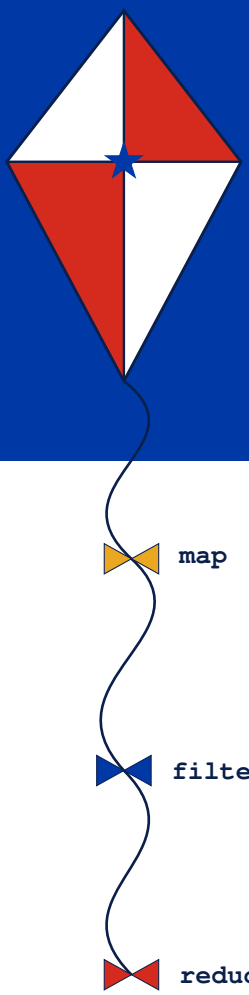
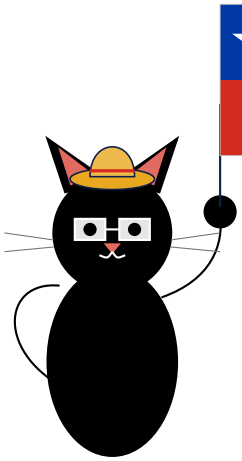
IIC2233 • PROGRAMACIÓN AVANZADA

PROGRAMACIÓN FUNCIONAL

Funciones puras • generadores • `map` • `filter` • `reduce` • `lambda`





Felipín se fue de fonda. Armó la ramada con una lista *global* de empanadas que modifica desde tres funciones distintas, mezcla `print` con cálculos, y para saber cuántas quedan … vuelve a recorrer todo con un `for` a mano. Resultado: nadie sabe cuántas empanadas quedan, ni por qué.

No seas como Felipín. En este resumen aprenderás a pensar en **flujos de transformación de datos**: funciones **puras**, generadores que entregan de a uno, y las herramientas `map`, `filter` y `reduce` para armar programas predecibles, componibles y eficientes.





Resumen de Contenidos

IIC2233 — Programación Avanzada • Edición Dieciocho

Resumen elaborado por el equipo de ayudantía, en base al material oficial del curso

Tabla de Contenidos

1 Introducción **3**

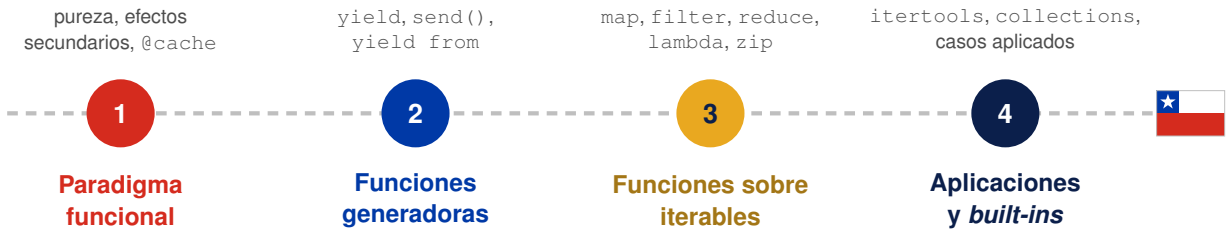
2 Paradigma funcional **3**
- 2.1 Paradigmas de programación . . . 3
- 2.2 ¿Qué es la programación funcional? . . . 4
- 2.3 Funciones puras . . . 5
- 2.4 Consecuencias de la programación funcional . . . 8
- 2.5 Conclusión . . . 10

3 Funciones generadoras **10**
- 3.1 `yield`: la función que se pausa . . . 10
- 3.2 Ejemplos . . . 12
- 3.3 Otra forma de hacer iterable una estructura propia . . . 14
- 3.4 Enviar mensajes al generador: `send()` . . . 15
- 3.5 `yield from`: generadores a partir de un iterable . . . 16
- 3.6 ¿Qué hace que una función sea generadora? . . . 17

4 Funciones que trabajan con iterables **18**
- 4.1 Funciones que necesitan de una función . . . 18
- 4.2 Funciones que no necesitan de una función . . . 26

5 Aplicaciones **29**
- 5.1 Funciones *built-in* de Python . . . 29
- 5.2 Librerías *built-in* . . . 31
- 5.3 Casos aplicados . . . 34

6 Hoja de referencia **38**



1 Introducción

Este resumen reúne el contenido de la semana de **Programación Funcional** de IIC2233. A primera vista puede parecer un conjunto de herramientas sueltas (`yield`, `map`, `lambda`, `itertools`...), pero todas responden a una misma forma de pensar: **construir programas como flujos de transformación de datos**, donde cada paso es una función que recibe un dato, lo transforma y lo entrega al siguiente, sin tocar nada más.

