Diseñando para un alto rendimiento

24 de julio de 2020

Una vez que ha escrito el código de trabajo, generalmente hay trabajo adicional que hacer. Necesita su código no solo para cumplir su tarea, sino también para hacerlo rápidamente. El rendimiento de su código es qué tan bien utiliza recursos como la memoria y el tiempo. Se dice que el software funciona a un nivel aceptable, lo que significa que utiliza los recursos de manera eficiente y responde a las tareas dentro de un marco de tiempo deseable, es eficiente.

El rendimiento del software afecta a las personas del mundo real todos los días, ya sea que estén tratando de subir sus últimos selfies a Instagram o haciendo análisis de mercado en tiempo real para elegir acciones. La forma en que el software debe funcionar a menudo se reduce a la percepción del usuario. Si algo se siente instantáneo, podría ser lo suficientemente rápido.

El rendimiento del software también puede afectar el resultado final. Si su software requiere que almacene algo en un disco o en una base de datos, minimizar la cantidad de almacenamiento requerida le ahorrará dinero. El software que informa las decisiones de hacer dinero puede generarle más dinero si se ejecuta más rápido. El rendimiento tiene un impacto en el mundo real.

Percepcion humana

Los humanos generalmente perciben los cambios más rápidos que 100 ms como instantáneos. Si hacen clic en un botón y la pantalla responde en 50 ms, están contentos. A medida que la capacidad de respuesta disminuye más de 100 ms, las personas comienzan a notar el retraso.
Para actividades de larga duración como la descarga de archivos grandes, el retraso no siempre se puede evitar. En estos casos, las actualizaciones precisas del progreso son importantes porque cambian la percepción del progreso para que se sienta más rápido.

1. Sufriendo a través del tiempo y el espacio

Si lees sobre software de alto rendimiento, es probable que encuentres las frases complejidad del tiempo y complejidad del espacio. Estos términos parecen sacados directamente de la mecánica cuántica o la astrofísica, pero también tienen un lugar en el software.

La complejidad del tiempo y el espacio son medidas de cuánto más tiempo de ejecución, memoria o almacenamiento en disco necesita su software a medida que aumentan sus entradas. Cuanto más rápido su software consuma tiempo o espacio, mayor será su complejidad.

La complejidad no pretende ser una medida cuantitativa exacta; más bien, le ayuda a comprender cualitativamente qué tan rápido y grande será su software en el peor de los casos. En esta sección, lo ayudaré a construir una intuición para las mediciones de complejidad para que pueda obtener el rendimiento en su trabajo. Sin embargo, hay un proceso formal para determinar la complejidad de su software, y lo abordaré en un momento.

1.1. La complejidad es un poco. . . compleja

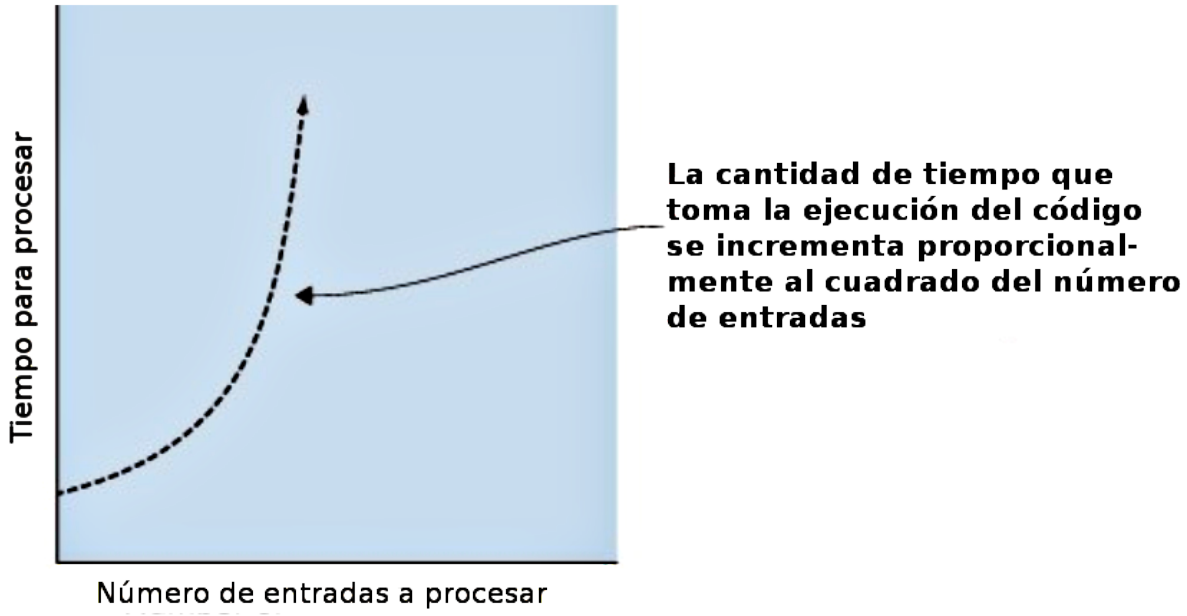
No dudaré en decir nada al respecto: medir la complejidad puede ser difícil y a veces es confuso. Para mí no tenía mucho sentido en la escuela: aprendí lo que sé ahora a través de la aplicación práctica repetida. Prepárate para hacer lo mismo.

Las determinaciones de complejidad se realizan a través de un proceso llamado análisis asintótico, que implica observar el código y determinar los límites de su peor desempeño.

> **Nota**
>
> Tenga en cuenta que las mediciones de complejidad se utilizan para contrastar formas de lograr una tarea en particular; no son tan útiles para comparar tareas no relacionadas. Es útil comparar dos algoritmos para ordenar una lista de números, por ejemplo, pero no puede comparar un algoritmo de clasificación de listas con un árbol de búsqueda. Asegúrate de comparar manzanas con manzanas.

La notación utilizada en el análisis asintótico puede parecer críptica al principio, pero tiene una traducción sencilla en inglés. Comúnmente verá la complejidad escrita en notación O grande, lo que significa el peor rendimiento para el código que se analiza. La notación O grande se parece a $O(n^2)$, a menudo leída como "`orden n-cuadrado`", donde `n` es el número de entradas y n^2 es la complejidad. Esto es la abreviatura de "*la cantidad de tiempo que tarda el código en ejecutarse aumenta proporcionalmente al cuadrado del número de entradas*", como se muestra en la figura 1. $O(n^2)$ es mucho más rápido de escribir. Usaré la notación O grande más en el resto del texto.

Figura 1: $O(n^2)$ es una abreviatura de notación O grande para una relación $y = x^2$.



