Módulo 4 (Intermedio): Generadores

Generadores, donde encontrarlos

Generador - ¿Con qué asocias esta palabra? Quizás se refiere a algún dispositivo electrónico. O tal vez se refiere a una máquina pesada diseñada para producir energía eléctrica u otra cosa.

Un generador de Python es **un fragmento de código especializado capaz de producir una serie de valores y controlar el proceso de iteración**. Esta es la razón por la cual los generadores a menudo se llaman **iteradores**, y aunque hay quienes pueden encontrar una diferencia entre estos dos, aquí los trataremos como uno mismo.

Puede que no te hayas dado cuenta, pero te has topado con generadores muchas, muchas veces antes. Echa un vistazo al fragmento de código:

```
for i in range(5):
    print(i)
```

La función `range()` es un generador, la cual también es un iterador.

¿Cuál es la diferencia?

Una función devuelve un valor bien definido, el cual, puede ser el resultado de una evaluación compleja, por ejemplo, de un polinomio, y se invoca una vez, solo una vez.

Un generador **devuelve una serie de valores**, y en general, se invoca (implícitamente) más de una vez.

En el ejemplo, el generador `range()` se invoca seis veces, proporcionando cinco valores de cero a cuatro.

El proceso anterior es completamente transparente. Vamos a arrojar algo de luz sobre el. Vamos a mostrarte el protocolo iterador.

El **protocolo iterador es una forma en que un objeto debe comportarse para ajustarse a las reglas impuestas por el contexto de las sentencias *for* e *in***. Un objeto conforme al protocolo iterador se llama iterador.

Un iterador debe proporcionar dos métodos:

- `__iter__()` el cual debe **devolver el objeto en sí** y que se invoca una vez (es necesario para que Python inicie con éxito la iteración).
- `__next__()` el cual debe **devolver el siguiente valor** (primero, segundo, etc.) de la serie deseada: será invocado por las sentencias for/in para pasar a la siguiente iteración; si no hay más valores a proporcionar, el método deberá **generar la excepción** *StopIteration*.

¿Suena extraño? De ningúna manera:

```
class Fib:
    def __init__(self, nn):
        print("__init__")
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1

    def __iter__(self):
```

```
        print("__iter__")
        return self

    def __next__(self):
        print("__next__")
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret


for i in Fib(10):
    print(i)
```

Hemos creado una clase capaz de iterar a través de los primeros n valores (donde n es un parámetro del constructor) de los números de Fibonacci.

Permítenos recordarte: los números de Fibonacci(Fibi) se definen de la siguiente manera:

Fib_1

1

Fib_2

1

Fib_i

Fib_i-1 + Fib_i-2

En otras palabras:

- Los primeros dos números de la serie Fibonacci son 1.
- Cualquier otro número de Fibonacci es la suma de los dos anteriores (por ejemplo, Fib3 = 2, Fib4 = 3, Fib5 = 5, y así sucesivamente).

Vamos a ver el código:

- Las líneas 2 a 6: el constructor de la clase imprime un mensaje (lo usaremos para rastrear el comportamiento de la clase), se preparan algunas variables: (__n para almacenar el límite de la serie, __i

para rastrear el número actual de la serie Fibonacci, y __p1 junto con __p2 para guardar los dos números anteriores).

- Las líneas 8 a 10: el método __iter__ está obligado a devolver el objeto iterador en sí mismo; su propósito puede ser un poco ambiguo aquí, pero no hay misterio; trata de imaginar un objeto que no sea un iterador (por ejemplo, es una colección de algunas entidades), pero uno de sus componentes es un iterador capaz de escanear la colección; el método __iter__ debe **extraer el iterador y confiarle la ejecución del protocolo de iteración**; como puedes ver, el método comienza su acción imprimiendo un mensaje.
- Las líneas 12 a 21: el método __next__ es responsable de crear la secuencia; es algo largo, pero esto debería hacerlo más legible; primero, imprime un mensaje, luego actualiza el número de valores deseados y, si llega al final de la secuencia, el método interrumpe la iteración al generar la excepción StopIteration; el resto del código es simple y refleja con precisión la definición que te mostramos anteriormente.
- Las líneas 24 y 25 hacen uso del iterador.