■ Los cuatro bloques de este resumen

- **Paradigma funcional** (Sección 2): qué es un paradigma de programación, qué significa que una función sea **pura**, qué son los *efectos secundarios* y qué consecuencias tiene programar así (por ejemplo, poder usar `@cache`).
- **Funciones generadoras** (Sección 3): funciones que **entregan valores de a uno** con `yield`, sin guardar todo en memoria; cómo se comunican con quien las llama (`send`) y cómo se componen (`yield from`).
- **Funciones que trabajan con iterables** (Sección 4): `map`, `filter`, `reduce`, las funciones `lambda`, y las funciones `enumerate` y `zip`.
- **Aplicaciones** (Sección 5): funciones *built-in* útiles sobre iterables, las librerías `itertools` y `collections`, y casos completos que combinan todo lo anterior (consultas anidadas, agrupación de datos, eficiencia en memoria).

★ Cómo leer este resumen

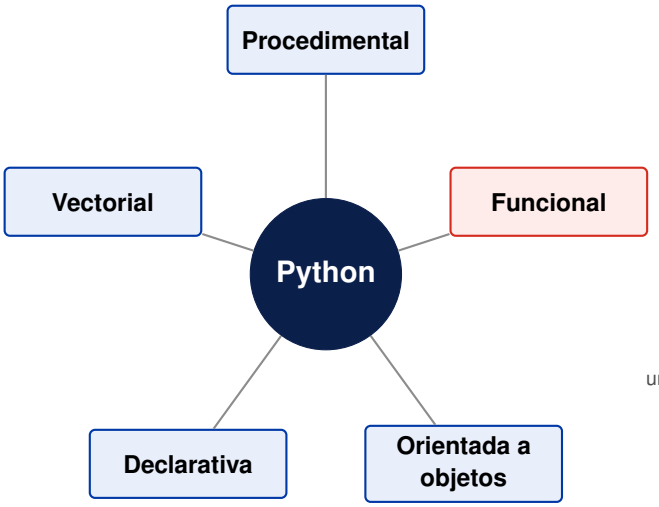
Cada bloque de código va acompañado de una explicación del tipo *"¿Qué hace este código?"*, y cuando el resultado es determinista se muestra la **salida** en un recuadro verde. Los recuadros **rojos** marcan errores típicos y trampas; los **dorados** son definiciones importantes; y las **analogías de fonda** son pequeñas metáforas para fijar las ideas. Si un ejemplo usa `lambda` antes de que se explique (Sección 4.1.4), no te preocupes: es solo una función corta escrita en una línea.

2 Paradigma funcional

2.1 Paradigmas de programación

Un **paradigma de programación** es un enfoque o estrategia para enfrentar un problema: nos dice *cómo* estructurar la solución. A inicios del curso vimos que existen varios, y que cada uno pone el énfasis en cosas distintas:

Paradigma	Idea central	Ejemplo
Procedimental	La solución es un programa **lineal**: una lista de instrucciones que indican qué hacer con la entrada en cada paso.	Lo que programamos en *Introducción a la Programación*.
Vectorial	Se usa en programas matemáticos con **paralelismo implícito**: se programa secuencialmente y el compilador distribuye el trabajo donde es posible.	Cálculos matemáticos sobre vectores y matrices.
Declarativa	El usuario **declara** el problema y el computador decide la mejor manera de resolverlo.	Consultas en SQL; problemas de optimización con restricciones y función objetivo.
Orientada a Objetos	Se modelan las funcionalidades mediante **interacciones entre objetos**, usando sus atributos y comportamientos.	El primer capítulo del curso.
Funcional	Es programación procedimental **de alto nivel**: la solución es un conjunto de funciones que reciben entradas y generan salidas. **No tienen estado**.	Este resumen.



Python es **multiparadigma**: una misma solución puede mezclar enfoques distintos.

■ Python es multiparadigma

Las soluciones en Python pueden escribirse de forma **procedimental**, **orientada a objetos** o **funcional**, e incluso mezclar los tres enfoques dentro de un mismo programa. Programación funcional no es "otro lenguaje": es **otra manera de usar el mismo Python**, con ciertas reglas de disciplina que a cambio nos dan código más predecible.

2.2 ¿Qué es la programación funcional?