1.2. Complejidad de tiempo

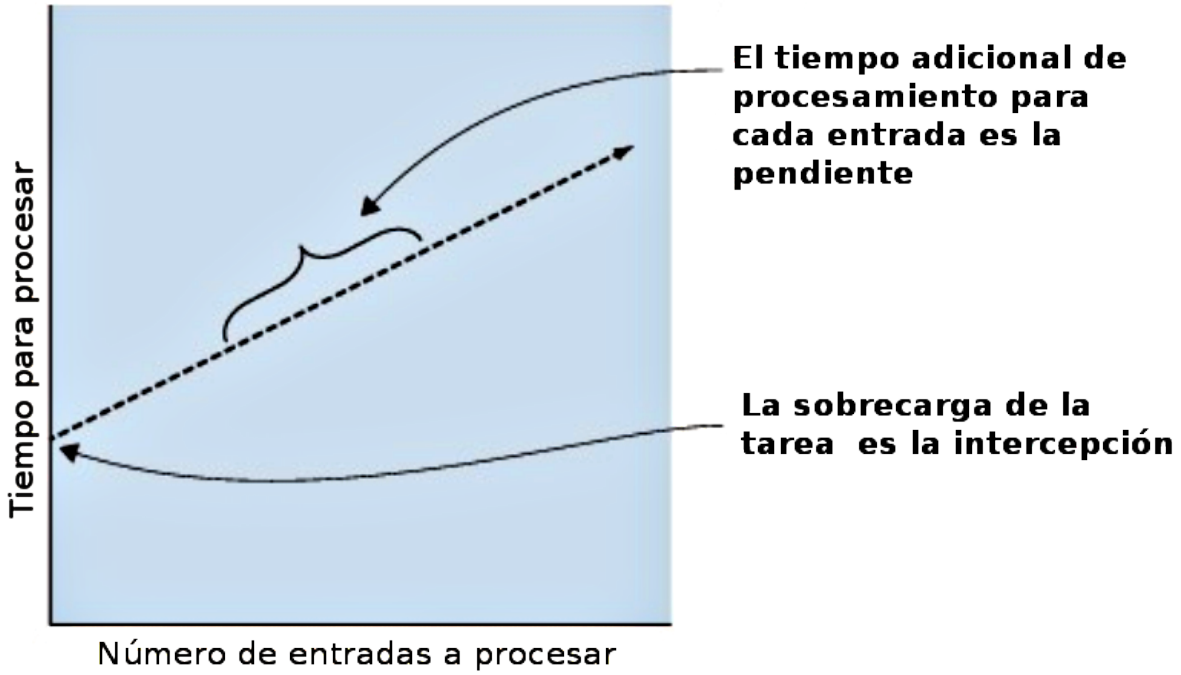
La complejidad del tiempo es una medida de la rapidez con que su código puede realizar una tarea en relación con sus entradas. A medida que aumenta el número de entradas, la complejidad del tiempo le indica a qué velocidad se ralentizará su código. Esto puede ayudarlo a razonar sobre cuánto tiempo debe llevar una tarea a medida que crece la escala de sus entradas.

LINEALIDAD

La complejidad lineal es una de las complejidades más comunes que surgen del código. Esta complejidad se llama así porque graficar el número de entradas frente al tiempo produce una línea recta. Si piensa en la ecuación para una línea en matemáticas, $y = mx + b$, puede pensar en x como el número de entradas e y como el tiempo que le toma ejecutar su programa. Puede haber algo de sobrecarga para su programa independientemente de la entrada (la b, o la intersección, en la ecuación), y cada entrada adicio-

nal agrega una cierta cantidad de tiempo de ejecución (m, o la pendiente). Esto se ilustra en la figura 2.

Figura 2: Visualizando una tarea con complejidad lineal



La complejidad lineal es frecuente en el software porque muchas operaciones necesitan realizar alguna tarea para cada elemento de una lista: imprimir una lista de nombres, sumar una lista de enteros, etc. A medida que la lista crece, la cantidad de tiempo que la computadora tiene que pasar crece proporcionalmente. Sumar 1,000 enteros toma aproximadamente la mitad del tiempo que sumar 2,000 enteros. Para cierto número de elementos, n, este tipo de actividades son lineales con n o, en la notación O grande, $O(n)$.

Puede detectar el código que probablemente sea $O(n)$ en Python buscando bucles. Es probable que un solo ciclo sobre una lista, conjunto u otra secuencia sea lineal:

```
names = ['Aliya', 'Beth', 'David', 'Kareem']
for name in names:
    print(name)
```

Esto sigue siendo cierto incluso si realiza varios pasos dentro del bucle:

```
names = ['Aliya', 'Beth', 'David', 'Kareem']
for name in names:
    greeting = 'Hi, my name is'
    print(f'{greeting} {name}')
```

Incluso sigue siendo cierto si recorre la misma lista varias veces:

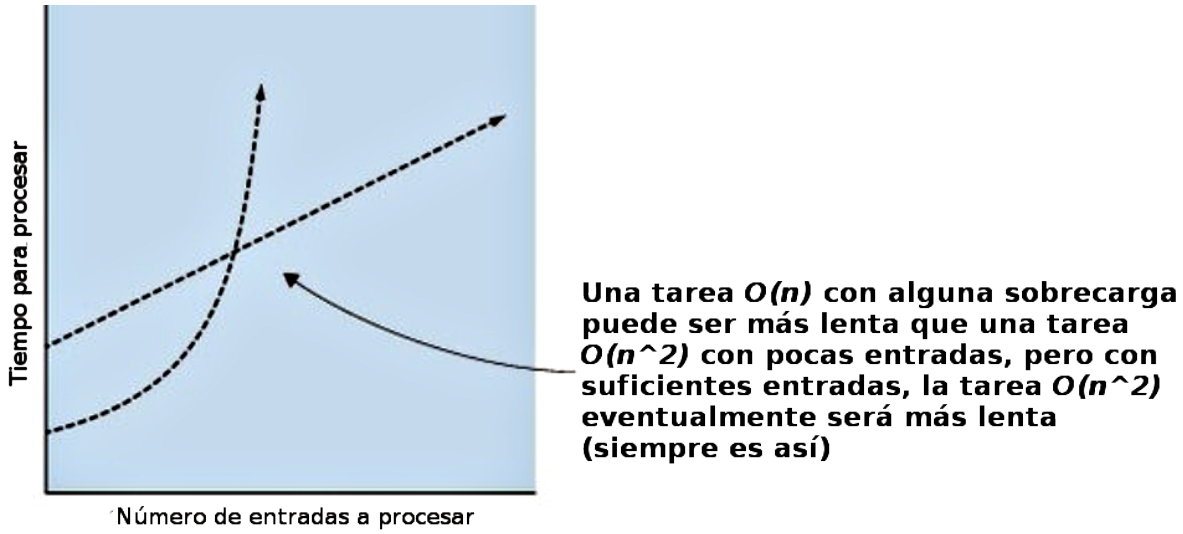
```
names = ['Aliya', 'Beth', 'David', 'Kareem']
for name in names:
    print(f'This is {name}!')

message = 'Let\'s welcome '
for name in names:
    message += f'{name} '
print(message)
```

Aunque está recorriendo la lista de nombres dos veces, piénselo de nuevo en términos de la ecuación para una línea. El primer ciclo lleva algo de tiempo, f, por elemento, y el segundo ciclo lleva algo de tiempo, g, por elemento. La ecuación lineal sería algo así como $y = fx + gx + b$, que es equivalente a $y = (f + g)x + b$. Sigue siendo una línea, incluso si es más pronunciada.

Aquí es donde entra en juego la parte "asintótica" del análisis asintótico. Aunque una actividad particular puede ser abruptamente lineal, otras operaciones más complejas aún pueden superarla si las entradas son lo suficientemente numerosas, como se muestra en la figura 3.