El código produce el siguiente resultado:

```
__init__
__iter__
__next__
1
__next__
1
__next__
2
__next__
3
__next__
5
__next__
8
__next__
13
__next__
21
__next__
34
__next__
55
__next__
```

Observa:

- El objeto iterador se instancia primero.
- Después, Python invoca el método __iter__ para acceder al iterador real.
- El método __next__ se invoca once veces: las primeras diez veces produce valores útiles, mientras que la ultima finaliza la iteración.

El ejemplo muestra una solución donde el **objeto iterador es parte de una clase más compleja**.

```
class Fib:
    def __init__(self, nn):
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1
```

```
    def __iter__(self):
        print("Fib iter")
        return self

    def __next__(self):
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret


class Class:
    def __init__(self, n):
        self.__iter = Fib(n)

    def __iter__(self):
        print("Class iter")
        return self.__iter


object = Class(8)

for i in object:
    print(i)
```

El código no es sofisticado, pero presenta el concepto de una manera clara.

Echa un vistazo al código en el editor.

Hemos colocado el iterador *Fib* dentro de otra clase (podemos decir que lo hemos compuesto dentro de la clase *Class*). Se instancia junto con el objeto de *Class*.

El objeto de la clase se puede usar como un iterador cuando (y solo cuando) responde positivamente a la invocación __iter__ - esta clase puede hacerlo, y si se invoca de esta manera, proporciona un objeto capaz de obedecer el protocolo de iteración.

Es por eso que la salida del código es la misma que anteriormente, aunque el objeto de la clase *Fib* no se usa explícitamente dentro del contexto del bucle *for*.

```
Class iter
1
1
2
3
5
8
13
21
```

La sentencia yield

El protocolo iterador no es difícil de entender y usar, pero también es indiscutible que **el protocolo es bastante inconveniente**.

La principal molestia que tiene es que **necesita guardar el estado de la iteración en las invocaciones subsecuentes de __iter__**.

Por ejemplo, el iterador *Fib* se ve obligado a almacenar con precisión el lugar en el que se detuvo la última invocación (es decir, el número evaluado y los valores de los dos elementos anteriores). Esto hace que el código sea más grande y menos comprensible.

Es por eso que Python ofrece una forma mucho más efectiva, conveniente y elegante de escribir iteradores.

El concepto se basa fundamentalmente en un mecanismo muy específico proporcionado por la palabra clave reservada *yield*.

Se puede ver a la palabra clave reservada *yield* como un hermano más inteligente de la sentencia *return*, con una diferencia esencial.

Echa un vistazo a esta función:

```
def fun(n):
    for i in range(n):
        return i
```

Se ve extraño, ¿no? Está claro que el bucle *for* no tiene posibilidad de terminar su primera ejecución, ya que el *return* lo romperá irrevocablemente.

Además, invocar la función no cambiará nada: el bucle *for* comenzará desde cero y se romperá inmediatamente.

Podemos decir que dicha función no puede guardar y restaurar su estado en invocaciones posteriores.

Esto también significa que una función como esta **no se puede usar como generador**.

Hemos reemplazado exactamente una palabra en el código, ¿puedes verla?

```
def fun(n):
    for i in range(n):
        yield i
```

Hemos puesto *yield* en lugar de *return*. Esta pequeña enmienda convierte la función en un generador, y el ejecutar la sentencia *yield* tiene algunos efectos muy interesantes.

Primeramente, proporciona el valor de la expresión especificada después de la palabra clave reservada *yield*, al igual que *return*, pero no pierde el estado de la función.

Todos los valores de las variables están congelados y esperan la próxima invocación, cuando se reanuda la ejecución (no desde cero, como ocurre después de un *return*).

