
HOWTO - Programación funcional

Versión 3.11.11

Guido van Rossum and the Python development team

diciembre 06, 2024

**Python Software Foundation
Email: docs@python.org**

Índice general

1	Introducción	2
1.1	Demostrabilidad formal	3
1.2	Modularidad	3
1.3	Facilidad de depurar y probar	4
1.4	Componibilidad	4
2	Iteradores	4
2.1	Tipos de datos que soportan iteradores	5
3	Expresiones generadoras y listas por comprensión	6
4	Generadores	8
4.1	Pasar valores a un generador	9
5	Funciones incorporadas	10
6	El módulo itertools	12
6.1	Crear nuevos iteradores	12
6.2	Aplicar funciones a los elementos	13
6.3	Seleccionar elementos	14
6.4	Funciones combinatorias	14
6.5	Agrupar elementos	15
7	El módulo functools	16
7.1	El módulo operator	17
8	Funciones pequeñas y la expresión lambda	17
9	Historia de revisiones y reconocimientos	19
10	Referencias	19
10.1	Generales	19
10.2	Específicas de Python	19
10.3	Documentación de Python	19
	Índice	20

Autor

A. M. Kuchling

Publicación

0.32

En este documento, haremos un recorrido de las características de Python adecuadas para implementar programas en un estilo funcional. Después de una introducción de los conceptos de programación funcional, veremos las características del lenguaje como iteradores y generadores y módulos de librería relevantes como `itertools` y `functools`.

1 Introducción

Esta sección explica el concepto básico de programación funcional; si solo está interesado en aprender acerca de las características del lenguaje Python, pase a la siguiente sección en [iteradores](#).

Los lenguajes de programación soportan la descomposición de problemas en muchas formas diferentes:

- La mayoría de los lenguajes de programación son **procedimentales**: los programas son listas de instrucciones que le dicen a la computadora qué hacer con la entrada del programa. C, Pascal e incluso las terminales Unix son lenguajes procedimentales.
- En los lenguajes **declarativos**, se escribe una especificación que describe el problema que se resolverá, y la implementación del lenguaje averigua como realizar el cálculo de forma eficiente. SQL es el lenguaje declarativo con el que probablemente esté más familiarizado; una consulta SQL describe el conjunto de datos que quiere recuperar, y el motor SQL decide si escanear tablas o usar índices, qué subcláusulas deben ejecutarse primero, etc.
- Los programas **orientados a objetos** manipulan colecciones de objetos. Los objetos tienen estado interno y soportan métodos que consultan o modifican su estado interno de alguna manera. Smalltalk y Java son lenguajes orientados a objetos. C++ y Python son lenguajes que soportan la programación orientada a objetos, pero no fuerzan el uso de las características orientadas a objetos.
- La programación **funcional** descompone un problema en un conjunto de funciones. Idealmente, las funciones solo reciben entradas y producen salidas, y no tienen ningún estado interno que afecte la salida producida para una entrada dada. Los lenguajes funcionales bien conocidos incluyen la familia ML (Standard ML, OCaml, y otras variantes) y Haskell.

Los diseñadores de algunos lenguajes de computadora eligen enfatizar en un enfoque particular para programar. Esto a menudo hace difícil escribir programas que usen un enfoque diferente. Otros lenguajes son lenguajes multiparadigma que soportan varios enfoques diferentes. Lisp, C++ y Python son multiparadigma; puede escribir programas o librerías que son en gran parte procedimentales, orientados a objetos o funcionales en todos estos lenguajes. En un programa grande, las diferentes secciones podrían escribirse usando diferentes enfoques; la GUI podría ser orientada a objetos mientras la lógica de procesamiento es procedimental o funcional, por ejemplo.

En un programa funcional, la entrada fluye a través de un conjunto de funciones. Cada función opera sobre su entrada y produce alguna salida. El estilo funcional desalienta las funciones con efectos secundarios que modifican el estado interno o hacen otros cambios que no son visibles en el valor de retorno de la función. Las funciones que no tienen ningún efecto secundario se llaman **puramente funcionales**. Evitar los efectos secundarios significa no usar estructuras de datos que se actualicen mientras se ejecuta un programa; cada salida de la función debe depender solo de su entrada.

Algunos lenguajes son demasiado estrictos y ni siquiera tienen declaraciones de asignación como `a=3` o `c = a + b`, pero es difícil evitar todos los efectos secundarios, como imprimir en pantalla o escribir en un archivo de disco. Otro ejemplo es llamar a las funciones `print()` o `time.sleep()`, ninguna de las dos retorna un valor útil, ambas son llamadas solo por sus efectos secundarios de enviar texto a la pantalla o pausar la ejecución por un segundo.

Los programas de Python escritos en estilo funcional usualmente no irán al extremo de evitar todas las E/S o todas las asignaciones; en cambio, proveerán una interfaz aparentemente funcional pero internamente usará características no funcionales. Por ejemplo, la implementación de una función todavía usará asignaciones a variables locales, pero no modificará variables globales ni tendrá otros efectos secundarios.

La programación funcional se puede considerar lo opuesto a la programación orientada a objetos. Los objetos son pequeñas capsulas que contienen algún estado interno junto con una colección de llamadas a métodos que le permiten modificar este estado, y los programas consisten en realizar el conjunto correcto de cambios de estado. La programación funcional quiere evitar cambios de estado tanto como sea posible y trabaja con flujos de datos entre funciones. En Python podría combinar los dos enfoques para escribir funciones que reciban y retornen instancias que representen objetos en su aplicación (mensajes de e-mail, transacciones, etc.).