En programación funcional, una **función** significa lo mismo que en matemáticas: **dado el mismo valor de entrada, se genera siempre el mismo (y único) valor de salida**. Por lo tanto, el valor de retorno depende **solamente** de los parámetros de entrada.

$$f(x) = x + 1 \qquad \Longrightarrow \qquad f(3) = 4 \text{ hoy, mañana y siempre.}$$

Bajo este paradigma **todo se ve como el *output* de una función**. Como el *output* solo depende del *input*, siempre podemos saber el valor de una variable que guarda el resultado de una función,

y esa variable **no cambiará** a menos que le asignemos el resultado de otra función. Esto da mucha claridad al código: tenemos la seguridad de que al ejecutar una función **no se modifica nada fuera de su ámbito** (*scope*).

★ Analogía de fonda

Piensa en una **máquina de sopaipillas**: si le echas la misma masa, siempre salen las mismas sopaipillas, y la máquina no se mete a la caja de la fonda, no cambia el menú ni le baja el volumen a la radio. Una función pura es esa máquina; una función impura es una que, además de hacer sopaipillas, anda tocando cosas que no le corresponden.

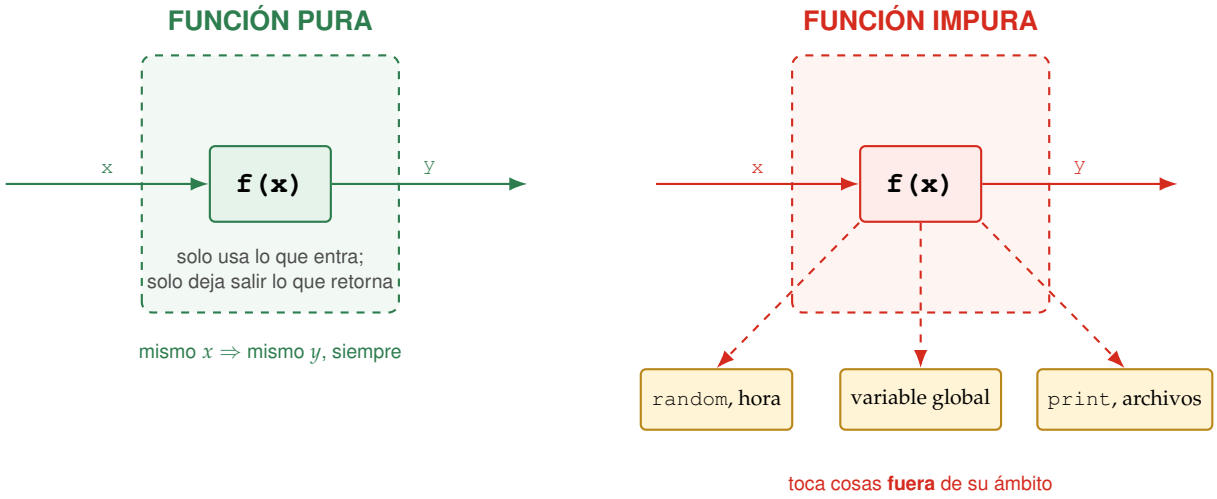
2.3 Funciones puras

■ Función pura

Una función es **pura** cuando cumple **las dos** condiciones siguientes:

1. **Mismo input ⇒ mismo output**: su valor de retorno depende únicamente de sus parámetros.
2. **No tiene efectos secundarios** (*side effects*): no afecta a nada que no dependa únicamente de la función.

La "pureza" mide el **grado de dependencia** de la función hacia el sistema o el programa; es decir, del contexto en el que se ejecuta. Un **efecto secundario** es cualquier cosa que la función hace o lee *más allá* de recibir parámetros y retornar un valor: modificar una variable global, mutar un argumento, imprimir en pantalla, leer o escribir archivos, consumir números aleatorios, etc.



2.3.1 Ejemplo 1: la lista global

```python
mi_lista = [1, 2, 3]

# Funcion IMPURA
def agregar_elemento(numero: int) -> None:
    mi_lista.append(numero)

for i in range(1, 6):
    agregar_elemento(4)
    print(f'Ejecucion {i}: {mi_lista}')
```

Salida

```
Ejecucion 1: [1, 2, 3, 4]
Ejecucion 2: [1, 2, 3, 4, 4]
Ejecucion 3: [1, 2, 3, 4, 4, 4]
Ejecucion 4: [1, 2, 3, 4, 4, 4, 4]
Ejecucion 5: [1, 2, 3, 4, 4, 4, 4, 4]
```

¿Qué hace este código? `agregar_elemento` no recibe la lista: la **busca afuera**, en la variable global `mi_lista`, y la modifica. Por eso es **impura**. Una forma sencilla de comprobarlo es llamarla varias veces **con el mismo argumento**: el efecto va cambiando en cada ejecución porque depende de un estado externo.