Figura 3: Complejidad de orden superior a grandes escalas



PROPORCIONAL AL CUADRADO

Otro tipo de complejidad temporal es proporcional al cuadrado de las entradas ($\mathrm{O}(\mathrm{n}^2)$). Esto surge en los casos en los que, para cada elemento de una lista, debe mirar todos los demás elementos de la lista. A medida que agrega más entradas, su código debe iterar sobre los elementos adicionales, pero también debe iterar sobre esos elementos adicionales en cada una de esas iteraciones. El aumento en el tiempo de ejecución se agrava.

Puede detectar esto en el código Python por la presencia de bucles anidados. El siguiente código verifica si una lista tiene elementos duplicados:

```
def has_duplicates(sequence):
    # Ref_1
    for index1, item1 in enumerate(sequence):
        # Ref_2
        for index2, item2 in enumerate(sequence):
                # Ref_3
            if item1 == item2 and index1 != index2:
                 return True
    return False
```

Ref_1 El bucle externo itera sobre cada elemento de la secuencia.

Ref_2 El ciclo interno itera sobre cada elemento nuevamente, para cada elemento en el ciclo externo.

Ref_3 Comprueba si dos elementos tienen el mismo valor, pero no el mismo elemento específico de la secuencia

$O(n^2)$ es el peor caso para este código porque incluso si solo los últimos elementos son duplicados, o si no existen duplicados, el código todavía tiene que iterar sobre todas las entradas antes de que finalice. Si los dos primeros elementos son duplicados, el código será mucho más rápido porque puede detenerse de inmediato, pero es útil examinar el peor de los casos para tener una mejor idea de lo que el código es capaz de hacer. La notación Big O siempre mide la complejidad del código en el peor de los casos por este motivo.

Anotaciones adicionales

A veces es útil calcular no solo el peor de los casos, sino también el caso promedio y el mejor. La notación Ω grande (omega grande) se usa para el análisis del mejor caso, y la notación Θ grande (theta grande) se usa para expresar que los límites superior e inferior son de la complejidad especificada. Por lo general, estos pueden ayudarlo a elegir el enfoque que mejor se adapte a lo que está tratando de lograr de varias opciones. La complejidad de muchos algoritmos se puede encontrar buscando en línea, como por ejemplo, "complejidad de clasificación rápida". También puede encontrar la complejidad temporal de algunas operaciones comunes en los documentos de Python (https://wiki.python.org/moin/TimeComplexity).

TIEMPO CONSTANTE

La complejidad ideal es el tiempo constante ($O(1)$), que no depende del tamaño de las entradas. ¡Nada puede ser mejor que el tiempo constante porque eso requeriría que el software se acelere a medida que aumenta su

entrada! El tiempo constante se realiza en algunos de los tipos de datos en Python, de los que hablaré más adelante.

Algunos problemas que normalmente serían lineales (o peores) pueden hacerse constantes después del cálculo inicial. Ese cálculo inicial en sí mismo puede no ser constante, pero si permite que muchos pasos posteriores se vuelvan constantes, puede ser una gran compensación.

1.3. Complejidad espacial

Al igual que la complejidad del tiempo, la complejidad del espacio es una medida de cómo su código usa el espacio en disco o la memoria a medida que crecen sus entradas. Sin embargo, la complejidad del espacio es fácil de pasar por alto, porque no siempre es algo que se observa directamente.

A veces, el uso ineficiente del espacio en disco se vuelve feo solo cuando aparece una ventana emergente que dice que no queda espacio en disco en su computadora. Es bueno pensar en el espacio mientras escribe su código para no consumir sus recursos.

El basurero

Otra cosa que hace que la complejidad del espacio sea más difícil en Python es que a menudo no administra la memoria usted mismo. En algunos idiomas, debe asignar explícitamente y liberar memoria, lo que le obliga a administrar cómo su código usa los recursos. Python utiliza la recolección de basura automática, que libera la memoria que contiene los objetos que ya no están en uso por un programa en ejecución.

MEMORIA

Una forma común en que los programas usan demasiada memoria es leyendo archivos de datos grandes completamente en la memoria cuando no es necesario. Suponga que tiene un archivo de texto que contiene una

fila para cada persona viva hoy y su color favorito. Le gustaría saber la cantidad de personas a las que les gusta más cada color. Puede considerar leer todo el archivo como una lista de filas y operar en la lista:

```
color_counts = {}

with open('all-favorite-colors.txt') as
   favorite_colors_file:
    # Ref_1
    favorite_colors = favorite_colors_file.read().
       splitlines()


for color in favorite_colors:
    if color in color_counts:
        color_counts[color] += 1
    else:
        color_counts[color] = 1
```

Ref_1 Lee todo el archivo en una lista de líneas

Hay mucha gente en el planeta Tierra. Incluso si el archivo solo contuviera una columna de colores favoritos y cada fila usara 1 byte de datos, el archivo tendría un tamaño de poco más de 7 GB. Es posible que tenga tanta memoria en su máquina, pero la tarea no requiere que tenga toda la información de fila disponible a la vez.

En Python, puede leer un archivo línea por línea en un ciclo `for`, y en cada iteración del ciclo, la siguiente línea reemplazará la línea actual en la memoria. Intente actualizar el código para leer una línea del archivo a la vez y vuelva cuando lo tenga.

```
color_counts = {}
with open('all-favorite-colors.txt') as
   favorite_colors_file:
```

```
# Ref_1
for color in favorite_colors_file:
   # Ref_2
   color = color.strip()

   if color in color_counts:
       color_counts[color] += 1
   else:
       color_counts[color] = 1
```

Ref_1 Lee solo una línea a la vez

Ref_2 Elimina el carácter de nueva línea final de cada línea

Al leer una línea a la vez y tirarlas después de obtener lo que necesita, el uso de su memoria será tan alto como la línea más grande del archivo.

¡Mucho mejor! La complejidad del espacio ha pasado de $\mathrm{O}(\mathrm{n})$ a $\mathrm{O}(1)$.

ESPACIO DEL DISCO

Me he encontrado con problemas de espacio en disco en aplicaciones de larga duración en el pasado. A veces son difíciles de ver porque no siempre causan un problema de inmediato. Pueden pasar semanas o meses antes de que se quede sin espacio en disco, ya sea porque su programa escribe pequeñas cantidades de datos a la vez o simplemente porque tiene un gran almacenamiento disponible.

Muchas aplicaciones web grandes emiten registros de su actividad para que puedan depurarse o analizarse. Si introduce una instrucción de registro en su código que se llama 1,000 veces por minuto en producción, esto podría comenzar a consumir espacio en disco rápidamente. Es posible que desee eliminar esa línea, moverla a un lugar que se llame con menos frecuencia o mejorar su estrategia para almacenar registros.

Encontrar oportunidades para cambiar un enfoque de una complejidad de orden superior a una de orden inferior casi siempre producirá mejores ganancias de rendimiento que tratar de obtener el rendimiento de una línea

de código particular. Utilice el análisis de complejidad para comprender dónde se encuentran estas oportunidades en su software. Siga leyendo para ver cómo puede abordar estas oportunidades utilizando algunas de las funciones integradas en Python.

2. Rendimiento y tipos de datos

Aunque su código debe diseñarse teniendo en cuenta la complejidad de tiempo y espacio, en última instancia, se construirá sobre los tipos de datos existentes de Python. Las siguientes secciones cubren varios casos de uso, así como qué tipos de datos son los más adecuados para ellos.

