase sobre 16 detalles algorítmicos 18
métodos de clase

versus métodos de instancia 81-83
script del lado del cliente 357
cierre 199, 200 nube

Entorno de desarrollo Python 314
Construcción en la nube 316
Cloud Code 316
computación en la nube 12
despliegue en la nube
 Servicios web Python, edificio 319, 320
Cloud Function sobre
 416 atributos,
 configuración 421-423 implementación 425,
 426 eventos 420

C

base de datos de candidatos 466
Hojas de estilo en cascada (CSS) 356 Unidades
centrales de procesamiento (CPU) 437
Cáliz 14 niño
clase 94
Ciena SAOS 474
Cisco IOS 474
Cisco IOS-XR 474
Servicios de red de Cisco
 Orquestador (NSO) 461
Cisco NX-OS 474

Código de Python, sumando a 423, 424
Opciones de nube IDE 315
nativo de la nube
 para aplicaciones de Python 314
opciones de tiempo de ejecución en la
 nube de Amazon AWS 317 de GCP
 318 de Microsoft Azure 318, 319 para
 Python 317 proveedores de servicios en
 la nube (CSP) 13

Concha de nube 329
Editor de shell en la nube 330

Índice 506

- aplicación de notificación de eventos de almacenamiento en la nube, compilación para 426-430 computación en clúster 12, 279 administrador de clúster 284 opciones de clúster para procesamiento paralelo 279 implementación de código, problemas 24 implementación, estrategias 24
- Codeanywhere 315 codificación fase 8 colección paralelización 287 interfaz de línea de comandos (CLI) 168 operación de compromiso 151 clases de componentes 103 composición de clase 103 usando, como enfoque de diseño alternativo 103-106 comprensión sobre 211 tipos 211
- Archivo de configuración de ComputingPython 12 utilizado, para configurar el registrador con múltiples controladores 146-148 constante esquema de nomenclatura 20 constructores constructores predeterminados 80 constructor no parametrizado 80 constructor parametrizado 80 usando, con clases 80 consumidor 491
- contenedor 407 contenedor 407 contenerización 407 protocolo de gestión de contexto 130 administrador de contexto usando 130, 131 entrega continua (CD) alrededor de 26, 315, 394 empleando 25
- integración continua (CI) alrededor de 25, 315, 394 empleando 24 capa de controlador 356 multitarea cooperativa 266
- CoreOS rkt 407 rutinas 266 Función de contador sobre 187 usando, para tareas iterativas 187 Crear, leer, actualizar y Eliminar (CRUD) 377 sección crítica 241, 262 Proceso estándar entre industrias para Minería de datos (CRISP-DM) 10 validación cruzada 440 excepciones personalizadas que definen 137, 138
- ## D
- hilos demonio 239 DASK 285 base de datos 357 pruebas basadas en datos (DDT) 174 Marco de datos columna, borrando 221 índice, borrando 221 índices y columnas, renombrando 222 navegando 218, 219 fila, borrando 221 fila o columna, agregando 219, 220

Objeto DataFrame
 trucos avanzados 222

 datos, reemplazando la función
 222-224, aplicando a la columna o fila
 224, 225

 opciones, para agregar columnas 220 filas,
 consultar 226, 227 datos estadísticos, obtener
 227, 228

Operaciones de DataFrame
 sobre 215, 216 índice
 personalizado, configuración de 217

canalizaciones de datos 24 procesamiento
de datos
 Google Cloud Platform, utilizando 332
canalizaciones de procesamiento de datos
 características 11

datos, intercambio entre procesos sobre
 254

 proceso de servidor, usando 257-259
 objetos ctype compartidos (memoria
 compartida), usando 254-257
estructuras de datos conceptos avanzados
 207 función decorada usando, con valor
 de retorno y argumento 203 decoradores

 alrededor de
 200 edificios, con argumentos propios 204, 205
 usados, para modificar el comportamiento de la
 función 200-202 por defecto excepto el bloque
 134 opciones de implementación, para funciones sin
 servidor en nubes públicas