Para volverla pura, el acceso a la lista debe ocurrir **por medio del input**, y además no debemos mutar la lista original, sino trabajar sobre una copia:

```python
from copy import deepcopy  # la nueva lista no debe depender de la original

mi_lista = [1, 2, 3]

# Funcion PURA
def agregar_elemento(lista: list, numero: int) -> list:
    copia_lista = deepcopy(lista)
    copia_lista.append(numero)
    return copia_lista

for i in range(1, 6):
    nueva_lista = agregar_elemento(mi_lista, 4)
    print(f'Ejecucion {i}: {mi_lista} {nueva_lista}')
```

Salida

```
Ejecucion 1: [1, 2, 3] [1, 2, 3, 4]
Ejecucion 2: [1, 2, 3] [1, 2, 3, 4]
Ejecucion 3: [1, 2, 3] [1, 2, 3, 4]
Ejecucion 4: [1, 2, 3] [1, 2, 3, 4]
Ejecucion 5: [1, 2, 3] [1, 2, 3, 4]
```

¿Qué hace este código? Ahora la función depende **únicamente** de sus parámetros, y su retorno es siempre el mismo para los mismos argumentos. Además, al modificar solo una **copia** creada dentro de su propio *scope*, la lista original queda intacta: no hay efecto secundario. Por eso, en cada ejecución, `mi_lista` sigue siendo `[1, 2, 3]`.

! Mutar un argumento también es un efecto secundario

No basta con que la lista llegue por parámetro. Si la función hiciera `lista.append(numero)` directamente sobre el parámetro, estaría **modificando un objeto que pertenece a quien la llamó**, y eso es un efecto secundario aunque no use variables globales. Por eso la versión pura hace primero una `deepcopy` y solo entonces modifica la copia.

2.3.2 Ejemplo 2: un catálogo de casos

Para decidir si una función es pura, siempre nos hacemos las **dos preguntas**: (1) *si mantengo el input, ¿siempre obtengo el mismo output?* y (2) *¿tiene efectos secundarios?* Revisemos varios casos:

```python
def aumentar(x: int) -> int:
    return x + 1

items = ['a', 'b', 'c']
def agregar_item(item: str) -> str:
    items.append(item)
    return item

from random import random
def fraccionar_random(x: float) -> float:
    return x * random()

from random import random, seed
def fraccionar_random2(x: float) -> float:
    seed(x)
    return x * random()

from random import Random
def fraccionar_random3(x: float) -> float:
    NuevoRandom = Random(x)
    return x * NuevoRandom.random()

def aumentar_con_print(x: int) -> int:
    print('Voy a retornar', x, 'aumentado en uno')
    return x + 1
```

Función	Mismo output	Sin efectos	Veredicto y motivo
`aumentar`	✓	✓	**Pura.** Solo usa `x`.
`agregar_item`	✓	✗	**Impura.** Retorna lo mismo que recibe, pero cambia el contenido de la lista `items`.
`fraccionar_random`	✗	✗	**Impura.** Depende de un valor aleatorio, y llamar a `random` altera el siguiente valor que se obtendrá.
`fraccionar_random2`	✓	✗	**Impura.** Con `seed(x)` la salida sí es repetible, pero `seed` modifica el generador **global**, afectando futuras llamadas a `random()` fuera de la función.
`fraccionar_random3`	✓	✓	**Pura.** `Random(x)` crea un generador **local**, dependiente solo de la *seed*; nada global cambia.
`aumentar_con_print`	✓	✗	**Impura.** `print` afecta lo que muestra el programa: es un efecto secundario.

★ Los efectos secundarios llegan más lejos de lo que parece

Una función pura **tampoco puede crear ni modificar archivos**, ya que eso altera elementos ajenos al programa. Regla mental: si al ejecutar la función algo cambia (o se puede observar) *fuera* de ella, aparte de su valor de retorno, no es pura.

2.4 Consecuencias de la programación funcional

Si escribimos funciones puras, ganamos varias propiedades importantes:

1. **Ya no se modifican elementos externos a una función.** Entonces, si una variable o estructura cambia inesperadamente, podemos **descartar** revisar las funciones puras: ellas no pudieron ser.
2. **El *output* solo depende del *input*.** Si el programa **recuerda** el resultado para un input dado, no necesita volver a llamar a la función (*memoización*).
3. **Al no haber efectos secundarios, las iteraciones** (`for`, `while`) **son más fáciles de modelar**, y el código se vuelve más amigable de razonar.