El diseño funcional puede parecer una restricción extraña bajo la cuál trabajar. ¿Por qué evitaría objetos y efectos secundarios? Hay ventajas teóricas y prácticas para el estilo funcional:

- Demostrabilidad formal.
- Modularidad.
- Componibilidad.
- Facilidad de depurar y probar.

1.1 Demostrabilidad formal

Un beneficio teórico es que es más fácil construir una demostración matemática de que un programa funcional es correcto.

Por un largo tiempo los investigadores se han interesado en buscar formas de demostrar matemáticamente que los programas son correctos. Esto es diferente de probar un programa sobre numerosas entradas y concluir que su salida es usualmente correcta, o leer el código fuente de un programa y concluir que el código se ve bien; en lugar de eso el objetivo es una demostración rigurosa de que un programa produce el resultado correcto para todas las entradas posibles.

La técnica utilizada para demostrar que los programas son correctos es anotar **invariantes**, propiedades de los datos de entrada y de las variables del programa que siempre son verdaderas. Por cada línea de código, debe mostrar que si las invariantes *X* e *Y* son verdaderas **antes** que la línea sea ejecutada, las invariantes ligeramente diferentes *X'* e *Y'* son verdaderas **después** que la línea se ejecutó. Esto continúa hasta que alcance el fin del programa, punto en el cuál las invariantes deben coincidir con las condiciones deseadas en la salida del programa.

La evitación de las asignaciones de la programación funcional surge porque las asignaciones son difíciles de manejar con esta técnica; las asignaciones pueden romper invariantes que eran verdaderas antes de la asignación sin producir nuevas invariantes que se puedan propagar hacia adelante.

Desafortunadamente, demostrar que los programas son correctos es en gran parte impráctico y no relevante al software de Python. Incluso los programas triviales requieren demostraciones que tienen varias páginas; la demostración de correctitud para un programa moderadamente complicado sería enorme, y pocos o ninguno de los programas que usa diariamente (el interprete de Python, su analizador XML, su navegador web) se podrían demostrar que son correctos. Aún si anotó o generó una demostración, entonces estaría la cuestión de verificar la demostración; quizás hay un error en ella, y equivocadamente cree que demostró que el programa es correcto.

1.2 Modularidad

Un beneficio más práctico de la programación funcional es que fuerza a romper su problema en pequeñas piezas. Como resultado los programas son más modulares. Es más fácil especificar y escribir una función pequeña que hace una cosa que una función grande que realiza una transformación complicada. Las funciones pequeñas también son más fáciles de leer y comprobar si hay errores.

1.3 Facilidad de depurar y probar

Probar y depurar un programa en estilo funcional es más fácil.

La depuración se simplifica porque las funciones generalmente son pequeñas y claramente especificadas. Cuando un programa no funciona, cada función es un punto de interfaz donde puede comprobar si los datos son correctos. Puede ver las entradas y salidas intermedias para aislar rápidamente la función que es responsable de un error.

Las pruebas son más fáciles porque cada función es un sujeto potencial para una prueba unitaria. Las funciones no dependen de un estado del sistema que necesite ser replicado antes de correr una prueba; en lugar de eso solo tiene que sintetizar la entrada correcta y comprobar que la salida coincida con las expectativas.

1.4 Componibilidad

Mientras trabaja en un programa en estilo funcional, escribirá un número de funciones con diferentes entradas y salidas. Algunas de estas funciones inevitablemente estarán especializadas en una aplicación en particular, pero otras serán útiles en una amplia variedad de programas. Por ejemplo, una función que recibe la ruta de un directorio y retorna todos los archivos XML en el directorio, o una función que recibe el nombre de un archivo y retorna su contenido, se puede aplicar a muchas situaciones diferentes.

Con el tiempo formará una librería personal de utilidades. A menudo ensamblará nuevos programas organizando funciones existentes en una nueva configuración y escribiendo unas pocas funciones especializadas para la tarea actual.

2 Iteradores

Comenzaré viendo una característica del lenguaje Python que es una base importante para escribir programas en estilo funcional: iteradores.

Un iterador es un objeto que representa un flujo de datos; este objeto retorna los datos de a un elemento a la vez. Un iterador de Python debe soportar un método llamado `__next__()` que no recibe argumentos y siempre retorna el siguiente elemento en el flujo. Si no hay más elementos en el flujo, `__next__()` debe lanzar la excepción `StopIteration`. Los iteradores no tienen que ser finitos; es perfectamente razonable escribir un iterador que produce un flujo de datos infinito.

La función incorporada `iter()` recibe un objeto arbitrario e intenta retornar un iterador que retornará los contenidos o elementos del objeto, lanzando `TypeError` si el objeto no soporta iteración. Muchos tipos de datos integrados de Python soportan iteración, siendo los más comunes las listas y los diccionarios. Un objeto se llama iterable si puede obtener un iterador para él.