 AWS Lambda 419
 Funciones de Azure 419

 Fase 8 de implementación de Google
Cloud Functions 419

 destructor derivado

 de la fase 7 del diseño
 de la clase 94
 acerca de 163
 usando, con clases 80

operaciones de desarrollo (DevOps) 24 proceso de
desarrollo, elaboración de estrategias acerca de 9
 computación en la nube 12 computación en clúster
 12 para dominios especializados 10

 ML 10, 11

 MVP primero, apuntando a 10
 programación de red 13 fases,
 iterando a través de 9 computación
 sin servidor 13, 14 programación de
 sistemas 12, 13 diccionarios 117, 118
 comprensión de diccionario usando 213

distutils 63

 Django 355, 397
Marco de descanso de Django (DRF) 397, 401
ventana
 acoplable sobre 407
 dirección URL 409

Docker componer 407
Motor acoplable 407
Docker Hub 407
Imagen acoplable 407
Docker Registry 407
docstring sobre 15 directrices
 15 comentarios línea por
 línea 16 estilos 15 tipos
 15 documentación 358

Índice 508

Diseño controlado por dominio (DDD) 395
 escritura de programas de controlador, para
 cálculo de Pi 305-307 digitación de pato en
 Python 106, 107

métodos dunder 19 dunders
 84
 Funciones duraderas 419
 escritura dinámica 106

mi

Elk 396
 encapsulación 85, 198
 extremo a extremo (E2E) 6, 154
 casos de prueba de manejo de
 errores, construcción con 163, 164, 171
 errores
 manejo de 132, 133
 sistemas controlados por
 eventos alrededor de
 490, 491 integración con
 490 bucle de eventos 266
 eventos
 procesamiento, de Apache Kafka 493 manejo
 de excepciones 132, 133 aumento 136, 137
 trabajo con, en Python 133-136

métodos relacionados con la ejecución 158
 Lenguaje de marcado extensible (XML) 13 conjuntos
 de datos externos 287 archivo externo
 RDD, creando a partir de 292
 Extraer, transformar y cargar (ETL) 333 Aumento de
 gradiente extremo
 (XGBoost) 438

F

pruebas de aceptación de fábrica (FAT) 155
 funciones de fábrica 199, 200
 Halcón 397
 FastAPI 377
 Cola FIFO 245 manejo
 de archivos 128 cierre
 de archivos 128, 129
 manejo, en Python
 127 apertura 128, 129
 operaciones 127 lectura 129,
 130 escritura 129, 130

Protocolo de transferencia de archivos
 (FTP) 13 función de filtro sobre 192
 usando, para transformaciones de datos
 194 filtros usando, para transformaciones de
 datos 194

método de finalizador
 agregando, con solicitud fixture 178
 Matraz
 sobre 355, 397
 usando, para REST API 377, 378
 Flask framework sobre
 359 sistemas de
 bases de datos, interactuando con
 368-372 errores, manejo en
 aplicaciones web 373-375 excepciones,
 manejo en aplicaciones web
 373-375 parámetros, extracción de

Solicitud HTTP 366, 367

solicitudes, manejo con métodos HTTP tipos	Consola web de GCP
361, 363, 364 aplicación web, construcción	uso de 329, 331, 332
con enrutamiento 359	disponibilidad general (GA) 155, 323 generadores
Extensión Flask-RESTful 377	de expresiones 126 generadores sobre 124 flujos
FN proyecto 419	infinitos 126 canalización 126, 127 usados, para
Nombre totalmente distingible (FDN) 488	procesamiento de datos 119, 124-127 captadores
Nombre de dominio completo	90
(FQDN) 470	
documentación de nivel funcional	
alrededor de	Bloqueo de intérprete global (GIL) 236
16 detalles algorítmicos 18	Google 316
requisitos funcionales (FR) 7 funciones	Motor de aplicaciones de Google (GAE) 27, 320
incrustadas, dentro de otra función 197 usando	Funciones de la nube de Google 419
trucos avanzados 186	Uso de Google Cloud Platform,
	para el procesamiento de datos 332
	Plataforma en la nube de Google (GCP) 13, 411
GRAMO	SDK de la nube de Google
Nube de GCP	Proyecto de nube de GCP, configuración 321-323
modelo de aprendizaje automático,	requisitos previos 321
implementación 452-455 modelo	Aplicación de Python, creación 324 con aplicación
de aprendizaje automático, predicción	de servicio web 320, implementación en Google
452-455	App Engine 329
Ejecución en la nube de GCP	Aplicación asyncio
microservicio para estudiantes,	de Google Drive, para descargar
despliegue a 411, 412	el archivo 272-274 Aplicación
GCP, opciones de tiempo de ejecución en la nube	multiprocesador, para descargar el archivo
Motor de aplicaciones 318	264-266 Aplicación multiproceso, para
Función de nube 318	descargar el archivo 247-250
CloudRun 318 flujo de	Cubo de almacenamiento de Google 426
datos 318 proceso de	Archivo de eventos de Google
datos 318	Storage 427 eliminar 427
Consola de GCP	finalizar 427 actualizar
utilizado, para la construcción basada en HTTP	metadatos 427
Función de nube 421	
Motores de tiempo de ejecución de GCP 357	