2.1. Tipos de datos para tiempo constante

Recuerde que el rendimiento ideal es uno de tiempo constante, que no aumenta a medida que aumenta el número de entradas. Los tipos `dict` (diccionario) y `set` (conjunto) de Python exhiben este comportamiento al agregar, eliminar y acceder a elementos. Son bastante similares bajo el capó, con la principal diferencia de que los diccionarios asignan claves a valores, mientras que los conjuntos representan un conjunto de elementos únicos e individuales. Iterar a través de los elementos en cualquiera de estos tipos de datos sigue siendo $\mathrm{O}(\mathrm{n})$ en tiempo porque depende del número de elementos contenidos en el objeto. Pero recuperar elementos específicos o verificar si existe un elemento específico es rápido, independientemente de la cantidad total de elementos.

Suponga que, en lugar de contar los colores favoritos del mundo, ahora está interesado en obtener el conjunto único de todos los colores favoritos para poder verificar si alguno de los colores no está representado. Todavía puede leer el archivo línea por línea como antes, pero ¿cómo podría representar los datos y verificar si hay colores específicos?

Intente representar los datos y verifique los colores específicos. Luego regrese aquí y compare su trabajo con la siguiente lista para ver cómo le fue.

Listado de código 1: Usando las características de Python para minimizar

```
all_colors = set()

with open('all-favorite-colors.txt') as
   favorite_colors_file:
    # Ref_1
    for line in favorite_colors_file:
        # Ref_2
        all_colors.add(line.strip())
# Ref_3
print('Amber Waves of Grain' in all_colors)
```

Ref_1 Iterar sobre el archivo sigue siendo $\mathrm{O}(\mathrm{n})$ tiempo.

Ref_2 Agregar a un conjunto es $\mathrm{O}(1)$ tiempo, pero $\mathrm{O}(\mathrm{n})$ espacio.

Ref_3 La membresía del conjunto es una pregunta $\mathrm{O}(1)$.

Al usar un conjunto para mantener una lista de los colores únicos encontrados en el archivo, puede verificar colores específicos en el conjunto en tiempo constante ($\mathrm{O}(1)$) después del ciclo.

2.2. Tipos de datos para tiempo lineal

El tipo de datos de lista en Python exhibe principalmente operaciones $\mathrm{O}(\mathrm{n})$; determinar la pertenencia a una lista o agregar un nuevo elemento a una ubicación arbitraria en una lista es más lento para las listas con más elementos. Agregar o eliminar al final de una lista lleva $\mathrm{O}(1)$ tiempo. Las listas

son útiles cuando los elementos que se almacenan no son identificables de forma exclusiva.

El tipo de tupla es similar a la lista en términos de rendimiento, con la diferencia clave de que las tuplas no se pueden cambiar después de su creación.

2.3. Complejidad espacial de operaciones en tipos de datos

Ahora que está familiarizado con la complejidad temporal de algunas de las estructuras de datos integradas de Python, le enseñaré algunos trucos para usarlas. Los tipos de datos que hemos visto hasta ahora son todos iterables, objetos que admiten la iteración sobre sus contenidos (en un bucle `for`, por ejemplo). La iteración sobre un conjunto de elementos casi siempre será $\mathrm{O}(\mathrm{n})$ en complejidad de tiempo; revisar cada elemento lleva más tiempo si hay más elementos. Pero, ¿qué pasa con la complejidad del espacio?

Para los tipos de datos que hemos visto hasta ahora, todos sus contenidos se almacenan en la memoria juntos. Si una lista tiene 10 elementos, ocupa aproximadamente 10 veces más espacio en la memoria que una lista con un solo elemento, como se muestra en la figura 4. Esto significa que su complejidad espacial también es $\mathrm{O}(\mathrm{n})$. Esto puede ser problemático, de la misma manera que leer 7,6 mil millones de registros en la memoria es problemático. Si no necesitamos todos esos datos a la vez, podríamos encontrar un enfoque más eficiente.

Use *generadores*. Los generadores son construcciones en Python que producen un solo valor a la vez, haciendo una pausa hasta que se solicita el siguiente valor (figura 5). Esto se parece mucho al enfoque que utilizó anteriormente para leer un archivo línea por línea. Al producir un valor a la vez, un generador evita almacenar todos los valores que produce en la memoria a la vez.

Figura 4: La huella de memoria de las listas

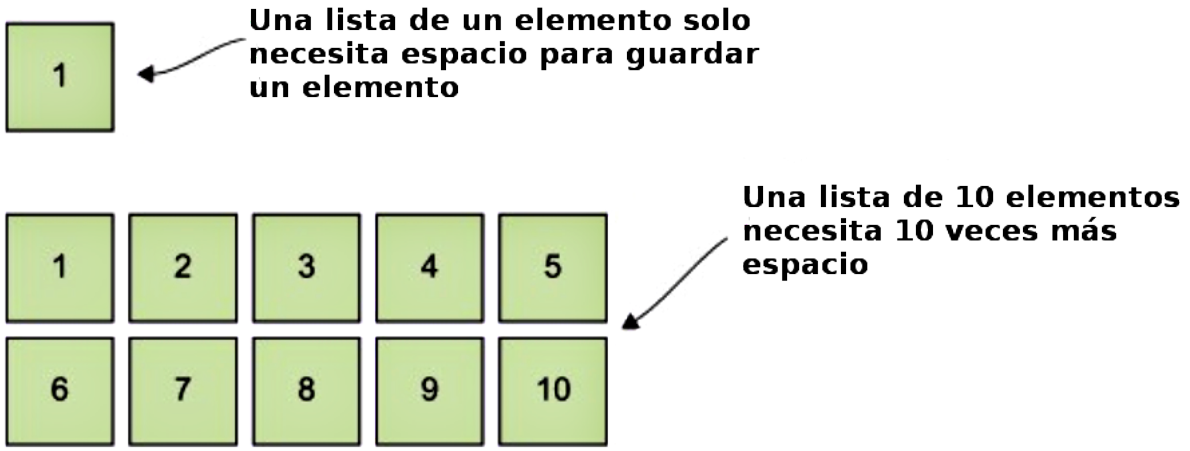
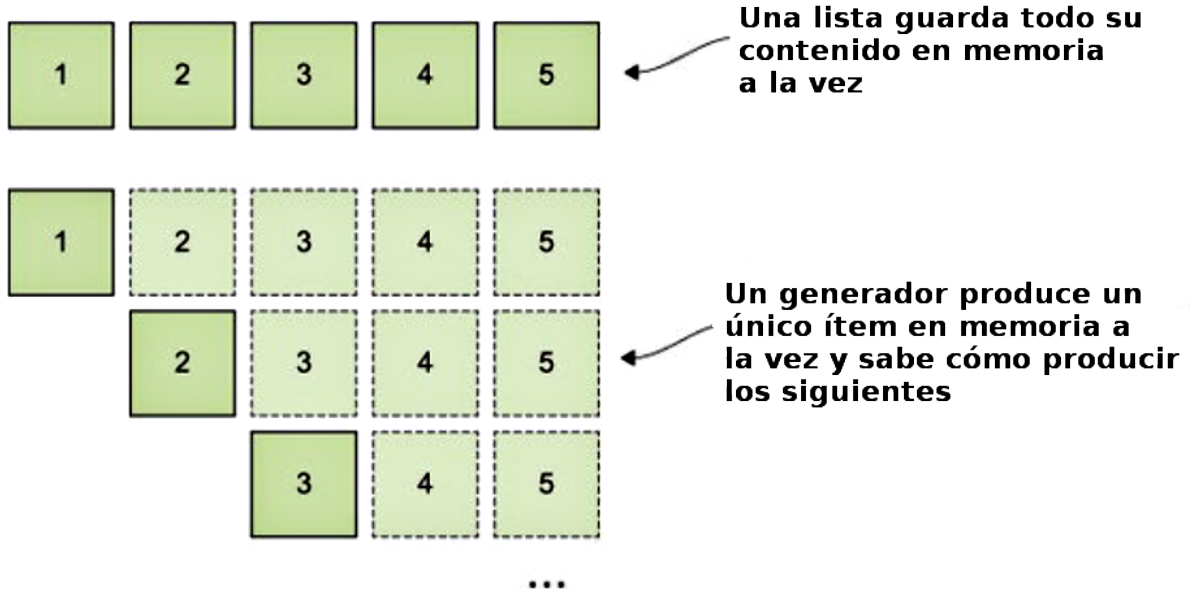