Puede experimentar con la interfaz de iteración manualmente:

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it
<...iterator object at ...>
>>> it.__next__() # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python espera objetos iterables en muchos contextos diferentes, siendo el más importante la sentencia `for`. En la sentencia `for X in Y`, `Y` debe ser un iterador o algún objeto para el que `iter()` puede crear un iterador. Estas dos sentencias son equivalentes:

```
for i in iter(obj):
    print(i)

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

Los iteradores se pueden materializar como listas o tuplas utilizando las funciones constructoras `list()` o `tuple()`:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

El desempaquetado de secuencias también soporta iteradores: si sabe que un iterador retornará `N` elementos, puede desempaquetarlos en una `N`-tupla:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> a, b, c = iterator
>>> a, b, c
(1, 2, 3)
```

Las funciones incorporadas como `max()` y `min()` pueden recibir un solo iterador como argumento y retornarán el elemento más grande o más pequeño. Los operadores `"in"` y `"not in"` también soportan iteradores: `X in iterator` es verdadero si `X` se encuentra en el flujo que retornó el iterador. Se encontrará con problemas obvios si el iterador es infinito; `max()`, `min()` nunca retornarán, y si el elemento `X` nunca aparece en el flujo, los operadores `"in"` y `"not in"` tampoco retornarán.

Note que solo puede ir hacia adelante en un iterador; no hay forma de obtener el elemento anterior, reiniciar el iterador o hacer una copia de él. Los objetos iteradores opcionalmente pueden proveer estas capacidades adicionales, pero el protocolo del iterador solo especifica el método `__next__()`. Por lo tanto las funciones pueden consumir toda la salida del iterador, y si necesita hacer algo diferente con el mismo flujo, tendrá que crear un nuevo iterador.

2.1 Tipos de datos que soportan iteradores

Ya hemos visto cómo las listas y tuplas soportan iteradores. De hecho, cualquier tipo de secuencia de Python, como cadenas de caracteres, automáticamente soportará la creación de un iterador.

Llamar a `iter()` en un diccionario retornará un iterador que recorrerá sobre las claves del diccionario:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print(key, m[key])
Jan 1
Feb 2
Mar 3
Apr 4
May 5
Jun 6
Jul 7
Aug 8
Sep 9
Oct 10
```

(continúe en la próxima página)

Nov 11
Dec 12

Note que a partir de Python 3.7, se garantiza que el orden de iteración del diccionario es el mismo que el orden de inserción. En versiones anteriores, el comportamiento no estaba especificado y podía variar entre implementaciones.

Aplicar `iter()` a un diccionario siempre recorre sobre las claves, pero los diccionarios tienen métodos que retornan otros iteradores. Si quiere iterar sobre valores o pares clave/valor, puede explícitamente llamar a los métodos `values()` o `items()` para obtener un iterador apropiado.

El constructor `dict()` puede aceptar un iterador que retorna un flujo finito de tuplas (`key, value`):

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'France': 'Paris', 'US': 'Washington DC'}
```

Los archivos también soportan iteración llamando al método `readline()` hasta que no haya más líneas en el archivo. Esto significa que puede leer cada línea de un archivo de esta forma:

```
for line in file:
    # do something for each line
    ...
```

Los conjuntos pueden recibir sus contenidos de un iterable y le permiten iterar sobre los elementos del conjunto:

```
>>> S = {2, 3, 5, 7, 11, 13}
>>> for i in S:
...     print(i)
2
3
5
7
11
13
```

3 Expresiones generadoras y listas por comprensión

Dos operaciones comunes en la salida de un iterador son 1) realizar alguna operación para cada elemento, 2) elegir un subconjunto de elementos que reúnen alguna condición. Por ejemplo, dada una lista de cadena de caracteres, podría querer remover los espacios finales de cada línea o extraer todas las cadenas de caracteres que contienen una subcadena de caracteres dada.

Las listas por comprensión y las expresiones generadoras (forma abreviada: «listcomps» y «genexps») son una notación concisa para tales operaciones, prestadas del lenguaje de programación funcional Haskell (<https://www.haskell.org/>). Puede remover todos los espacios de un flujo de cadena de caracteres con el siguiente código:

```
>>> line_list = [' line 1\n', 'line 2 \n', ' \n', '']

>>> # Generator expression -- returns iterator
>>> stripped_iter = (line.strip() for line in line_list)

>>> # List comprehension -- returns list
>>> stripped_list = [line.strip() for line in line_list]
```

Puede seleccionar solo ciertos elementos agregando una condición `"if"`:

```
>>> stripped_list = [line.strip() for line in line_list
...                  if line != ""]
```

Con una lista por comprensión, obtiene una lista de Python; `stripped_list` es una lista que contiene las líneas resultantes, no un iterador. Las expresiones generadoras retornan un iterador que calcula los valores cuando es necesario, sin necesidad de materializar todos los valores a la vez. Esto significa que las listas por comprensión son inútiles si está trabajando con iteradores que retornan un flujo infinito o una gran cantidad de datos. En estas situaciones son preferibles las expresiones generadoras.

Las expresiones generadoras están rodeadas por paréntesis («()») y las listas por comprensión están rodeadas por corchetes («[]»). Las expresiones generadoras tienen la forma:

```
( expression for expr in sequence1
    if condition1
    for expr2 in sequence2
    if condition2
    for expr3 in sequence3
    ...
    if condition3
    for exprN in sequenceN
    if conditionN )
```

Nuevamente, para una lista por comprensión solo los corchetes exteriores son diferentes (corchetes en lugar de paréntesis).

Los elementos de la salida generada serán valores sucesivos de `expression`. Las cláusulas `if` son todas opcionales; si está presente, `expression` es solo evaluado y añadido al resultado cuando `condition` es verdadero.