Índice 510

- Microservicio de
 - calificaciones creando 401-406
 - Gradiente 438
 - algoritmos de aumento de gradiente 437
 - Grafana 396
 - interfaz gráfica de usuario (GUI) 319, 483 unidades de procesamiento de gráficos (GPU) 251, 437
 - GráficoX 283
 - Fase verde 179 hilos verdes 266
 - Herramienta GridSearchCV 448, 449
 - Gestión de red gRPC
 - Interfaz (NMI) 466
 - Gunicornio 357
- H**
- Hadoop
 - alrededor de 279
 - componentes básicos 279
 - Sistema de archivos distribuido Hadoop
 - (HDFS) 9, 280
 - Mapa de HadoopReduce
 - componentes básicos 280
 - trabajo 281
 - Hadoop YARN 285
 - controlador 360 tablas hash 117 funciones auxiliares 198 diagrama de histograma 444 Método de retención 445 Función en la nube basada en HTTP
 - edificio, con GCP Console 421
 - El método HTTP tipifica
 - contenidos dinámicos, renderizando 365, 366
 - contenido estático, renderizando 365, 366 usado, para manejar Flask framework 363, 364
 - solicitud HTTP
 - sobre 362
 - cuerpo 362
 - campos de encabezado 362 línea de solicitud 362
 - respuesta HTTP
 - sobre 362 cuerpo 362 campos de encabezado 362 línea de estado 362
 - hiperparámetros sobre 441 optimizar 448
 - Lenguaje de marcado de hipertexto (HTML) 27, 356
- yo
- identificador (ID) 175
 - secuencia inmutable 114
 - convenciones de importación
 - esquema de nomenclatura 22
 - importlib.import_module declaración importación absoluta, versus importación relativa 42
 - usando 41, 42 declaración de importación usando 35 trabajo 36, 38 registro de información 148, 149
 - Tecnología de la información (TI) 13
 - infraestructura como servicio (IaaS) 12 herencia múltiple 96, 97 simple 94, 95 clases internas 86 función interna 197

entrada y salida (E/S) 9 atributos de instancia alrededor de 76 frente a atributos de clase 76-79 métodos de instancia frente a métodos de clase 81-83

Desarrollo y Aprendizaje Integrados

Medio ambiente (IDLE) 26

entorno de desarrollo integrado

(IDE) 26, 161, 314

pruebas de integración 154

inventario de interfaz

obteniendo 487, 488

interfaces obteniendo, vía

NETCONF 479, 480

Tarea de ingeniería de Internet

Fuerza (IETF) 465

Sensores de Internet de las cosas (IoT) 418

artículos 118

iterable 119

Protocolo de iteración 120

iteradores

usado, para procesamiento de datos 119-123

j

Javascript (JS) 27

Notación de objetos de JavaScript

(JSON) 13, 146

Jenkins 316

Jinja2 357, 365

Enebro JunOS 474

k

Kafka 393

Keras 438

componentes clave, programación multiproceso en Python subprocessos daemon 239, 240 cola sincronizada, utilizando 245, 246 módulo de subprocessos 237-239 sincronización de subprocessos 241, 243, 244 claves 117 validación cruzada de k-fold 447