Figura 5: Ahorrando espacio con generadores



Si ha usado la función de rango en Python anteriormente, ya ha usado un generador. `range` acepta argumentos que especifican los límites del rango que desea. Si el rango almacenara todos los números del rango en la memoria, un código como `range(100_000_000)` consumiría su memoria disponible en poco tiempo. En cambio, el rango almacena solo los límites del rango y produce valores de uno en uno. ¿Pero cómo?

Para utilizar el espacio de manera eficiente, los generadores utilizan la pa-

labra clave Python `yield`. Después de producir un valor, devuelven la ejecución al código de llamada. Por lo tanto, el rendimiento produce un valor y luego la ejecución.

`yield` funciona de manera muy similar a la declaración de retorno de Python, excepto que puede realizar operaciones después de obtener un valor. Esto se puede usar para configurar el siguiente valor que desea producir. La siguiente lista muestra aproximadamente cómo se comporta el rango debajo del capó. Tenga en cuenta el uso de la palabra clave `yield` y el incremento del valor actual después de utilizarla.

Listado de código 2: Usando `yield` para pausar y preparar

```
def range(*args):
    # Ref_1
    if len(args) == 1:
        start = 0
        stop = args[0]
    else:
        start = args[0]
        stop = args[1]

    current = start

    while current < stop:
          # Ref_2
        yield current
                # Ref_3
        current += 1
```

Ref_1 Analiza argumentos para determinar los límites del rango

Ref_2 produce cada valor (uno a la vez)

Ref_3 Realiza la configuración necesaria para el siguiente valor

Hay un patrón en esta implementación que verá repetido a menudo en los generadores:

1. Realice la configuración principal requerida para producir todos los valores.

2. Crea un bucle.

3. Produzca un valor en cada iteración del bucle.

4. Actualice el estado para la próxima iteración del bucle.

Intente inspeccionar los valores de su generador de rango ahora. Puede convertirlo en una lista usando `list(range(5, 10))`, por ejemplo. También puede avanzar un valor a la vez guardando `range(5, 10)` en una variable y haciendo llamadas sucesivas al siguiente (`my_range`).

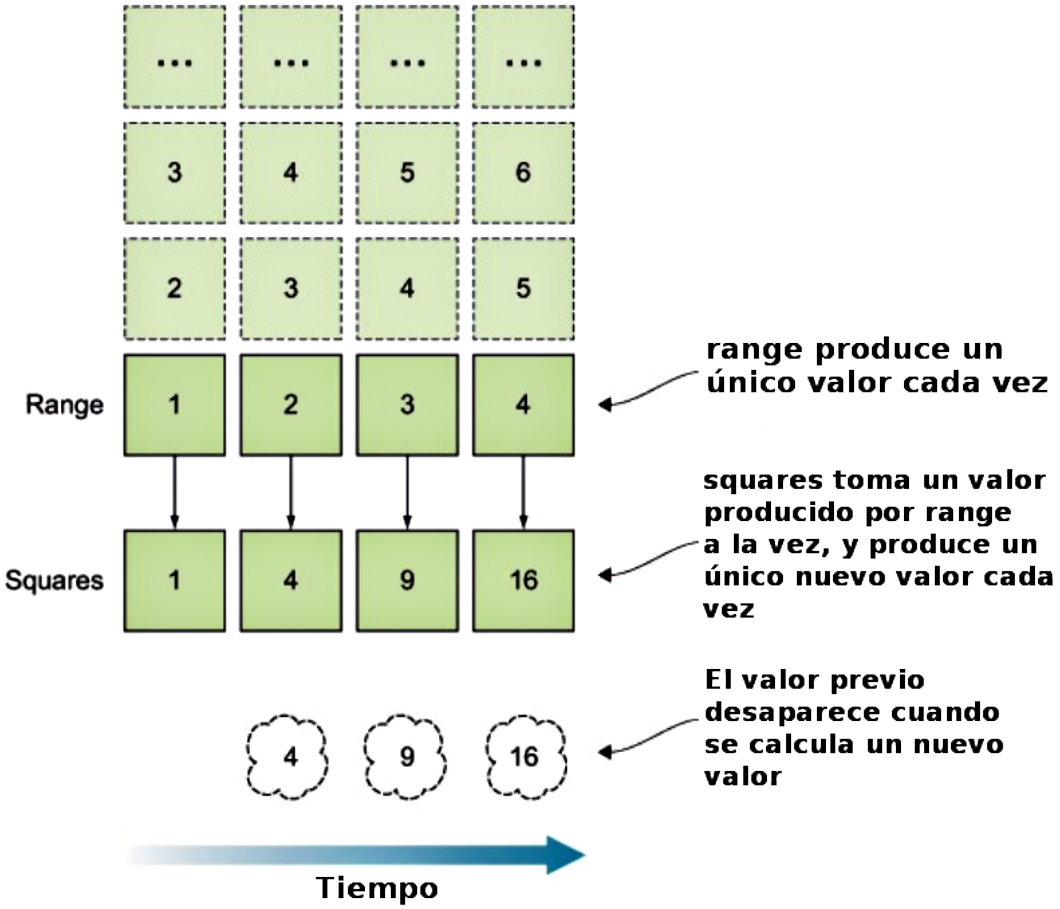
Ahora que tiene este patrón a mano, me gustaría que escriba su propio generador. Su función de generador, cuadrados, tomará una lista de enteros y producirá el cuadrado de cada uno de ellos. Pruébalo y vuelve a la siguiente lista para ver cómo te fue.

Listado de código 3: Un generador corto que produce números cuadrados

```
def squares(items):
    for item in items:
        yield item ** 2
```

La función de cuadrados termina siendo bastante compacta porque no hay configuración o administración de estado que hacer. También dije que esta función acepta una lista, pero lo bueno de esto es que puedes pasar a otro generador. `squares(range(100_000_000))` funciona igual de bien. Solo almacenará un elemento del rango y un resultado al cuadrado a la vez, ahorrando aún más espacio (como se muestra en la figura 6).

Figura 6: Uso de memoria de generadores encadenados



Recomiendo usar generadores sobre listas siempre que pueda, porque siempre puede construir una lista completa en la memoria desde un generador si es necesario escribiendo `list(range (10000))` o `list(squares ([1, 2, 3, 4] ))`. El uso de generadores ahorrará memoria, pero también puede ahorrar tiempo porque el código que consumen los valores del generador puede no necesitarlos de todos modos.