Hay una limitación importante: dicha **función no debe invocarse explícitamente** ya que no es una función; **es un objeto generador**.

La invocación devolverá **el identificador del objeto**, no la serie que esperamos del generador.

Debido a las mismas razones, la función anterior (la que tiene el return) solo se puede invocar explícitamente y

no se debe usar como generador.

Cómo construir un generador:

Permítenos mostrarte el nuevo generador en acción.

Así es como podemos usarlo:

```
def fun(n):
    for i in range(n):
        yield i


for v in fun(5):
    print(v)
```

¿Puedes adivinar la salida?

```
0
1
2
3
4
```

Cómo construir tu propio generador

¿Qué pasa si necesitas un **generador para producir las primeras n potencias de 2** ?

Nada difícil. Solo mira el código en el editor.

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


for v in powers_of_2(8):
    print(v)
```

¿Puedes adivinar la salida? Ejecuta el código para verificar tus conjeturas.

Listas por comprensión

Los generadores también se pueden usar con **listas por comprensión**, justo como aquí:

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
```

```
        power *= 2


t = [x for x in powers_of_2(5)]
print(t)
```

Ejecuta el ejemplo y verifica la salida.

La función list()

La función `list()` puede transformar una serie de invocaciones de generador subsequentes en una **lista real**:

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


t = list(powers_of_2(3))
print(t)
```

Nuevamente, intenta predecir el resultado y ejecuta el código para verificar tus predicciones.

El operador in

Además, el contexto creado por el operador *in* también te permite usar un generador.

El ejemplo muestra cómo hacerlo:

```
def powers_of_2(n):
    power = 1
    for i in range(n):
        yield power
        power *= 2


for i in range(20):
    if i in powers_of_2(4):
        print(i)
```

¿Cuál es la salida del código? Ejecuta el programa y verifica.

El generador de números Fibonacci

Ahora veamos un generador de números de la serie Fibonacci, asegurandonos que se vea mucho mejor que la versión orientada a objetos basada en la implementación directa del protocolo iterador.

Aquí está:

```
def fibonacci(n):
    p = pp = 1
```

```
    for i in range(n):
        if i in [0, 1]:
            yield 1
        else:
            n = p + pp
            pp, p = p, n
            yield n

fibs = list(fibonacci(10))
print(fibs)
```

Adivina la salida (una lista) producida por el generador y ejecuta el código para verificar si tenías razón.

Debes poder recordar las reglas que rigen la creación y el uso de un fenómeno de Python llamado **listas por comprensión: una forma simple de crear listas y sus contenidos**.

```
list_1 = []

for ex in range(6):
    list_1.append(10 ** ex)

list_2 = [10 ** ex for ex in range(6)]

print(list_1)
print(list_2)
```

Existen dos partes dentro del código, ambas crean una lista que contiene algunas de las primeras potencias naturales de diez.

La primer parte utiliza una forma rutinaria del bucle *for*, mientras que la segunda hace uso de listas por comprensión y construye la lista en el momento, sin necesidad de un bucle o cualquier otro código.

Pareciera que la lista se crea dentro de sí misma; esto es falso, ya que Python tiene que realizar casi las mismas operaciones que en la primera parte, pero el segundo formalismo es simplemente más elegante y le evita al lector cualquier detalle innecesario.

El ejemplo genera dos líneas idénticas que contienen el siguiente texto:

```
[1, 10, 100, 1000, 10000, 100000]
[1, 10, 100, 1000, 10000, 100000]
```

Hay una sintaxis muy interesante que queremos mostrarte ahora. Su usabilidad no se limita a listas por comprensión.

Es una **expresión condicional: una forma de seleccionar uno de dos valores diferentes en función del resultado de una expresión Booleana**.

Observa:

```
expresión_uno if condición else expresión_dos
```

Puede parecer un poco sorprendente a primera vista, pero hay que tener en cuenta que **no es una instrucción condicional**. Además, no es una instrucción en lo absoluto. Es un operador.

El valor que proporciona es expresión_uno cuando la condición es True (verdadero), y expresión_dos cuando sea

falso.