Kubernetes 285

L

funciones lambda sobre

416 edificio 196, 197

Cola LIFO 245

regresión lineal 435

comprensión de listas sobre

212 usando 212 listas

115, 116 IDE local para

desarrollo en la nube 316

repositorio local 23 puntos

finales de servicios de ubicación

usando 484 registrador, con controlador

incorporado y formateador personalizado

usando 143, 144

registrador, con controlador de

archivos usando 144, 145

registrador, con múltiples controladores

configurando, para el archivo de

configuración 146-148

usando 145, 146

Ixc 407

Índice 512

METRO

bibliotecas de aprendizaje automático, en Python
sobre 437
Keras 438
PyTorch 438
Scikit-Learn 437
TensorFlow 438
XGBoost 438

Biblioteca de aprendizaje automático (MLlib) 283

aprendizaje automático (ML)
alrededor de 8, 10, 11, 434, 437
Python, usando 437

aprendizaje automático (ML), algoritmo de componentes
centrales 435 conjunto de datos 435 extracción de
características 435 modelos 436 modelo, prueba
437 modelo, entrenamiento 436 creación de modelos
de aprendizaje automático 439 implementación, en
GCP Cloud 452-455 evaluación 439 evaluación,
con validación cruzada 447 hiperparámetros, ajuste
fino 447 predicción, en GCP Cloud 452-455 guardado,
en el archivo 451, 452

modelo de aprendizaje automático, proceso de construcción
sobre 440, 441
análisis de datos 440
Conjunto de datos de Iris, análisis
442-445 modelado 441 pruebas 441,
445, 446 entrenamiento 445, 446
métodos mágicos 19, 84 archivo main.py

Python 325, 326 Base de información de
gestión (MIB) 465

función de mapeo
sobre 192
usando, para transformaciones de datos 192-194

función de mapeador 280
MapReduce 280
MariaDB 357

pérdida de memoria 130
Mesos 407

sobrecarga de métodos 98, 99
anulación de métodos 99 esquema
de nomenclatura de métodos 19

opciones de desarrollo de
microservicios, Python sobre 397

Botella 398
Django 397
Halcón 397
Frasco 397
Nameko 397
Tornado 398

microservicios
acerca de
392 ventajas 393, 394
mejores prácticas 395, 396
características 392
contenedoreización 407-409
opciones de implementación 398, 399
desventajas 394 estilo de arquitectura
de microservicios 393 creación de aplicaciones
basadas en microservicios 397 ejemplo 393

Microsoft Azure 315
Microsoft Azure, opciones de tiempo de ejecución en la nube
Funciones de la aplicación 318
Servicio de aplicaciones 318
Azure Databricks 319

- Factoría de datos de Azure 319
Lote 318
- producto mínimo viable (MVP) 9 modelo 370
modelo capa 356
- Patrón de diseño del controlador de vista de modelo (MVC) 356
- Patrón de diseño de plantilla de vista de modelo (MVT) 356 módulos sobre 32 código, ejecutando 46 importando 33-35 importando, por otro módulo 45 importando, con declaración de importación 35 cargando 44 esquema de nombres 21 ejecutando, como programa principal 45 variables especiales, configurando 44 variable específica , importando 38-40 módulos estándar 46, 47
- Algoritmo de Montecarlo 305
- multipaso URL 302
- múltiples decoradores usando 205-207
- múltiples archivos operando en 131, 132
- múltiples procesos creando 251 creando, con el objeto Pool 253, 254 creando, con el objeto Process 251, 252 programación multiproceso, componentes clave de Python 237
- mysql 357
- norte
- Bloque de error de nombre 134 Nameko 397
- esquema de nombres en desarrollo 18 convenciones de importación 22 en argumentos 22 en clases 21 en constante 20 en métodos 19 en módulo 21 en paquete 21 en variables 19 resumiendo, en PEP 8 18 herramientas 22
- NAPALM sobre 460, 474 configuración de dispositivo, obteniendo 474, 475 dispositivo de red, configurando 476, 477
- Procesamiento de lenguaje natural (NLP) 309 diccionario anidado sobre 207, 208 agregar a 209, 210 crear 208, 209 definir 208, 209 eliminar de 211 elementos, acceder desde 210 función anidada 197
- NETCONF sobre 465 contenido 465 descripción, actualización de interfaz 481-483 interfaces, obtención 479, 480 mensajes 465