Evaluación perezosa

La idea de producir un valor a la vez, y que consumir código puede no necesitar todos los valores que puede producir, a menudo se conoce como evaluación perezosa. Es perezosa porque le gustaría hacer el menor trabajo posible, y solo una vez que se le haya pedido explícitamente que lo haga. Imagine a sus generadores dejando escapar un suspiro exagerado cada vez que se les pide que den un valor.

3. Haz que funcione, hazlo bien, hazlo rápido

El adagio "*hazlo funcionar, hazlo bien, hazlo rápido*" proviene de Kent Beck, el creador de la programación extrema. A primera vista, esto podría significar que primero debe escribir código de trabajo, luego volver a trabajarlo para que sea claro y conciso, y solo entonces hacerlo con rendimiento. Pero me gusta pensar que estas tres reglas son los pasos que das en cada pequeña iteración a medida que escribes el código. Recuerde que el diseño, la implementación y la refactorización ocurren en ciclos ajustados a medida que codifica.

3.1. Haciéndolo funcionar

Francamente, esto es en lo que los desarrolladores dedican gran parte de su tiempo. Intentar convertir una declaración o idea del problema en un código que logre el objetivo.

Los desarrolladores a menudo trabajan todo el tiempo a través de un problema antes de pasar a la refactorización o el rendimiento. Puede parecer un problema de huevo y gallina: ¿cómo puedo hacer "eso" rápido si "eso" aún no se ha hecho?

Así como la descomposición es útil para el software en sí, también es útil como herramienta para dividir los objetivos en fragmentos manejables. Ca-

da uno de esos objetivos más pequeños se puede implementar y examinar gradualmente en el camino para lograr el objetivo más amplio. También es mucho más fácil en este enfoque "hacer que funcione", porque "es" un objetivo más granular. Puede esbozar algunas ideas para "calcular la velocidad de un objeto que cae" más fácilmente que "hacer un motor físico".

3.2. Haciendo lo correcto

Hacer que funcione se trata de tratar de llegar del punto A al punto B. Si tiene claro el objetivo de la tarea, "¿Funciona?" Es una respuesta binaria.

Hacer las cosas bien se trata de refactorizar. La refactorización busca volver a implementar el código existente de una manera más clara o mejor adaptada, mientras proporciona el mismo resultado consistente[1].

La refactorización no tiene un momento claro de "hecho". Te encontrarás iterando mientras implementas, así como cuando vuelvas a visitar el código para agregar una nueva funcionalidad. Para una métrica sobre cuándo definitivamente debería refactorizar, la regla de tres de Martin Fowler dice que alrededor del tiempo en que haya implementado algo similar tres veces, debe refactorizar su código para proporcionar una abstracción de ese comportamiento. Me gusta esta premisa porque sugiere un equilibrio en torno a la refactorización: no abstraiga algo inmediatamente, o incluso después de haberlo duplicado dos veces, porque podría ser prematuro. Espere a ver qué casos de uso surgen. Le permitirán generalizar la solución de manera más efectiva y asegurarse de que sea necesaria.

Otro aspecto para hacer algo bien es usar las fortalezas del lenguaje para su ventaja. Eche un vistazo al siguiente código, que determina el número entero más frecuente en una lista:

[1] Hay una escuela de pensamiento que dice que puedes crear pruebas para el código que escribes y, si las pruebas son suficientes y aprobadas, puedes apoyarte en ellas mientras haces cambios para asegurarte de que no hayas roto nada. Hay varios textos fantásticos sobre este tema. Ver Harry Percival, Desarrollo impulsado por pruebas con Python, segunda edición (O'Reilly, 2017).

```
    # Ref_1
    def get_number_with_highest_count(counts):
        max_count = 0
        for number, count in counts.items():
            if count > max_count:
                max_count = count
                number_with_highest_count = number
        return number_with_highest_count


    def most_frequent(numbers):
        counts = {}
        # Ref_2
        for number in numbers:
            if number in counts:
                counts[number] += 1
            else:
                counts[number] = 1

        return get_number_with_highest_count(counts)
```

Ref_1 Determina el número con el conteo más alto en un `dict` que asigna números a conteos

Ref_2 Cuenta la aparición de números para ver cuál tiene el conteo más alto

He hecho que esto funcione, pero Python tiene algunas herramientas para hacerlo más fácil. La primera herramienta ayuda con el código que incrementa el conteo. Para cada número en la lista, debe verificar si ya lo hemos visto para saber si puede incrementar el conteo o si debe inicializar el conteo. Python tiene un tipo de datos incorporado para evitar este trabajo extra: el `defaultdict`. Puede decirle a un `defaultdict` el tipo de los valores que almacena, y el valor predeterminado será un valor sensible de ese tipo si se accede a una nueva clave:

```
# Ref_1
from collections import defaultdict

def get_number_with_highest_count(counts):
     max_count = 0
     for number, count in counts.items():
         if count > max_count:
             max_count = count
             number_with_highest_count = number
     return number_with_highest_count


def most_frequent(numbers):
     # Ref_2
     counts = defaultdict(int)
     for number in numbers:
         # Ref_3
         counts[number] += 1

     return get_number_with_highest_count(counts)
```

Ref_1 Importa `defaultdict` desde el módulo de colecciones

Ref_2 Los recuentos son enteros, por lo que el tipo predeterminado de cada valor en el `defaultdict` debe ser `int`.

Ref_3 El valor predeterminado para `int` es 0, por lo que la primera vez que veamos un número, su recuento será `0 + 1 = 1`.

No está mal: se guardó una línea de código y el espíritu de la función es un poco más claro. Pero puedes hacerlo aún mejor. Python también tiene un ayudante para contar cosas en una secuencia:

```
# Ref_1
from collections import Counter
```

```
def get_number_with_highest_count(counts):
    max_count = 0
    for number, count in counts.items():
        if count > max_count:
            max_count = count
            number_with_highest_count = number
    return number_with_highest_count


def most_frequent(numbers):
    # Ref_2
    counts = Counter(numbers)
    return get_number_with_highest_count(counts)
```

Ref_1 Contador también está en el módulo de colecciones.

Ref_2 Actúa casi idénticamente al `dict` de recuentos que realizó manualmente

Has guardado algunas líneas más, y ahora el espíritu de `most_frequent` es bastante claro: cuenta los números únicos y devuelve el que tenga la cuenta más alta. Pero, ¿qué pasa con `get_number_with_highest_count`? Se trata de encontrar el valor máximo en un diccionario que asigna números a sus recuentos.

Python proporciona dos herramientas que también pueden simplificar esta función.

El primero es `max`. `max` acepta un iterable (listas, conjuntos, diccionarios, etc.) y devuelve el valor máximo de ese iterable. En el caso de un diccionario, `max` devuelve el valor máximo de las claves por defecto. Las claves del diccionario de recuentos son los números mismos, no los recuentos. `max` acepta un segundo argumento, key, que es una función que le dice a `max` qué parte del iterable usar.