2.4.1 Memoización con `@cache`

Veamos la segunda consecuencia con Fibonacci. Esta versión imprime un mensaje cada vez que se **ejecuta** para un número:

```python
def fibonacci(n: int) -> int:
    print('buscando el numero', n)
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(4)
```

Salida

```
buscando el numero 4
buscando el numero 3
buscando el numero 2
buscando el numero 1
buscando el numero 0
buscando el numero 1
buscando el numero 2
buscando el numero 1
buscando el numero 0
```

Hay muchas llamadas **repetidas** (por ejemplo, `fibonacci(2)` se calcula dos veces), y si ignoramos los `print` (que solo están para *mostrar* cómo corre el código), lo único que queremos es el output a partir del input. Podemos decirle a Python que **no vuelva a ejecutar** la función con el decorador `@cache` de `functools`, y que asuma que el output se mantiene dado el mismo input:

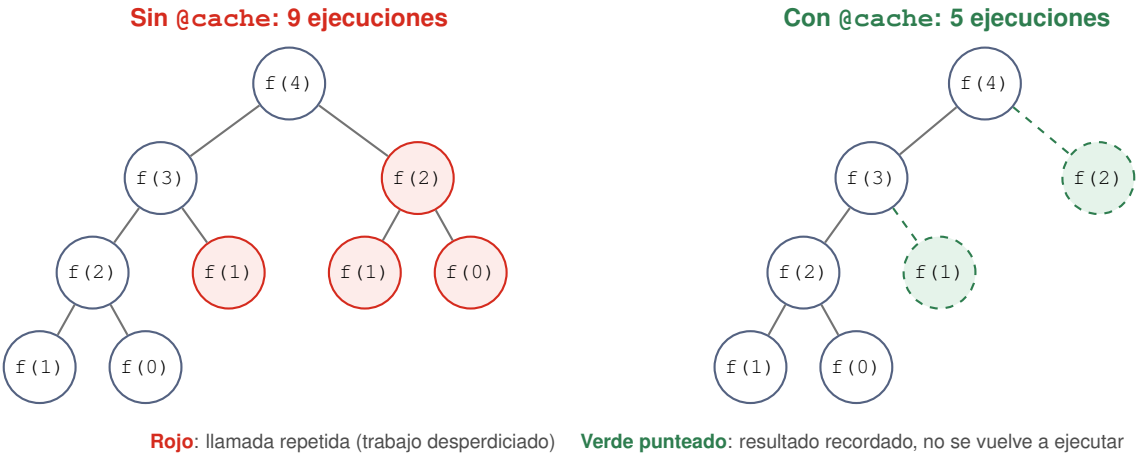
```python
from functools import cache

@cache
def fibonacci_cached(n: int) -> int:
    print('buscando el numero', n)
    if n < 2:
        return n
    return fibonacci_cached(n - 1) + fibonacci_cached(n - 2)

fibonacci_cached(4)
```

Salida

```
buscando el numero 4
buscando el numero 3
buscando el numero 2
buscando el numero 1
buscando el numero 0
```



Rojo: llamada repetida (trabajo desperdiciado) **Verde punteado**: resultado recordado, no se vuelve a ejecutar

¿Qué hace este código? Con `@cache`, la primera vez que se llama `fibonacci_cached(n)` para un valor de `n`, se ejecuta y se **guarda** el resultado; las siguientes veces que aparece ese mismo `n`, Python devuelve el resultado guardado **sin ejecutar** el cuerpo (por eso ya no vemos mensajes repetidos).

! `@cache` solo funciona bien con funciones puras

El decorador `@cache` asume que **el mismo input siempre da el mismo output**, y que no hace falta ejecutar la función de nuevo. Si la función tuviera efectos secundarios (como el `print` de arriba), esos efectos **dejarían de ocurrir** en las llamadas recordadas. Ese es justamente el motivo por el que vemos menos mensajes: **hacer `print` vuelve impura a la función**, y `@cache` la trata como si no lo fuera. Además, los argumentos de la función deben ser **hasheables** (por ejemplo, `int`, `str` o tuplas; no `list`).

Ahora que quitamos el `print`, la función es pura y podemos aprovechar el decorador con valores mucho más grandes:

```python
@cache
def fibonacci_cached(n: int) -> int:
    if n < 2:
        return n
    return fibonacci_cached(n - 1) + fibonacci_cached(n - 2)

fibonacci_cached(300)    # respuesta inmediata
```

Salida

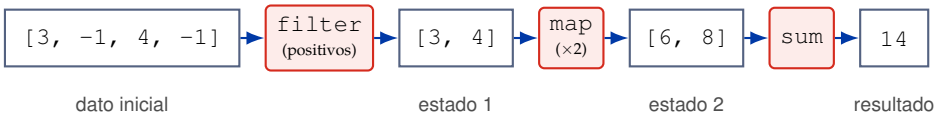
```
222232244629420445529739893461909967206666939096499764990979600
```

Sin memoización, esta misma llamada **nunca terminaría en la práctica**: la cantidad de llamadas crece de forma exponencial. Con `@cache`, cada valor de `n` entre `0` y `300` se calcula **una única vez**.