Índice 514

-
- operaciones 465
 - transporte 466
 - usado, para interactuar con dispositivos de red 477, 478
 - Agente 465 de NETCONF
 - Netmiko
 - sobre 460, 470
 - configuración de dispositivo, obteniendo 470, 471
 - dispositivo de red, configurando 472, 473 enlace de referencia 473
 - red como servicio (NAAS) 13 automatización de red
 - sobre 459, 461, 490
 - solicitudes 463
 - impugnaciones 462, 463
 - méritos 462
 - Configuración de red (NETCONF) 13 actualización de puerto de dispositivo de red 488, 489 dispositivos de red
 - interacciones 464
 - obteniendo 487, 488
 - Administrador de funciones de red para Paquete (NFMP) módulo 488
 - Sistema de gestión de red (NMS) 460 sistemas de gestión de red que se integran con 483, 484 programación de red 13
 - Nokia SROS 474 sin bloqueo 243 requisitos no funcionales (NFR) 7
 - North Bound Interfaces (NBI) 483 creación de aplicaciones de notificación, para eventos de almacenamiento en la nube 426-430
 - biblioteca numpy 214
- O**
- Object Full Name (OFN) 488 programación orientada a objetos (OOP) evitando, en Python 107, 108
 - Asignador relacional de objetos (ORM) 368 objetos 76 objetos, intercambio entre procesos alrededor de 259
 - Objeto de tubería, usando 261, 262
 - Objeto de cola, usando 260, 261
 - principios de programación
 - orientada a objetos sobre 85 clases, que se extienden con herencia 93 datos y acciones, que abarcan 85-87 encapsulación de datos 85 protección de datos 90 información, ocultación 87 herencia múltiple 96, 97 decoradores de propiedades, uso 92, 93 herencia simple 94-96 captadores y definidores tradicionales, usando 90-92
 - Enlace de referencia de implementación de OpenConfig 479 Prueba de aceptación operativa (OAT) 155 oráculo 357
- PAG**
- paquete
 - sobre 32
 - acceso, desde cualquier ubicación 58
 - edificio 53-57 edificio, según las pautas de PyPA 64-67 archivo de inicialización 53 archivo de inicialización, usos 53, 54

- instalación, desde código fuente
 - local usando pip 67-69
- instalación, desde PyPI 71
- convenciones de nomenclatura 53
- esquema de nomenclatura 21
- publicación, hasta Test PyPI 70
- compartir 63 sys.path, agregar 58, 60 Pandas DataFrame
- trucos avanzados 214
- estructuras de datos clave
 - de la biblioteca pandas 214, 215
- Plataforma Paragon Automation 461
- procesamiento paralelo de datos
 - PySpark, usando para 288
 - opciones de clúster 279
- enfoque de extracción de parámetros
 - app.get 368 app.post 368 app.route 367 casos de prueba de parametrización, construcción con 174
- prueba parametrizada 174
- Paramiko
 - acerca de la configuración del dispositivo 460, 467, obteniendo 467, 469
- PEP 8
 - acerca de 22 estructura de código, pautas 18
 - resumen del esquema de nombres 18
 - enlace de referencia 18
- Programa de
 - controlador de cálculo Pi, escrito para la construcción de tuberías de 305-307 pip 63, para Cloud Dataflow
- 341-345, 347
- Plataforma como servicio (PAAS) 317
- polimorfismo sobre 98 abstracción 101
 - sobrecarga de métodos 98, 99 anulación de métodos 99-101
- Sistema operativo portátil
 - Interfaz (POSIX) 12
- PostgreSQL 357
- Cola de prioridad 245
- método privado 87
- variable privada 87
- Bloque de control de procesos (PCB) 251
- sincronización
 - de procesos entre 262, 263
- ID de proceso (PID) 251
- productor 491
- Servicio de catálogo de productos 393
- Servicio de inventario de productos 393
- método protegido 88, 89 variable protegida 88, 89 búferes de protocolo (Protobufs) 466 protocolos, para interactuar con
 - dispositivos de red alrededor de 464 gRPC/gNMI 466, 467
 - NETCONF 465, 466
 - RESTCONF 466
 - SNMP465
 - SSH 464
- editor 491
- Patrón de diseño de editor-suscriptor 491
- Py4J 288
- PyCharm 27
- PyCharm Pro 361
- PyDev 27
- Pilinto 22