Las expresiones generadoras siempre se tienen que escribir dentro de paréntesis, pero los paréntesis que indican la llamada a una función también cuentan. Si quiere crear un iterador que se pase inmediatamente a una función puede escribir:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

Las cláusulas `for...in` contienen las secuencias sobre las que se itera. Las secuencias no tienen que tener la misma longitud, porque son iteradas de izquierda a derecha, **no** en paralelo. Por cada elemento en `sequence1`, se recorre `sequence2` desde el inicio. Luego se recorre `sequence3` por cada par de elementos resultante de `sequence1` y `sequence2`.

Para ponerlo en otra forma, una lista por comprensión o expresión generadora es equivalente al siguiente código Python:

```
for expr1 in sequence1:
    if not (condition1):
        continue # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue # Skip this element
        ...
        for exprN in sequenceN:
            if not (conditionN):
                continue # Skip this element

    # Output the value of
    # the expression.
```

Esto significa que cuando hay múltiples cláusulas `for...in` pero no cláusulas `if`, la longitud de la salida resultante será igual al producto de las longitudes de todas las secuencias. Si tiene dos listas de longitud 3, la lista de salida tendrá 9 elementos:

```
>>> seq1 = 'abc'
>>> seq2 = (1, 2, 3)
>>> [(x, y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

Para evitar introducir una ambigüedad en la gramática de Python, si `expression` está creando una tupla, debe estar rodeada de paréntesis. La primera lista por comprensión de abajo tiene un error de sintaxis, mientras que la segunda es correcta:

```
# Syntax error
[x, y for x in seq1 for y in seq2]
# Correct
[(x, y) for x in seq1 for y in seq2]
```

4 Generadores

Los generadores son una clase especial de funciones que simplifican la tarea de escribir iteradores. Las funciones regulares calculan un valor y lo retornan, pero los generadores retornan un iterador que retorna un flujo de valores.

Sin duda está relacionado con cómo funcionan las llamadas a funciones regulares en Python o C. Cuando llama a una función, esta obtiene su espacio de nombres privado donde se crean sus variables locales. Cuando la función alcanza una sentencia `return`, las variables locales son destruidas y el valor se retorna al llamador. Una llamada posterior a la misma función crea un nuevo espacio de nombres privado y un conjunto limpio de variables locales. Pero, ¿qué pasa si las variables locales no fueron desechadas en la salida de una función? ¿Qué pasa si más tarde podría reanudar la función desde donde quedó? Esto es lo que proveen los generadores; se pueden pensar como funciones que se reanudan.

Este es el ejemplo más simple de una función generadora:

```
>>> def generate_ints(N):
...     for i in range(N):
...         yield i
```

Cualquier función que contiene una palabra clave `yield` es una función generadora; esto es detectado por el compilador bytecode de Python que compila la función de forma especial como resultado.

Cuando llama a una función generadora, no retorna un solo valor; en lugar de eso retorna un objeto generador que soporta el protocolo iterador. Al ejecutar la expresión `yield`, el generador produce el valor de `i`, de forma similar a una sentencia `return`. La gran diferencia entre una sentencia `yield` y un `return` es que al alcanzar un `yield` se suspende el estado de ejecución del generador y se preservan las variables locales. En la próxima llamada al método `__next__()` del generador, la función reanudará la ejecución.

Este es un ejemplo de uso del generador `generate_ints()`:

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "stdin", line 1, in <module>
  File "stdin", line 2, in generate_ints
StopIteration
```

De igual forma podría escribir `for i in generate_ints(5), o a, b, c = generate_ints(3).`

Dentro de una función generadora, `return value` causa que se lance `StopIteration(value)` del método `__next__()`. Una vez que esto pase, o que se alcance el final de la función, termina la procesión de valores y el generador no puede producir más valores.

Podría lograr el efecto de los generadores manualmente escribiendo su propia clase y guardando todas las variables locales del generador como variables de instancia. Por ejemplo, retornar una lista de enteros se podría hacer estableciendo `self.count` a 0, y teniendo el método `__next__()` que incrementa `self.count` y lo retorna. Sin embargo, para un generador moderadamente complicado, escribir una clase correspondiente puede ser mucho más confuso.

El banco de pruebas incluido con la librería de Python, `Lib/test/test_generators.py`, contiene un número de ejemplos más interesantes. Este es un generador que implementa un recorrido inorden de un árbol usando generadores recursivamente.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Otros dos ejemplos en `test_generators.py` producen soluciones al problema de N-reinas (ubicar N reinas en un tablero de ajedrez de NxN de forma que ninguna reina amenace a otra) y el problema del caballo (encontrar una ruta que lleve a un caballo a cada cuadro de un tablero de ajedrez de NxN sin visitar ningún cuadro dos veces).

4.1 Pasar valores a un generador

En Python 2.4 y anteriores, los generadores solo producían salida. Una vez que el código de un generador era invocado para crear un iterador, no había forma de pasar ninguna información nueva a la función cuando se reanuda su ejecución. Podría lograr esta habilidad haciendo que el generador mire una variable global o pasando en algún objeto mutable que el llamador luego modifica, pero estos enfoques son confusos.

En Python 2.5 hay una forma más simple de pasar valores a un generador. `yield` se convirtió en una expresión, retornando un valor que se puede asignar a una variable o sobre el que se puede operar:

```
val = (yield i)
```

Recomiendo que **siempre** ponga paréntesis alrededor de una expresión `yield` cuando esté haciendo algo con el valor retornado, como en el ejemplo de arriba. Los paréntesis no siempre son necesarios, pero siempre es más fácil agregarlos en lugar de tener que recordar cuándo son necesarios.