Python solo pasará un argumento a la clave: el valor del iterable. En el caso de los diccionarios, Python itera sobre sus claves, por lo que la función pasada al argumento clave para `max` solo obtendrá los números pero no sus recuentos. Debe indicarle que, cuando se le da un número, debe indexar el diccionario de conteos en ese número para obtener el valor de conteo. Escribir una función separada en el módulo no funcionará porque los recuentos no estarán disponibles en absoluto en su espacio de nombres. ¿Cómo puedes evitar esto?

En la programación funcional, es común pasar funciones como argumentos a otras funciones, y a veces esas funciones pasadas son cortas y lo suficientemente claras como para que no necesiten nombres. A diferencia de la mayoría de las funciones que probablemente haya escrito en Python, son funciones anónimas y se llaman **`lambdas`**. Las lambdas son de hecho funciones; aceptan argumentos y devuelven valores. No tienen nombres, y no puede llamarlos directamente, pero puede usarlos como argumentos en línea para otras funciones para hacer las cosas.

En el caso de la función `get_number_with_highest_count`, puede pasar una lambda a `max` que acepte un número y devuelva `counts[número]`. Esto resuelve el problema del espacio de nombres y proporciona el comportamiento que desea dar a `max`. Veamos cuán sucinto hará que quede el código:

```
from collections import Counter


def get_number_with_highest_count(counts):
    # Ref_1
    return max(
        counts,
        key=lambda number: counts[number]
    )


def most_frequent(numbers):
```

```
    counts = Counter(numbers)
    return get_number_with_highest_count(counts)
```

Ref_1 Al iterar sobre los números en recuentos, utiliza recuentos[número] (recuento de ese número) como valor de comparación[2]

Eso es conciso y aún claro. Comprender qué herramientas tiene un idioma para qué actividades a menudo lo ayudará a producir código más corto (y en eso Python tiene de sobra, tanto módulos estándar como otros que se pueden descargar en PyPI).

Más corto no siempre es mejor, por supuesto. Podrías ir más allá y mover el `max` directamente a la función `most_frequent`, pero cuando uso funciones como `max` que no siempre son perfectamente claras sobre su comportamiento, me gusta tener la función separada con un nombre más claro.

Una vez que llega a un punto en el que el código que ha escrito funciona, y está lo suficientemente claro sobre cómo funciona que alguien más podría recogerlo y usarlo, lo ha hecho bien.

3.3. Haciéndolo rápido

Ajustar el rendimiento de su código a menudo puede llevar tanto tiempo como escribir el código en primer lugar. El análisis de complejidad y las mejoras posteriores requieren cuidado y una buena mirada a los tipos de datos y operaciones en su código. Deberá sopesar el tiempo perdido en el ajuste del rendimiento frente a la necesidad de llevar su trabajo al mercado. Como se mencionó al comienzo de este capítulo, también deberá decidir cuándo el código es lo suficientemente eficiente[3].

La perfección es el enemigo de lo suficientemente bueno, como dicen, por lo que a menudo es mejor enviar algo valioso pero lento que no enviar nada en absoluto.

[2] Este truco es muy útil para ordenar listas de tuplas, o listas de listas, especificando la posición con respecto a la cual se hará la ordenación

[3] Si programas en Python, intenta siempre hacer algo bello y no simples chapuzas rápidas. Aprovecha la belleza intrínseca del lenguaje

Si su prioridad es llevar algo al mercado, considere establecer algunos hitos de rendimiento para usted que pueda alcanzar de forma iterativa después de su lanzamiento inicial. De esta manera, puede concentrarse en hacer que algo funcione, hacerlo funcionar correctamente para facilitar la mejora futura y enviar su producto. Es probable que encuentre cuellos de botella nuevos e inesperados al ejecutar su código en producción.

Su nivel aceptable de rendimiento también variará en función de sus objetivos. Si está mostrando un modal para iniciar sesión en un sitio después de hacer clic en Iniciar sesión, debe suceder instantáneamente o sus usuarios se irán. Si está tratando de crear un sistema de informes anuales para que los clientes puedan ver sus ventas, es probable que esperen un poco.

La arquitectura del sistema (todos los diferentes servicios, páginas, interacciones, etc.) también informará y afectará el rendimiento. Los sistemas más grandes requieren más comunicación de red entre API, bases de datos y cachés. También pueden tener algunos procesos que ocurren fuera del flujo de trabajo del usuario, como la acumulación nocturna de métricas para el análisis. Puede examinar otros servicios dentro de esta arquitectura que realizan tareas similares a las suyas para tener una idea de la línea base. A partir de ahí, puede crear una expectativa informada del rendimiento de su software y esforzarse por lograrlo. El rendimiento de los sistemas grandes trasciende el código.

A medida que escriba más código, traiga lo que ha aprendido sobre el rendimiento de los tipos de datos y las técnicas para aplicar en su software. Puede comenzar a desarrollar un sentido para las líneas de código que pueden causar problemas de rendimiento. Los bucles anidados y las enormes listas en memoria comenzarán a llamarte la atención.

4. Herramientas

Las pruebas de rendimiento en el mundo real deben seguir un enfoque basado en la evidencia. Este es un resultado directo del hecho de que los sistemas con usuarios reales inevitablemente experimentarán un comporta-

miento diferente; la combinación de entradas inesperadas, temporización, hardware, latencia de red y más, contribuyen al rendimiento del sistema. Como tal, hurgar en su código con la esperanza de tropezar con grandes ganancias de rendimiento puede no ser el mejor uso de su tiempo.

4.1. Cronométralo

El módulo `timeit` en Python es una herramienta para probar el tiempo de ejecución de fragmentos de código. Se puede usar desde la línea de comando o directamente en su código para tener más control. El módulo `timeit` es útil para verificar la cordura de los cambios de rendimiento que tiene la intención de hacer.

Imagina que te gustaría ver cuánto tiempo lleva sumar los enteros desde 0 a 999. Para cronometrar esta actividad desde la línea de comandos, puede invocar el módulo `timeit` con Python:

```
python -m timeit "total␣=␣sum(range(1000))"
```

Esto hará que `timeit` ejecute el código de suma muchas veces, y finalmente imprima algunas estadísticas sobre el tiempo de ejecución:

```
 20000 loops, best of 5: 18.9 usec per loop
```

De esta salida puede concluir que la suma de 0–999 generalmente toma menos de 20 microsegundos.

Para ver cómo la suma de 0–4999 afecta el resultado, puede cambiar su comando y volver a ejecutarlo:

```
python -m timeit "total␣=␣sum(range(5000))"
2000 loops, best of 5: 105 usec per loop
```

A partir de esto, puede concluir que sumar los enteros `0-4999` lleva un poco más de cinco veces más que `0-999`.

Tenga en cuenta que `timeit` realmente está ejecutando su código, y la ejecución en el mundo real tiene pequeñas variaciones debido a muchas variables. Además del código, cosas como el nivel de la batería y la velocidad del reloj de la CPU pueden afectar el tiempo. Como tal, es bueno ejecutar sus comandos de sincronización varias veces para ver qué tan estable es la medición y buscar mejoras significativas desde esa línea de base al realizar cambios. Entonces, aunque `timeit` le brinda mediciones cuantitativas, es mejor usarlo para comparar dos implementaciones diferentes cualitativamente, centrándose en la tendencia. Aquí es donde notará esas mejoras de orden de magnitud que aceleran notablemente su código.