Índice 516

- Estudio de
 - caso PySpark 307-309
 - exploración, para operaciones RDD 292
 - Operaciones de acción RDD 293, 294
 - Operaciones de transformación RDD 293 usando, para procesamiento de datos en paralelo 288
- PySpark DataFrame sobre
 - 294 creando 295-297
 - trabajando en 297-300
- Documentación de PySpark,
 - clase SparkSession
 - enlace de referencia 291
- cáscara PySpark 289
- PySpark SQL 300 casos
- de prueba de dispositivos
 - pytest, compilación con 176, 177 casos de prueba de pytest framework, compilación con manejo de errores 171 casos de prueba, compilación sin clase base 170 casos de prueba, compilación con parametrización 174, 175 casos de prueba, construcción con accesorios pytest 176, 177 casos de prueba, construcción con marcadores pytest 172, 173 trabajando con 169, 170 marcadores pytest casos de prueba, construcción con 172, 173
- Python
 - sobre 4, 460, 461
 - clases abstractas 101
 - clases de abstracción 103
 - comunidad 4
 - cultura 4
 - capacitación en datos, mejores prácticas 439
- pato escribiendo 106, 107
- excepciones, trabajando con 133-136
- archivos, manejando 127 para automatización de red 464 opciones de desarrollo de microservicios 397 subprocesamiento múltiple 234 OOP, evitando 107, 108 usando, para aprendizaje automático (ML) 437 PythonAnywhere 315 Creación de aplicaciones de Python 324 opciones de nube 314 main.py Archivo de Python 325, 326 archivo de requisitos 326-328 Archivo YAML 325 Punto ciego de Python 236 Adición de código de Python a Cloud Function 423, 424 docString 15 documentación 14 documentación funcional/a nivel de clase 16 Comentarios de Python 14 Colección de Python RDD, creación a partir de 292 Comentarios de Python 14 Contenedores de datos de Python sobre 114 Diccionarios 117 Listas 115, 116
- juegos 118
- cadenas 114, 115
- tuplas 116, 117
- Entorno de desarrollo de Python sobre 26 IDE nativo de la nube 315
- INACTIVO 26
- para la nube 314
- PyCharm 27

- PyDev 27
araña 28
Texto sublime 26
Código de Visual Studio (Código VS) 27
Propuesta de mejora de Python (PEP) 15
Ventajas del módulo de registro
de Python 138, 139
componentes principales 139
registrar predeterminado, usando 141,
142 registrar, usando con el controlador de
archivos 145 registrar con nombre, usando
142, 143 usando 138, 139 trabajando con 141
- Módulo de registro de Python, componentes principales
- aproximadamente
139 niveles 140 registrar 140
formateador de registro 140 controlador de registro 141
- Índice de paquetes de Python (PyPI)
sobre el
paquete 63, instalando desde 71
Autoridad de empaquetado de Python
(PyPA) 53, 159
variable de entorno PYTHONPATH
usando 61
Proyecto Python, fases sobre
6 codificación 8
implementación 8 diseño 7
análisis de requisitos 7
pruebas 8
- Paquete de sitio de Python
Archivo .pth, utilizando 61,
63 marcos de prueba de Python
pytest, trabajando con 169
- unittest, trabajando con 157
trabajando con 155, 156
servicios web Python
para edificio, para despliegue en la nube 319, 320
Consola web de GCP, usando 329, 331, 332
Servicios web de Python, gestión de dependencia
de requisitos clave 320 configuración del entorno
319 interfaz web 319
- PyTorch 438
- q**
- cadena de consulta 362
- R**
- RabbitMQ 393
condición de carrera 241
Herramienta RandomizedSearchCV 450, 451
Operaciones de acción RDD
con PySpark 293, 294
Creación de
objetos RDD 287
creación, desde un archivo externo 292
creación, desde la colección de Python 292
- RDD operaciones
sobre 286 acciones
286
PySpark, exploración de 292
transformaciones 286
- RDD 283, 285, 287
Operaciones de transformación RDD con
PySpark 293
ReactJS 356
Rojo, Verde, Refactor 179
Fase roja 179