(PEP 342 explica las reglas exactas, que son que una expresión `yield` siempre debe estar entre paréntesis excepto cuando se encuentra en la expresión de primer nivel en el lado derecho de una asignación. Esto significa que puede escribir `val = yield i` pero tiene que usar paréntesis cuando hay una operación, como en `val = (yield i) + 12`.)

Los valores son enviados a un generador llamando a su método `send(value)`. Este método reanuda el código del generador y la expresión `yield` retorna el valor especificado. Si se llama al método regular `__next__()`, `yield` retorna `None`.

Este es un simple contador que incrementa por 1 y permite cambiar los valores del contador interno.

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

Y este es un ejemplo del cambio del contador:

```
>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
>>> next(it)
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    it.next()
StopIteration
```

Como `yield` a menudo retornará `None`, siempre debería comprobar este caso. No use simplemente su valor en expresiones a menos que esté seguro que el método `send()` será el único método utilizado para reanudar su función generadora.

Además del método `send()`, hay otros dos métodos para generadores:

- `throw(value)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator's execution is paused.
- `close()` lanza una excepción `GeneratorExit` dentro del generador para terminar la iteración. Al recibir esta excepción, el código del generador debe lanzar `GeneratorExit` o `StopIteration`; capturar la excepción y hacer cualquier otra cosa es ilegal y disparará un `RuntimeError`. `close()` también se llamará por el recolector de basura de Python cuando se recolecte el generador.

Si necesita ejecutar un código de limpieza cuando ocurre un `GeneratorExit`, sugiero usar un `try: ... finally:` en lugar de capturar `GeneratorExit`.

El efecto acumulativo de estos cambios es convertir a los generadores de productores de información unidireccionales en productores y consumidores.

Los generadores también se convierten en **corrutinas**, una forma más generalizada de subrutinas. Las subrutinas inician en un punto y salen en otro punto (el inicio de la función, y la sentencia `return`), pero las corrutinas pueden iniciar, salir, y reanudarse en muchos puntos diferentes (las sentencias `yield`).

5 Funciones incorporadas

Veamos con más detalle las funciones incorporadas usadas a menudo con iteradores.

Dos de las funciones incorporadas de Python, `map()` y `filter()` duplican las características de las expresiones generadoras:

`map(f, iterA, iterB, ...)` retorna un iterador sobre la secuencia

```
f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]),
....
```

```
>>> def upper(s):
...     return s.upper()
```

```
>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

Por supuesto puede lograr el mismo efecto con una lista por comprensión.

`filter(predicate, iter)` retorna un iterador sobre todos los elementos de la secuencia que reúnen cierta condición, y de forma similar está duplicado por las listas por comprensión. Un **predicado** es una función que retorna un valor de verdad de alguna condición; para usarlo con `filter()`, el predicado debe recibir un solo valor.

```
>>> def is_even(x):
...     return (x % 2) == 0
```

```
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

Esto también se puede escribir como una lista por comprensión:

```
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

`enumerate(iter, start=0)` enumera los elementos en el iterable retornando 2-tuplas que contienen la enumeración (desde *start*) y cada elemento.

```
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print(item)
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` a menudo se usa cuando se recorre una lista y se registran los índices en los que se reúnen una cierta condición:

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

`sorted(iterable, key=None, reverse=False)` reúne todos los elementos de un iterable en una lista, ordena la lista, y retorna el resultado ordenado. Los argumentos *key* y *reverse* se pasan a través del método `sort()` de la lista construida.

```
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(Para un tratamiento más detallado del ordenamiento, ver [sortinghowto](#).)

Las funciones incorporadas `any(iter)` y `all(iter)` ven los valores de verdad de los contenidos de un iterable. `any()` retorna `True` si algún elemento en el iterable es un valor verdadero, y `all()` retorna `True` si todos los elementos son valores verdaderos:

```
>>> any([0, 1, 0])
True
>>> any([0, 0, 0])
False
>>> any([1, 1, 1])
True
>>> all([0, 1, 0])
False
>>> all([0, 0, 0])
False
```

(continúe en la próxima página)

```
>>> all([1, 1, 1])
True
```

`zip(iterA, iterB, ...)` toma un elemento de cada iterable y los retorna en una tupla:

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
('a', 1), ('b', 2), ('c', 3)
```

Esto no construye una lista en memoria y consume todos los iteradores de entrada antes de ejecutar; en lugar de eso, las tuplas se construyen y retornan solo si son requeridas. (El término técnico para este comportamiento es *evaluación perezosa*.)

Se pretende que el iterador se use con iterables que tengan todos la misma longitud. Si los iterables son de diferentes longitudes, el flujo resultante tendrá la misma longitud que el iterable más corto.

```
zip(['a', 'b'], (1, 2, 3)) =>
('a', 1), ('b', 2)
```

Sin embargo, debería evitar hacer esto, porque se puede tomar un elemento de los iteradores más largos y descartarlo. Esto significa que ya no puede seguir usando los iteradores porque corre el riesgo de saltarse un elemento descartado.

6 El módulo `itertools`

The `itertools` module contains a number of commonly used iterators as well as functions for combining several iterators. This section will introduce the module's contents by showing small examples.

Las funciones del módulo caen en unas pocas clases generales:

- Funciones que crean un nuevo iterador basado en un iterador existente.
- Funciones para tratar los elementos de un iterador como argumentos de función.
- Funciones para seleccionar porciones de la salida de un iterador.
- Una función para agrupar la salida de un iterador.