La interfaz de línea de comandos para `timeit` es excelente, pero puede ser engorrosa cuando desea probar piezas de código más grandes o más complejas. Si necesita más control sobre lo que se está cronometrando, puede usar `timeit` desde su código. Si desea cronometrar una parte específica del código sin cronometrar todo el código de configuración que requiere, puede separar el paso de configuración para que no se incluya su tiempo de ejecución:

```
from timeit import timeit
# Ref_1
setup = 'from datetime import datetime'
# Ref_2
statement = 'datetime.now()'
# Ref_3
result = timeit(setup=setup, stmt=statement)
print(f'Took an average of {result}ms')
```

Ref_1 Este código prepara el escenario para la prueba de tiempo.

Ref_2 Este código se ejecuta dentro del temporizador.

Ref_3 `timeit` produce un resultado de tiempo, en milisegundos.

Esto terminará cronometrando solo la llamada `datetime.now()` sin cronometrar la importación necesaria para realizar la llamada.

Suponga que le gustaría probar que verificar si un elemento está en un conjunto es más rápido que verificar si está en una lista. ¿Cómo harías eso usando el módulo `timeit`?

Construya sus entradas usando `set(range(10000))` y `list(range(10000))`, y programe la tarea de averiguar si hay 300 de diferencia entre en ellas.

¿Cuánto más rápido es el set?

El módulo `timeit` me ha salvado de pasar por una madriguera de conejo varias veces al decirme que mi hipótesis sobre acelerar un código era incorrecta. Es un verdadero ahorro de tiempo.

4.2. Perfiles de CPU

Cuando usabas `timeit`, el módulo estaba perfilando tu código. La creación de perfiles significa analizar su código mientras se ejecuta para recopilar algunas métricas sobre su comportamiento. El módulo `timeit` midió cuánto tiempo tardó en ejecutarse su código en total, pero otra forma perspicaz de medir el rendimiento de su código es a través de perfiles de CPU. La creación de perfiles de CPU le permite ver qué partes de su código realizan cálculos costosos, así como con qué frecuencia se llaman. Este tipo de salida es útil porque le ayuda a comprender dónde debería mirar primero cuando intente acelerar su código.

Supongamos que ha escrito una función que no es demasiado costosa pero que se llama muchas veces en su aplicación. También ha escrito una función que es costosa pero que solo se llama una vez. Si solo tiene tiempo para arreglar uno, ¿cuál será? Sin perfilar el código, es difícil saber cuál acelerará más su código.

Puede resolverlo utilizando el módulo `cProfile` de Python.

Nota

Si intenta importar el módulo `cProfile` pero obtiene un error, puede usar el módulo de `profile` en su lugar.

El módulo `cProfile` imprime algunos datos sobre cada método o función llamada mientras ejecuta su programa. Para cada llamada, te mostrará

- El número de veces que se produjo la llamada (`ncalls`)
- El tiempo dedicado solo a esa llamada, sin incluir las cosas que llama a su vez (tiempo de espera)
- El tiempo promedio que pasó solo en esa llamada, a través de las llamadas `ncalls` (llamada `percall`)
- El tiempo acumulado que pasó en esa llamada, incluido el tiempo que pasó en sub-llamadas (`cumtime`)

Esta información es útil porque expondrá cosas que son lentas, que tienen un gran tiempo acumulado, pero también expondrá cosas que son rápidas pero llamadas muchas veces. La siguiente lista muestra un programa de juguete que llama a una función 1000 veces. La llamada a la función tarda una cantidad aleatoria de tiempo, hasta 10 milisegundos, en ejecutarse.

Listado de código 4: Perfilar el rendimiento de la CPU de un programa

```
import random
import time


def an_expensive_function():
    # Ref_1
    execution_time = random.random() / 100
    time.sleep(execution_time)


if __name__ == '__main__':
    # Ref_2
    for _ in range(1000):
        an_expensive_function()
```

Ref_1 Toma una cantidad aleatoria de tiempo (hasta 10 milisegundos) para ejecutar

Ref_2 Ejecuta la función 1000 veces

Guarde este programa en un módulo `cpu_profiling.py`. Luego puede perfilarlo desde la línea de comando usando `cProfile`:

```
python -m cProfile --sort cumtime cpu_profiling.py
```

En una gran cantidad de llamadas, puede esperar que una función que demore entre 0 y 10 milisegundos demore aproximadamente 5 milisegundos en promedio (`percall`). Al llamarlo 1000 veces (`ncalls`), puede esperar que tome unos 5 segundos en total (`cumtime`). Ejecute `cProfile` en el programa para ver si cumple con sus expectativas.

Verá una gran cantidad de resultados, pero la clasificación por tiempo acumulado significa que las llamadas a una función de alto costo estarán cerca de la parte superior:

```
$ python -m cProfile --sort cumtime cpu_profiling.py
         5138 function calls (5095 primitive calls) in
            5.644 seconds

    Ordered by: cumulative time

   ncalls   tottime   percal  cumtime    percall
filename:lineno(function)
      4/1    0.000    0.000     5.644      5.644 {built-in
          method
builtins.exec}
        1    0.002    0.002     5.644      5.644
cpu_profiling.py:1(<module>)
     1000    0.003    0.000     5.625      0.006
        cpu_profiling.py:5
   (an_expensive_function)
```

```
     1000    5.622    0.006     5.622      0.006 {built-in
         method
time.sleep}
     ...
```

En esta ejecución, an_expensive_function tomó un promedio de aproximadamente 6 milisegundos por llamada en el lapso de 1000 llamadas, lo que condujo a un acumulado de 5.625 segundos dentro de esa función.

Al mirar la salida de cProfile, querrá buscar llamadas con un alto valor de percall o un gran salto en cumtime. Estas características significan que la llamada ocupa una buena parte del tiempo de ejecución de su programa. Acelerar una función lenta puede mejorar la velocidad del programa bastante, y reducir el tiempo de ejecución de una función que se llama miles de veces puede ser una gran victoria.

5. Pruébalo

Considere el siguiente código. Contiene una función, sort_expensive, que tiene que ordenar una lista de 1000 enteros en el rango 0–999,999. También contiene una función, sort_cheap, que solo tiene que ordenar una lista de 10 enteros en el rango 0–999.

Los algoritmos de clasificación son generalmente más caros que $O(1)$, por lo que la función sort_expensive tomará más tiempo que sort_cheap. Si solo ejecutó cada función una vez, sort_cheap seguramente ganaría. Pero si necesita ejecutar sort_cheap 1,000 veces, no está claro qué operación será más rápida.

```
import random


def sort_expensive():
    the_list = random.sample(range(1_000_000), 1_000)
```

```
    the_list.sort()


def sort_cheap():
    the_list = random.sample(range(1_000), 10)
    the_list.sort()


if __name__ == '__main__':
    sort_expensive()
    for i in range(1000):
        sort_cheap()
```

Debe crear un perfil del código para comprender el rendimiento. Vea cómo le va a cada tarea utilizando los módulos `timeit` y `cProfile`.

Resumen

- Diseñe para un rendimiento tanto inicial como iterativo a lo largo de su desarrollo.
- Piense detenidamente sobre el tipo de datos correcto para la tarea.
- Prefiera generadores a listas cuando no necesite todos los valores a la vez, para ahorrar en el uso de memoria.
- Use los módulos Python `timeit` y `cProfile`/`profile` para probar sus hipótesis sobre la complejidad y el rendimiento.