Índice 518

reduce la función
 sobre 192 usando,
 para transformaciones de datos 195, 196 función
 reductora 280
 Fase de refactorización
 180 expresiones regulares
 12 pruebas de regresión 155
 aprendizaje por refuerzo 436
 importación relativa sobre 43 versus
 importación absoluta 42

Invocación de método remoto (RMI) 376 Llamada
 de procedimiento remoto (RPC) 283, 376, 397, 465
 Repl.it 315 Transferencia de estado
 representacional (REST) 13, 376, 460 dispositivo de
 solicitud utilizado, para agregar el método de
 finalizador 178 modelo de solicitud-respuesta
 361 requisito análisis 7 Construcción de API REST
 376, 377 desarrollo, para acceso a base de datos
 379-382 Frasco, usando 377, 378 usado, para
 construir aplicación web 382-389 API REST,
 conceptos Punto final de API 378 análisis de
 argumento 379 recurso 378

enrutamiento
 378 retorno de la inversión (ROI) 462
 módulos reutilizables
 escritura 47
 módulos reutilizables, características
 estilo de codificación convencional 50, 51
 funcionalidad de generalización 49, 50

funcionalidad independiente 47-49
 documentación bien definida 52
 registrador raíz 141
 enrutamiento utilizado,
 para crear una aplicación web 359 base de datos en
 ejecución 466 errores de tiempo de ejecución 132

S

desarrollo de aplicaciones basadas en microservicios
 de muestra 400
 escala 283
 Scikit-Aprenda
 sobre 437
 Herramienta GridSearchCV 448, 449
 Herramienta RandomizedSearchCV 450
 enlace de referencia 451
 Protocolo de copia segura (SCP) 13
 Shell seguro (SSH) 13
 Secure Sockets Layer (SSL) 358 seguridad
 358 semáforo 243 computación sin
 servidor 13, 14

Serverless Framework 14, 420 función
 serverless acerca de 416 beneficios 418
 construcción 420 componentes 417
 opciones de implementación 419
 eventos 417

código funcional 417
 resultado 417
 recursos 417
 casos de uso 418
 bibliotecas y marcos sin servidor 14

- comprensión de conjuntos
 - usando 213, 214
- conjuntos 118, 119
- setters 90
- setuptools 63
- herramientas de shell 12
- Gestión de red sencilla
 - Protocolo (SNMP) 13, 465
- Simple Storage Service (S3) 9 kit de desarrollo de software (SDK) 314 opciones de control de fuente, exploración 23 etapas 23
 - repositorio de control de fuente pertenencias no deseadas 23
- SparkContext 284
- Programa SparkContext
 - creando 290, 291
- Núcleo de chispa 283
- SparkSession 284
 - Métodos de la clase
 - SparkSession 291
 - Programa SparkSession
 - creando 290
 - Chispa SQL 283, 300, 302
 - Spark Streaming 283
 - métodos especiales 83, 84
 - 396
 - araña 28
 - SQLite 357
 - SSH 464
 - Bibliotecas de Python basadas en SSH, que interactúan con dispositivos de red alrededor de 467
 - Netmiko 470
 - Paramiko 467
 - SSH versión 2 (SSHv2) 465
 - decoradores estándar 204
- módulos estándar 46, 47
- métodos estáticos 81 enfoque de transmisión 490 método str.encode (codificación, errores) usando 115
 - lenguaje de consulta estructurado (SQL) 9
- estructura de gestión
- Información (SMI) 465
 - Aplicación API para estudiantes que reutiliza 409
 - Microservicio para estudiantes
 - implementación, a GCP Cloud Run 411, 412
 - Actualización de aplicación
 - web de estudiantes 409, 410
 - subclase 94
- Sublime Text 26
- suscriptor 491
 - creación de suscripción, para Apache Kafka 492, 493
 - eliminación 495 renovación 494
- Subversion (SVN) 26
- superclase 94 aprendizaje
- supervisado 436 clasificador de vector de soporte (SVC) 435
- Swagger 358
 - error de sintaxis 132
 - programación de sistemas 12
 - pruebas de sistema 154
- T
 - pruebas basadas en tablas 174
 - nubes de etiquetas 307 tasklets 266
 - Motor de plantilla Tekton 316 357
 - TensorFlow 438