```
the_list = []

for x in range(10):
    the_list.append(1 if x % 2 == 0 else 0)

print(the_list)
```

El código llena una lista con *unos* y *ceros*, si el índice de un elemento particular es impar, el elemento se establece a 0, y a 1 de lo contrario.

¿Simple? Quizás no a primera vista. ¿Elegante? Indiscutiblemente.

¿Se puede usar el mismo truco dentro de una comprensión de lista? Sí, por supuesto.

```
the_list = [1 if x % 2 == 0 else 0 for x in range(10)]

print(the_list)
```

Compacto y elegante: estas dos palabras vienen a la mente al mirar el código.

Entonces, ¿qué tienen en común, generadores y listas por comprensión? ¿Hay alguna conexión entre ellos? Si. Una conexión algo suelta, pero inequívoca.

Solo un cambio puede **convertir cualquier comprensión en un generador**.

Listas por comprensión frente a generadores

Ahora observa el código a continuación y ve si puedes encontrar el detalle que convierte una comprensión de lista en un generador:

```
the_list = [1 if x % 2 == 0 else 0 for x in range(10)]
the_generator = (1 if x % 2 == 0 else 0 for x in range(10))

for v in the_list:
    print(v, end=" ")
print()

for v in the_generator:
    print(v, end=" ")
print()
```

Son los **paréntesis**. Los corchetes hacen una comprensión, los paréntesis hacen un generador.

El código, cuando se ejecuta, produce dos líneas idénticas:

```
1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0
```

¿Cómo puedes saber que la segunda asignación crea un generador, no una lista?

Hay algunas pruebas que podemos mostrarte. Aplica la función `len()` a ambas entidades.

`len(the_list)` dará como resultado 10. Claro y predecible. `len(the_generator)` generará una excepción,

y verás el siguiente mensaje:

```
TypeError: object of type 'generator' has no len()
```

Por supuesto, guardar la lista o el generador no es necesario; puedes crearlos exactamente en el lugar donde los necesites, justo como aquí:

```
for v in [1 if x % 2 == 0 else 0 for x in range(10)]:
    print(v, end=" ")
print()

for v in (1 if x % 2 == 0 else 0 for x in range(10)):
    print(v, end=" ")
print()
```

Nota: la misma apariencia de la salida no significa que ambos bucles funcionen de la misma manera. En el primer bucle, la lista se crea (y se itera) como un todo; en realidad, existe cuando se ejecuta el bucle.

En el segundo bucle, no hay ninguna lista, solo hay valores subsecuentes producidos por el generador, uno por uno.

La función lambda

La función lambda es un concepto tomado de las matemáticas, más específicamente, de una parte llamada el *Cálculo Lambda*, pero estos dos fenómenos no son iguales.

Los matemáticos usan el *Cálculo Lambda* en sistemas formales conectados con: la lógica, la recursividad o la demostrabilidad de teoremas. Los programadores usan la función *lambda* para simplificar el código, hacerlo más claro y fácil de entender.

Una función lambda es una función sin nombre (también puedes llamarla **una función anónima**). Por supuesto, tal afirmación plantea inmediatamente la pregunta: ¿cómo se usa algo que no se puede identificar?

Afortunadamente, no es un problema, ya que se puede mandar llamar dicha función si realmente se necesita, pero, en muchos casos la función *lambda* puede existir y funcionar mientras permanece completamente de incógnito.

La declaración de la función lambda no se parece a una declaración de función normal; compruébalo tu mismo:

```
lambda parámetros: expresión
```

Tal cláusula devuelve el valor de la expresión al tomar en cuenta el valor del argumento lambda actual.

Como de costumbre, un ejemplo será útil. Nuestro ejemplo usa tres funciones lambda, pero con nombres. Analízalo cuidadosamente:

```
two = lambda: 2
sqr = lambda x: x * x
pwr = lambda x, y: x ** y

for a in range(-2, 3):
    print(sqr(a), end=" ")
```

```
    print(pwr(a, two()))
```

Vamos a analizarlo:

La primer *lambda* es una función anónima **sin parámetros** que siempre devuelve un 2. Como se ha a**signado a una variable llamada dos**, podemos decir que la función ya no es anónima, y se puede usar su nombre para invocarla.

La segunda es una **función anónima de un parámetro** que devuelve el valor de su argumento al cuadrado. Se ha nombrado también como tal.

La tercer lambda **toma dos parámetros** y devuelve el valor del primero elevado al segundo. El nombre de la variable que lleva la lambda habla por si mismo. No se utiliza pow para evitar confusiones con la función incorporada del mismo nombre y el mismo propósito.

El programa produce el siguiente resultado:

```
4 4
1 1
0 0
1 1
4 4
```

Este ejemplo es lo suficientemente claro como para mostrar como se declaran las funciones lambda y cómo se comportan, pero no dice nada acerca de por que son necesarias y para qué se usan, ya que se pueden reemplazar con funciones de Python de rutina.

¿Cómo usar lambdas y para qué?

La parte más interesante de usar lambdas aparece cuando puedes usarlas en su forma pura: **como partes anónimas de código destinadas a evaluar un resultado**.

Imagina que necesitamos una función (la nombraremos *print_function*) que imprime los valores de una (otra) función dada para un conjunto de argumentos seleccionados.

Queremos que *print_function* sea universal, debería aceptar un conjunto de argumentos incluidos en una lista y una función a ser evaluada, ambos como argumentos; no queremos codificar nada.

```
def print_function(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')


def poly(x):
    return 2 * x**2 - 4 * x + 2


print_function([x for x in range(-2, 3)], poly)
```

Analicémoslo. La función `print_function()` toma dos parámetros:

- El primero, una lista de argumentos para los que queremos imprimir los resultados.
- El segundo, una función que debe invocarse tantas veces como el número de valores que se recopilan dentro del primer parámetro.

Nota: También hemos definido una función llamada `poly()`, esta es la función cuyos valores vamos a imprimir.

El cálculo que realiza la función no es muy sofisticado: es el polinomio (de ahí su nombre) de la forma:

```
f(x) = 2x2 - 4x + 2
```

El nombre de la función se pasa a `print_function()` junto con un conjunto de cinco argumentos diferentes: el conjunto está construido con una cláusula de comprensión de la lista.

El código imprime las siguientes líneas:

```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

¿Podemos evitar definir la función `poly()`, ya que no la vamos a usar más de una vez? Si, podemos: este es el beneficio que puede aportar una función lambda.

Mira el ejemplo de abajo. ¿Puedes ver la diferencia?

```
def print_function(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')

print_function([x for x in range(-2, 3)], lambda x: 2 * x**2 - 4 * x + 2)
```

La función `print_function()` se ha mantenido exactamente igual, pero no hay una función `poly()`. Ya no la necesitamos, ya que el polinomio ahora está directamente dentro de la invocación de la función `print_function()` en forma de una lambda definida de la siguiente manera:

```
lambda x: 2 * x**2 - 4 * x + 2
```

El código se ha vuelto más corto, más claro y más legible.

Permítenos mostrarte otro lugar donde las lambdas pueden ser útiles. Comenzaremos con una descripción de `map()`, una función integrada de Python. Su nombre no es demasiado descriptivo, su idea es simple y la función en sí es muy utilizable.

Lambdas y la función map()

En el más simple de todos los casos posibles, la función `map()`:

```
map(function, list)
```

Toma dos argumentos:

- Una función.
- Una lista.

La descripción anterior está extremadamente simplificada, ya que:

- El segundo argumento `map()` puede ser cualquier entidad que se pueda iterar (por ejemplo, una tupla o un generador).

- `map()` puede aceptar más de dos argumentos.

La función `map()` **aplica la función pasada por su primer argumento a todos los elementos de su segundo argumento y devuelve un iterador que entrega todos los resultados de funciones subsequentes**.