2.4.2 Pensar en flujos de datos

Que nuestras funciones sean puras también cambia **cómo planteamos** un programa:

1. Ya no pensamos en **almacenar** variables, sino en **modificar** (transformar) datos.
2. El razonamiento se basa en **flujos de transformación de datos** (*data flow*).
3. Cada función recibe un **estado particular** del dato; a partir de él lo transforma y **lo retorna** (sin alterar el original).



2.5 Conclusión

Implementar bien una solución funcional (funciones puras, sin efectos secundarios, pensando en flujos de datos) es la base de todo lo que viene: generadores, `map`, `filter`, `reduce` y las consultas anidadas son **herramientas para armar esos flujos**.

★ Una aclaración sobre el material del curso

Para facilitar el aprendizaje, los *notebooks* del curso muestran `print` **dentro** de funciones que, en rigor, deberían ser puras. Es solo con fines didácticos, para que veas qué ocurre paso a paso. En un programa funcional "de verdad", los `print` irían **fuera** de las funciones de transformación.

3 Funciones generadoras

★ Repaso rápido

Un **generador** es un caso especial de **iterador**: permite recorrer una secuencia de datos **sin almacenarlos** en una estructura, evitando un uso innecesario de memoria. Produce cada valor **cuando se le pide**, y una vez que terminamos de iterarlo, **desaparece** (se agota). Ya conocemos una forma de crearlos: la *expresión generadora* `(2 * i for i in range(10))`. En esta sección veremos la otra: las **funciones generadoras**.

3.1 `yield`: la función que se pausa

Las funciones de Python también pueden funcionar como generadores, mediante la sentencia `yield`. Es un análogo de `return` con una diferencia crucial: `yield` **entrega el valor indicado, pero deja la función en pausa**, de modo que en la próxima llamada la ejecución **parte desde donde se dejó**. Una función que entrega un valor con `yield` cede el control solo de forma **temporal**, asumiendo que pronto se le pedirá otro valor.

	return	yield
Qué hace	Entrega un valor y **termina** la función.	Entrega un valor y **pausa** la función.
Variables locales	Se pierden al terminar.	Se **conservan** mientras el generador exista.
Próxima llamada	Empieza de nuevo desde la primera línea.	**Continúa** justo después del `yield`.
Qué retorna la función	El valor mismo.	Un objeto **generador** (sin ejecutar nada aún).

Creemos nuestra primera función generadora, una cuenta regresiva:

```python
from typing import Generator

def conteo_decreciente(n: int) -> Generator:
    print(f"Contando en forma decreciente desde {n}")
    while n > 0:
        yield n
        n -= 1
```

Lo primero que llama la atención es que, cuando **llamamos** a la función generadora, **no se ejecuta nada**: ni siquiera se imprime el mensaje inicial.

```python
x = conteo_decreciente(4)      # no imprime nada
print(type(x))
```

Salida

```
<class 'generator'>
```

¿Qué hace este código? Invocar una función generadora **no ejecuta su cuerpo**: solo **retorna un generador**. Recién cuando le pedimos valores (con un `for` o con `next`), el cuerpo comienza a correr, hasta llegar al primer `yield`. Aquí, `x` es un generador de los números desde el `4` hasta el `1`.

Como los generadores implementan `__iter__` retornando `self`, podemos usarlos directamente en un `for`; también podemos avanzarlos manualmente con `next`:

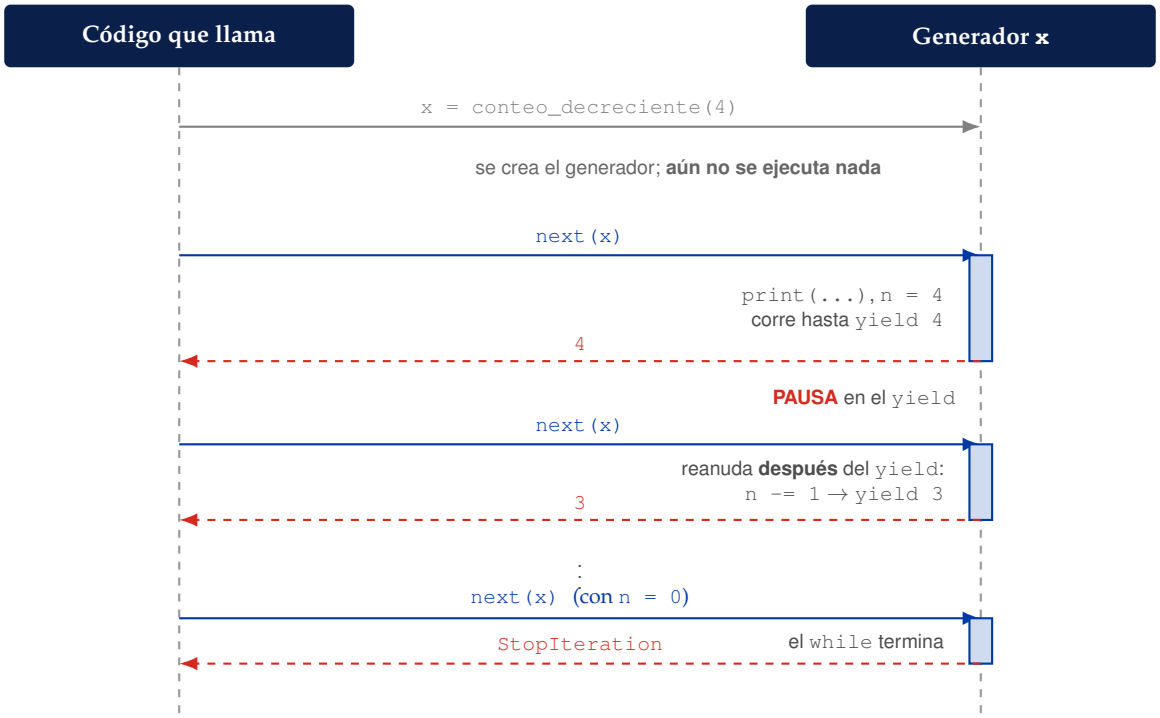
```python
for number in x:
    print(number)

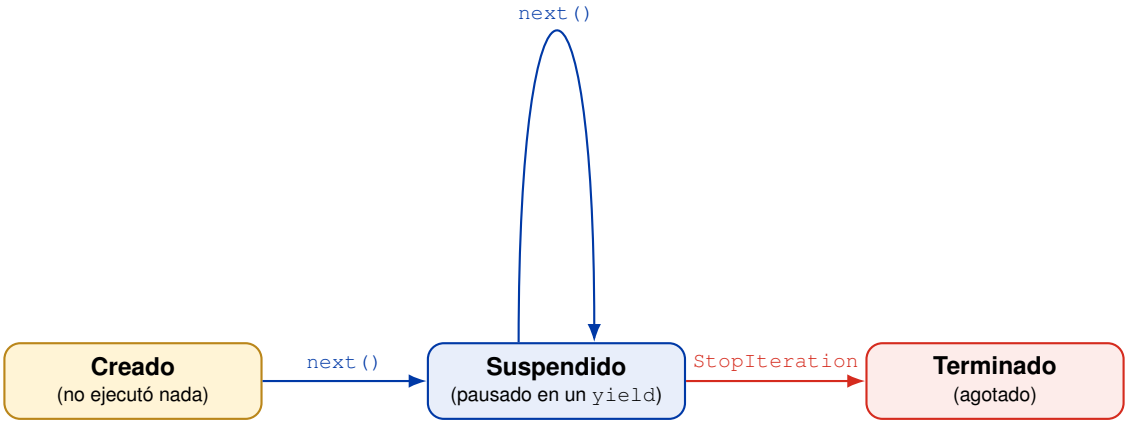
# --- otra forma, avanzando de a uno ---
x = conteo_decreciente(5)
print(next(x))
```

Salida

```
Contando en forma decreciente desde 4
4
3
2
1
Contando en forma decreciente desde 5
```

```
5
```





★ Analogía de fonda

Un generador es **la tía de las empanadas** que las va sacando del horno *una por una, a medida que las piden*. Una lista, en cambio, sería tener las mil empanadas ya servidas sobre el mesón, ocupando espacio aunque nadie las haya pedido todavía. Y cuando la tía saca la última, se acabó: no puedes volver a pedir la primera con el mismo horno.

3.2 Ejemplos

3.2.1 Un generador que nunca se agota: Fibonacci

```python
def fibonacci() -> Generator:
    a, b = 0, 1
    while True:              # este generador nunca "se agota"
```

```
        yield a
        a, b = b, a + b

generador_fibonacci = fibonacci()

# Imprimimos los primeros 5 elementos
for i in range(5):
    print(next(generador_fibonacci))
```

Salida

```
0
1
1
2
3
```

¿Qué hace este código? El `while True` parecería un ciclo infinito, pero nunca se ejecuta "de golpe": cada vez que se pide un valor, corre solo hasta el siguiente `yield` y se detiene. Así podemos representar una secuencia **infinita** ocupando memoria constante, y nosotros decidimos cuántos elementos consumir (aquí, cinco).