Índice 520

- prueba de automatización 151
- argumentos de casos de prueba 175
- casos de prueba
- aproximadamente 157 edificio, con base TestCase clase 159, 160
- edificio, con manejo de errores 163, 164, 171 edificio, sin base clase 170
- edificio, con parametrización 174 edificio, con accesorios pytest 176, 177 edificio, con marcadores pytest 172, 173 edificio, con dispositivos de prueba 162 implementando, diferencias clave 171
- caso de prueba, etapas actuar 156 organizar 156 afirmar 156
- limpieza 156
- datos de prueba
- 175 desarrollo basado en pruebas (TDD)
- alrededor de 9
 - ejecutando 179
- Fase verde 179
- Fase roja 179
- Fase de refactorización 180
- reglas 179 accesorios de prueba
- alrededor de 156 casos de prueba, construyendo con 162 niveles de prueba alrededor de 152
- pruebas de aceptación 155
- pruebas de integración 154
- pruebas del sistema 154
- pruebas unitarias 153
- fase de prueba 8
- Pruebe el paquete PyPI, publicando en 70 ejecutor de pruebas 157
- Banco de pruebas
- acerca de 157
 - agregando en 163
 - cambiando en 163
 - ejecutando 165-168
 - anterior, puntos clave 161 nubes de texto 307
- El Zen de Python extractos 5, 6
- Bloque de control de subprocessos
- (TCB) alrededor de 234, 235 contador de programa (PC) 234 pila 234
- Registros del sistema (REG) 234 tema 491
- Tornado 398 filtro
- de funciones de transformación 286
- mapa 286
- unión 286
- Protocolo de control de transmisión (TCP) 13
- Seguridad de la capa de transporte (TLS) 358
- Baratija 315
- tuplas 116, 117 hilo 63
- filosofía de dos pizzas 396
- tu**
- enlace de referencia unittest 158
- marco de prueba unitaria
- varias suites de prueba, ejecutando 165-168
- configuración 157

caso de prueba 157
casos de prueba, construcción con base
 TestCase clase 159, 160 casos
de prueba, construcción con manejo de
 errores 163, 164 casos de prueba,
construcción con accesorios de prueba 162 corredor de
prueba 157
conjunto de
pruebas 157 trabajar con
157 pruebas unitarias 153
aprendizaje no supervisado 436
pruebas de aceptación del usuario (UAT) 155
Protocolo de datagramas de usuario (UDP) 13
funciones definidas por el usuario (UDF) 333
interfaz de usuario (UI) 107, 356 interfaz de
usuario (UI), tecnologías script del lado del cliente
 357 motor de plantilla 357

 Marco de interfaz de
 usuario 356 uWSGI 357

V
métodos de validación 158
valores 118 variables, esquema
de nombres alrededor de 19

 Booleano 20
 colecciones 20
 diccionario 20
Vertex AI plataforma 452 ver
capa 356
Caja virtual 302
estudio visual 315
Código de Visual Studio (Código VS) 27

W
creación de
 aplicaciones web, con REST API 382-388
 creación, con enrutamiento 359 aplicación web,
función de controlador de ejemplo de código 360
 initialización 360 ruta 360 cliente web 361
 servidor web 361

requisitos de desarrollo web sobre 355

API 358
servidor de aplicaciones 357
base de datos 357
documentación 358
seguridad 358
interfaz de usuario (UI) 356
marcos web 355, 356 servidor web
 357 marcos web 355, 356

Webhooks 490
servidor web 357
Interfaz de puerta de enlace del servidor web
 (WSGI) 14, 360, 397
WebSockets 491

 Servidor Werkzeug 357 rueda
 63 prueba de caja blanca 153
nube de palabras

 sobre 307 con
 PySpark 307-309 nodos
trabajadores 284

Índice 522

X

XGBoost 438

Y

Archivo YAML 325

Enlace de referencia

de modelos YANG 465, 477

Otro lenguaje de marcado más

(YAML) 146

Otro negociador de recursos más

(YARN) 280

declaración de

rendimiento usando, en lugar de devolver 178

Z

Zappa 14

Bloque ZeroDivisionError 134

Aprovisionamiento de toque cero (ZTP) 463

Funciones de la

tecla de función Zip 189-191

usando, para tareas iterativas 188-191