Puedes usar el iterador resultante en un bucle o convertirlo en una lista usando la función `list()`.

¿Puedes ver un papel para una lambda aquí?

```
list_1 = [x for x in range(5)]
list_2 = list(map(lambda x: 2 ** x, list_1))
print(list_2)

for x in map(lambda x: x * x, list_2):
    print(x, end=' ')
print()
```

Hemos usado dos lambdas en él.

Esta es la explicación:

- Se construye la list_1 con valores del 0 al 4.
- Después, se utiliza `map()` junto con la primer lambda para crear una nueva lista en la que todos los elementos han sido evaluados como 2 elevado a la potencia tomada del elemento correspondiente de list_1.
- list_2 se imprime.
- En el siguiente paso, se usa nuevamente la función `map()` para hacer uso del generador que devuelve, e imprimir directamente todos los valores que entrega; como puedes ver, hemos usado la segunda lambda aquí - solo eleva al cuadrado cada elemento de list_2.

Intenta imaginar el mismo código sin lambdas. ¿Sería mejor? Es improbable.

Lambdas y la función filter()

Otra función de Python que se puede embellecer significativamente mediante la aplicación de una lambda es `filter()`.

Espera el mismo tipo de argumentos que `map()`, pero hace algo diferente: **filtra su segundo argumento mientras es guiado por direcciones que fluyen desde la función especificada en el primer argumento** (la función se invoca para cada elemento de la lista, al igual que en `map()`).

Los elementos que devuelven True de la función **pasan el filtro**, los otros son rechazados.

```
from random import seed, randint

seed()
data = [randint(-10,10) for x in range(5)]
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

print(data)
print(filtered)
```

Nota: hemos hecho uso del módulo *random* para inicializar el generador de números aleatorios (que no debe confundirse con los generadores de los que acabamos de hablar) con la función `seed()`, para producir cinco valores enteros aleatorios de -10 a 10 usando la función `randint()`.

Luego se filtra la lista y solo se aceptan los números que son pares y mayores que cero.

Por supuesto, no es probable que recibas los mismos resultados, pero así es como se veían nuestros resultados:

```
[6, 3, 3, 2, -7]
[6, 2]
```

Una breve explicación de cierres

Comencemos con una definición: **cierres es una técnica que permite almacenar valores a pesar de que el contexto en el que se crearon ya no existe**. ¿Complicado? Un poco.

Analicemos un ejemplo simple:

```
def outer(par):
    loc = par


var = 1
outer(var)

print(var)
print(loc)
```

El ejemplo es obviamente erróneo.

Las dos últimas líneas provocarán una excepción *NameError* - ni par ni loc son accesibles fuera de la función. Ambas variables existen cuando y solo cuando la función `outer()` esta siendo ejecutada.

```
def outer(par):
    loc = par

    def inner():
        return loc
    return inner


var = 1
fun = outer(var)
print(fun())
```

Hay un elemento completamente nuevo - una función (llamada *inner*) dentro de otra función (llamada *outer*).

¿Cómo funciona? Como cualquier otra función excepto por el hecho de que `inner()` solo se puede invocar desde dentro de `outer()`. Podemos decir que `inner()` es una herramienta privada de `outer()`, ninguna otra parte del código la puede acceder.

Observa cuidadosamente:

- La función `inner()` devuelve el valor de la variable accesible dentro de su alcance, ya que `inner()` puede utilizar cualquiera de las entidades a disposición de `outer()`.
- La función `outer()` devuelve la función `inner()` por si misma; mejor dicho, devuelve una copia de la función `inner()` al momento de la invocación de la función `outer()`; la función congelada contiene su

entorno completo, incluido el estado de todas las variables locales, lo que también significa que el valor de loc se retiene con éxito, aunque `outer()` ya ha dejado de existir.

En efecto, el código es totalmente válido y genera:

```
1
```

La función devuelta durante la invocación de `outer()` es un **cierre**.

Un cierre se debe invocar exactamente de la misma manera en que se ha declarado.