6.1 Crear nuevos iteradores

`itertools.count(start, step)` retorna un flujo infinito de valores equiespaciados. Opcionalmente puede suministrar el número de inicio, que por defecto es 0, y el intervalo entre números, que por defecto es 1:

```
itertools.count() =>
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
itertools.count(10, 5) =>
10, 15, 20, 25, 30, 35, 40, 45, 50, 55, ...
```

`itertools.cycle(iter)` guarda una copia de los contenidos de un iterable provisto y retorna un nuevo iterador que retorna sus elementos del primero al último. El nuevo iterador repetirá estos elementos infinitamente.

```
itertools.cycle([1, 2, 3, 4, 5]) =>
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

`itertools.repeat(elem, [n])` retorna el elemento provisto *n* veces, o retorna el elemento indefinidamente si no se provee *n*.

```

itertools.repeat('abc') =>
    abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
    abc, abc, abc, abc, abc

```

`itertools.chain(iterA, iterB, ...)` recibe un número arbitrario de iterables como entrada, y retorna todos los elementos del primer iterador, luego todos los elementos del segundo, y así sucesivamente, hasta que hayan sido agotados todos los iterables.

```

itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
    a, b, c, 1, 2, 3

```

`itertools.islice(iter, [start], stop, [step])` retorna un flujo que es un segmento del iterador. Con solo un argumento *stop*, retornará los primeros *stop* elementos. Si suministra un índice de inicio, obtendrá los elementos *stop-start*, y si suministra un valor para *step*, los elementos se saltarán en consecuencia. A diferencia de la segmentación de listas y cadenas de caracteres de Python, no puede usar valores negativos para *start*, *stop* o *step*.

```

itertools.islice(range(10), 8) =>
    0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
    2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
    2, 4, 6

```

`itertools.tee(iter, [n])` replica un iterador; retorna *n* iteradores independientes que retornarán los contenidos del iterador fuente. Si no suministra un valor para *n*, por defecto es 2. Replicar iteradores requiere guardar algunos de los contenidos del iterador fuente, esto puede consumir memoria significativa si el iterador es grande y uno de los nuevos iteradores se consume más que los otros.

```

itertools.tee(itertools.count()) =>
    iterA, iterB

where iterA ->
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

and iterB ->
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

```

6.2 Aplicar funciones a los elementos

El módulo `operator` contiene un conjunto de funciones correspondientes a los operadores de Python. Algunos ejemplos son `operator.add(a, b)` (suma dos valores), `operator.ne(a, b)` (lo mismo que `a != b`) y `operator.attrgetter('id')` (retorna un invocable que obtiene el atributo `.id`).

`itertools.starmap(func, iter)` asume que el iterable retornará un flujo de tuplas, y llama a *func* usando estas tuplas como los argumentos:

```

itertools.starmap(os.path.join,
    [('/bin', 'python'), ('/usr', 'bin', 'java'),
     ('/usr', 'bin', 'perl'), ('/usr', 'bin', 'ruby')])
=>
    /bin/python, /usr/bin/java, /usr/bin/perl, /usr/bin/ruby

```

6.3 Seleccionar elementos

Otro grupo de funciones elige un subconjunto de elementos de un iterador basado en un predicado dado.

`itertools.filterfalse(predicate, iter)` es el opuesto de `filter()`, retornando todos los elementos para los que el predicado retorna falso:

```
itertools.filterfalse(is_even, itertools.count()) =>
1, 3, 5, 7, 9, 11, 13, 15, ...
```

`itertools.takewhile(predicate, iter)` retorna los elementos mientras el predicado retorne verdadero. Una vez que el predicado retorna falso, el iterador indicará el final de sus resultados.

```
def less_than_10(x):
    return x < 10

itertools.takewhile(less_than_10, itertools.count()) =>
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
0
```

`itertools.dropwhile(predicate, iter)` descarta los elementos mientras el predicado retorne verdadero, y luego retorna el resto de los resultados del iterable.

```
itertools.dropwhile(less_than_10, itertools.count()) =>
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

`itertools.compress(data, selectors)` recibe dos iteradores y retorna solo aquellos elementos de *data* para los cuales el elemento correspondiente de *selectors* es verdadero, deteniéndose cuando alguno se termine:

```
itertools.compress([1, 2, 3, 4, 5], [True, True, False, False, True]) =>
1, 2, 5
```

6.4 Funciones combinatorias

La función `itertools.combinations(iterable, r)` retorna un iterador proporcionando todas las combinaciones de *r*-tuplas de los elementos contenidos en *iterable*.

```
itertools.combinations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 3), (2, 4), (2, 5),
(3, 4), (3, 5),
(4, 5)

itertools.combinations([1, 2, 3, 4, 5], 3) =>
(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5),
(2, 3, 4), (2, 3, 5), (2, 4, 5),
(3, 4, 5)
```

Los elementos dentro de cada tupla permanecen en el mismo orden en el que *iterable* los retornó. Por ejemplo, el número 1 siempre está antes que 2, 3, 4 o 5 en los ejemplos de arriba. Una función similar, `itertools.permutations(iterable, r=None)`, remueve esta restricción en el orden, retornando todas las disposiciones posibles de longitud *r*:

```
itertools.permutations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
```

(continúe en la próxima página)

(proviene de la página anterior)

```
(2, 1), (2, 3), (2, 4), (2, 5),
(3, 1), (3, 2), (3, 4), (3, 5),
(4, 1), (4, 2), (4, 3), (4, 5),
(5, 1), (5, 2), (5, 3), (5, 4)

itertools.permutations([1, 2, 3, 4, 5]) =>
(1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
...
(5, 4, 3, 2, 1)
```

Si no suministra un valor para *r* se usa la longitud del iterable, lo que significa que se permutan todos los elementos.