En el ejemplo anterior (vea el código a continuación):

```
def outer(par):
    loc = par

    def inner():
        return loc
    return inner


var = 1
fun = outer(var)
print(fun())
```

La función `inner()` no tenía parámetros, por lo que tuvimos que invocarla sin argumentos.

```
def make_closure(par):
    loc = par

    def power(p):
        return p ** loc
    return power


fsqr = make_closure(2)
fcub = make_closure(3)

for i in range(5):
    print(i, fsqr(i), fcub(i))
```

Es totalmente posible **declarar un cierre equipado con un número arbitrario de parámetros**, por ejemplo, al igual que la función `power()`.

Esto significa que el cierre no solo utiliza el ambiente congelado, sino que también puede **modificar su comportamiento utilizando valores tomados del exterior**.

Este ejemplo muestra una circunstancia más interesante: puedes **crear tantos cierres como quieras usando el mismo código**. Esto se hace con una función llamada `make_closure()`. Nota:

- El primer cierre obtenido de make_closure() define una herramienta que eleva al cuadrado su argumento.
- El segundo está diseñado para elevar el argumento al cubo.

Es por eso que el código produce el siguiente resultado:

```
0 0 0
1 1 1
```

```
2 4 8
3 9 27
4 16 64
```

Puntos Clave

1. Un **iterator** es un objeto de una clase que proporciona al menos **dos** métodos (sin contar el constructor):

- `__iter__()` se invoca una vez cuando se crea el iterador y devuelve el **propio** objeto del iterador.
- `__next__()` se invoca para proporcionar **el valor de la siguiente iteración** y genera la excepción StopIteration cuando la iteración **llega a su fin**.

2. La sentencia *yield* solo puede ser utilizada dentro de funciones. La sentencia *yield* suspende la ejecución de la función y hace que la función regrese el argumento de yield como resultado. Esta función no puede invocarse de forma regular, su único propósito es ser utilizada como un **generador** (es decir, en un contexto que requiera una serie de valores, como un bucle for).

3. Una **expresión condicional** es una expresión construida usando el operador if-else. Por ejemplo:

```
print(True if 0 >= 0 else False)
```

Da como salida: True.

4. Una **lista por comprensión** se convierte en un **generador** cuando se emplea dentro de **paréntesis** (usado entre corchetes, produce una lista regular). Por ejemplo:

```
for x in (el * 2 for el in range(5)):
    print(x)
```

Da como salida: 02468.

5. Una **función lambda** es una herramienta para crear **funciones anónimas**. Por ejemplo:

```
def foo(x, f):
    return f(x)

print(foo(9, lambda x: x ** 0.5))
```

Da como salida: 3.0.

6. La función `map(fun, list)` crea una copia del argumento list, y aplica la función fun a todos sus elementos, retornando un generador que proporciona el nuevo contenido de la lista elemento por elemento. Por ejemplo:

```
short_list = ['mython', 'python', 'fell', 'on', 'the', 'floor']
new_list = list(map(lambda s: s.title(), short_list))
print(new_list)
```

Da como salida: ['Mython', 'Python', 'Fell', 'On', 'The', 'Floor'].

7. La función `filter(fun, list)` crea una copia de aquellos elementos de list, lo cual hace que la función fun retorne True. El resultado de la función es un generador proporcionando el nuevo contenido de la lista elemento por elemento. Por ejemplo

```
short_list = [1, "Python", -1, "Monty"]
new_list = list(filter(lambda s: isinstance(s, str), short_list))
print(new_list)
```

Da como salida: ['Python', 'Monty'].

8. Un cierre es una técnica que permite almacenar valores a pesar de que el contexto en el que han sido creados no existe más. Por ejemplo:

```
def tag(tg):
    tg2 = tg
    tg2 = tg[0] + '/' + tg[1:]

    def inner(str):
        return tg + str + tg2
    return inner

b_tag = tag('<b>')
print(b_tag('Monty Python'))
```

Da como salida: Monty Python