Note que estas funciones producen todas las combinaciones posibles por posición y no requieren que los contenidos de *iterable* sean únicos:

```
itertools.permutations('aba', 3) =>
('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')
```

La tupla idéntica ('a', 'a', 'b') aparece dos veces, pero las dos cadenas de caracteres “a” provienen de diferentes posiciones.

La función `itertools.combinations_with_replacement(iterable, r)` relaja una restricción diferente: los elementos se pueden repetir dentro de una misma tupla. Conceptualmente un elemento se elige para la primera posición de cada tupla y luego se reemplaza antes de que se elija el segundo elemento.

```
itertools.combinations_with_replacement([1, 2, 3, 4, 5], 2) =>
(1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
(2, 2), (2, 3), (2, 4), (2, 5),
(3, 3), (3, 4), (3, 5),
(4, 4), (4, 5),
(5, 5)
```

6.5 Agrupar elementos

La última función que trataré, `itertools.groupby(iter, key_func=None)`, es la más complicada. `key_func(elem)` es una función que calcula un valor clave para cada elemento retornado por el iterable. Si no suministra una función de clave, la clave será simplemente el elemento mismo.

`groupby()` reúne todos los elementos consecutivos del iterable subyacente que tienen el mismo valor clave, y retorna un flujo de 2-tuplas que contienen un valor clave y un iterador para los elementos con esa clave.

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'),
             ('Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'),
             ...
            ]

def get_state(city_state):
    return city_state[1]

itertools.groupby(city_list, get_state) =>
('AL', iterator-1),
('AK', iterator-2),
('AZ', iterator-3), ...

where
iterator-1 =>
('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
```

(continúe en la próxima página)

```
(('Anchorage', 'AK'), ('Nome', 'AK'))
iterator-3 =>
(('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'))
```

`groupby()` asume que los contenidos del iterable subyacente ya se ordenó basado en la clave. Note que los iteradores retornados también usan el iterable subyacente, así que tiene que consumir los resultados de iterador-1 antes de solicitar iterador-2 y su clave correspondiente.

7 El módulo `functools`

The `functools` module contains some higher-order functions. A **higher-order function** takes one or more functions as input and returns a new function. The most useful tool in this module is the `functools.partial()` function.

Para programas escritos en un estilo funcional, a veces querrá construir variantes de funciones existentes que tienen algunos de los parámetros predefinidos. Considere una función de Python `f(a, b, c)`; puede querer crear una nueva función `g(b, c)` que sea equivalente a `f(1, b, c)`; está completando un valor para uno de los parámetros de `f()`. Esto se llama «aplicación parcial de funciones».

El constructor para `partial()` recibe los argumentos (`function, arg1, arg2, ..., kwarg1=value1, kwarg2=value2`). El objeto resultante es invocable, por lo que puede invocar a `function` con los argumentos rellenados.

Aquí hay un ejemplo pequeño pero realista:

```
import functools

def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print('%s: %s' % (subsystem, message))
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

`functools.reduce(func, iter, [initial_value])` realiza acumulativamente una operación en todos los elementos del iterable y, por lo tanto, no se puede aplicar a infinitos iterables. `func` debe ser una función que recibe dos elementos y retorna un solo valor. `functools.reduce()` recibe los primeros dos elementos A y B retornados por el iterador y calcula `func(A, B)`. Luego pide el tercer elemento, C, calcula `func(func(A, B), C)`, combina este resultado con el cuarto elemento retornado, y continua hasta que se agote el iterable. Si el iterable no retorna ningún valor, se lanza una excepción `TypeError`. Si se suministra el valor inicial, se usa como punto inicial y `func(initial_value, A)` es el primer cálculo.

```
>>> import operator, functools
>>> functools.reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> functools.reduce(operator.concat, [])
Traceback (most recent call last):
...
TypeError: reduce() of empty sequence with no initial value
>>> functools.reduce(operator.mul, [1, 2, 3], 1)
6
>>> functools.reduce(operator.mul, [], 1)
1
```

Si usa `operator.add()` con `functools.reduce()`, sumará todos los elementos del iterable. Este caso es tan común que hay una función incorporada especial llamada `sum()` para calcularla:


```
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4], 0)
10
>>> sum([1, 2, 3, 4])
10
>>> sum([])
0
```

Sin embargo, para muchos usos de `functools.reduce()` puede ser mas claro simplemente escribir el ciclo `for`:

```
import functools
# Instead of:
product = functools.reduce(operator.mul, [1, 2, 3], 1)

# You can write:
product = 1
for i in [1, 2, 3]:
    product *= i
```

A related function is `itertools.accumulate(iterable, func=operator.add)`. It performs the same calculation, but instead of returning only the final result, `accumulate()` returns an iterator that also yields each partial result:

```
itertools.accumulate([1, 2, 3, 4, 5]) =>
1, 3, 6, 10, 15

itertools.accumulate([1, 2, 3, 4, 5], operator.mul) =>
1, 2, 6, 24, 120
```

7.1 El módulo operator

El módulo `operator` se mencionó anteriormente. Contiene un conjunto de funciones que corresponden a los operadores de Python. Estas funciones a menudo son útiles en código en estilo funcional porque le salvan de escribir funciones triviales que realizan una sola operación.

Algunas de las funciones en este módulo son:

- Operaciones matemáticas: `add()`, `sub()`, `mul()`, `floordiv()`, `abs()`, ...
- Operaciones lógicas: `not_()`, `truth()`.
- Operaciones bit a bit: `and_()`, `or_()`, `invert()`.
- Comparaciones: `eq()`, `ne()`, `lt()`, `le()`, `gt()` y `ge()`.
- Identidad de objeto: `is_()`, `is_not()`.

Consulte la documentación del módulo `operator` para una lista completa.

8 Funciones pequeñas y la expresión lambda

Cuando se escriben programas en estilo funcional, a menudo necesitará pequeñas funciones que actúen como predicados o que combinen elementos de alguna manera.

Si hay una función incorporada o un módulo de Python, no necesita definir una nueva función en absoluto:

```
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

Si la función que necesita no existe, necesita escribirla. Una forma de escribir funciones pequeñas es usar la expresión `lambda`. `lambda` recibe un número de parámetros y una expresión que combina estos parámetros, y crea una función anónima que retorna el valor de la expresión:

```
adder = lambda x, y: x+y

print_assign = lambda name, value: name + '=' + str(value)
```

Una alternativa es simplemente usar la sentencia `def` y definir una función en la forma usual:

```
def adder(x, y):
    return x + y

def print_assign(name, value):
    return name + '=' + str(value)
```

¿Qué alternativa es preferible? Esa es una pregunta de estilo; mi rumbo usual es evitar usar `lambda`.

Una razón para mi preferencia es que `lambda` es muy limitado en las funciones que puede definir. El resultado tiene que ser calculable como una sola expresión, que significa que no tiene comparaciones multivía `if... elif... else` o sentencias `try... except`. Si intenta hacer mucho en una sentencia `lambda`, terminará con una expresión demasiado complicada que es difícil de leer. Rápido, ¿qué hace el siguiente código?

```
import functools
total = functools.reduce(lambda a, b: (0, a[1] + b[1]), items)[1]
```

Puede averiguarlo, pero toma tiempo desenredar la expresión para averiguar que está pasando. Usar una corta sentencia `def` anidada hace las cosas un poco mejor:

```
import functools
def combine(a, b):
    return 0, a[1] + b[1]

total = functools.reduce(combine, items)[1]
```

Pero lo mejor de todo sería si simplemente hubiese usado un ciclo `for`:

```
total = 0
for a, b in items:
    total += b
```

O la función incorporada `sum()` y una expresión generadora:

```
total = sum(b for a, b in items)
```

Muchos usos de `functools.reduce()` son más claros cuando se escriben como ciclos `for`.

Fredrik Lundh una vez sugirió el siguiente conjunto de reglas para refactorizar los usos de `lambda`:

1. Escribir una función `lambda`.
2. Escribir un comentario explicando qué demonios hace esa `lambda`.
3. Estudiar el comentario por un momento, y pensar en un nombre que capture la esencia del comentario.
4. Convertir la `lambda` a una sentencia `def`, usando ese nombre.
5. Remover el comentario.

Me gustan mucho estas reglas, pero es libre de disentir acerca de si este estilo libre de `lambda` es mejor.

9 Historia de revisiones y reconocimientos

Al autor le gustaría agradecer a las siguientes personas por ofrecer sugerencias, correcciones y asistencia con varios borradores de este artículo: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Versión 0.1: publicada el 30 de junio de 2006.

Versión 0.11: publicada el 1 de julio de 2006. Errores tipográficos arreglados.

Versión 0.2: publicada el 10 de julio de 2006. Secciones `genexp` y `listcomp` unidas en una sola. Errores tipográficos arreglados.

Versión 0.21: Agregadas más referencias sugeridas en la lista de correos tutor.

Versión 0.30: Agrega una sección sobre el módulo `functional` escrito por Collin Winter; agrega una sección corta sobre el módulo `operator`; y unas pocas otras ediciones.

10 Referencias

10.1 Generales

Structure and Interpretation of Computer Programs, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. The book can be found at <https://mitpress.mit.edu/sicp>. In this classic textbook of computer science, chapters 2 and 3 discuss the use of sequences and streams to organize the data flow inside a program. The book uses Scheme for its examples, but many of the design approaches described in these chapters are applicable to functional-style Python code.

<https://www.defmacro.org/ramblings/fp.html>: A general introduction to functional programming that uses Java examples and has a lengthy historical introduction.

https://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional: Entrada general de Wikipedia que describe la programación funcional.

<https://es.wikipedia.org/wiki/Corrutina>: Entrada para corrutinas.

<https://es.wikipedia.org/wiki/Currificaci%C3%B3n>: Entrada para el concepto de currificación.

10.2 Específicas de Python

<https://gnosis.cx/TPiP/>: The first chapter of David Mertz's book *Text Processing in Python* discusses functional programming for text processing, in the section titled «Utilizing Higher-Order Functions in Text Processing».

Mertz también escribió una serie de artículos de 3 partes sobre programación funcional para el sitio DeveloperWorks de IBM; ver [parte 1](#), [parte 2](#), y [parte 3](#),

10.3 Documentación de Python

Documentación del módulo `itertools`.

Documentación del módulo `functools`.

Documentación del módulo `operator`.

PEP 289: «Expresiones generadoras»

PEP 342: «Corrutinas a través de generadores mejorados» describe las características del nuevo generador en Python 2.5.

Índice

P

Python Enhancement Proposals

PEP 289, [19](#)

PEP 342, [9](#), [19](#)