3.2.2 Leer un archivo sin cargarlo completo en memoria

```
from os import path
from typing import Generator

def cargar_archivo(ruta: str) -> Generator:
    """Lee un archivo sin cargar toda su informacion en memoria."""
    with open(ruta, encoding='utf-8') as file:
        for line in file:
            yield line

lenguas_extintas = cargar_archivo(path.join('data', 'lenguas_extintas.csv')
    )

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

¿Qué hace este código? La función abre el archivo y va **entregando una línea a la vez**. Fíjate que `for line in file` ya es, en sí mismo, una iteración perezosa: el objeto archivo entrega sus líneas de a una. En cambio, `file.readlines()` retorna una **lista con todas las líneas**, es decir, carga **todo el archivo en la memoria** del computador. Para archivos grandes, esa diferencia importa muchísimo.

3.2.3 Un generador sobre otra colección: máximo acumulativo

```
def maximo_acumulativo(valores: list) -> Generator:
    """Retorna el maximo visto hasta ahora en una coleccion de valores."""
    max_ = float('-inf')
    for valor in valores:
        max_ = max(valor, max_)
        yield max_
```

```
lista = [1, 10, 14, 7, 9, 12, 19, 33]
for i in maximo_acumulativo(lista):
    print(i)
```

Salida

```
1
10
14
14
14
14
19
33
```

¿Qué hace este código? La función recorre la lista y, tras cada elemento, entrega **el mayor valor visto hasta ese momento**. Como el `yield` está dentro del `for`, el generador va **conservando** entre pausa y pausa el valor de `max_`: eso es exactamente lo que una función normal con `return` no puede hacer.

3.3 Otra forma de hacer iterable una estructura propia

Si tenemos una estructura de datos propia, podemos escribir una **función generadora dentro de `__iter__`**, en vez de crear nuestra propia clase iteradora con `__next__`. Usemos el mismo ejemplo del resumen anterior (`IterableListaNumeros`):

```
class IterableListaNumeros:
    def __init__(self, objeto: list) -> None:
        self.objeto = objeto.copy()

    def __iter__(self) -> Generator:
        while self.objeto:
            yield self.objeto.pop(0)

datos = [1, 2, 3, 4, 5]
iterable = IterableListaNumeros(datos)

for x in iterable:
    print(x, end=" ")              # 1 2 3 4 5

iterador = iter(iterable)
print(type(iterador))              # <class 'generator'>
```

¿Qué hace este código? Como `__iter__` contiene un `yield`, es una función generadora: al llamarla (implícitamente, con el `for` o con `iter()`) Python retorna un **generador**, que cumple el rol de iterador. Nos ahorramos escribir la clase iteradora y el método `__next__`: Python construye todo eso por nosotros.

! Cuidado: este iterable se consume

En el código anterior la copia de la lista se hace **una sola vez, en `__init__`**, y `__iter__` la **vacía** con `pop(0)`. Por eso, si recorres `iterable` una segunda vez, **no entrega nada**: se comportó como iterador y no como iterable. Para que se pueda recorrer múltiples veces basta copiar dentro de `__iter__` (o directamente no mutar):

```
    def __iter__(self) -> Generator:
```

```
        yield from self.objeto          # o: for x in self.objeto: yield
    x
```

3.4 Enviar mensajes al generador: `send()`

Los generadores no solo *entregan* valores: también pueden **recibirlos**. El método `send()` envía un valor **hacia** el generador, y la expresión `yield` lo recibe. Si escribimos `v = yield valor`, la variable `v` guardará el valor enviado con `send()`.

Veamos una función generadora que entrega un contador que se incrementa según lo que le enviamos:

```
def funcion_generadora_send() -> Generator:
    contador = 0

    while True:
        valor_recibido = yield contador
        print("Hemos recibido {}".format(valor_recibido))

        if valor_recibido is None:    # consideraremos 0 si nos llega un
    None
            valor_recibido = 0

        print("Sumaremos {} a nuestro contador".format(valor_recibido))
        contador += valor_recibido
```

! No se puede enviar antes del primer `yield`

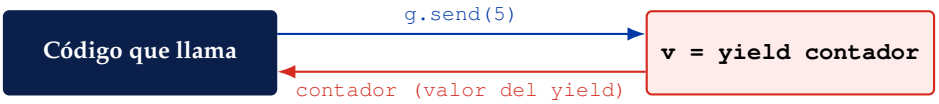
Antes de poder enviar valores, **hay que avanzar hasta el primer `yield`** con `next`. Si intentamos `send(5)` sobre un generador recién creado, Python levanta un **`TypeError`**: no hay ningún `yield` esperando recibir el valor.

```
generador_send = funcion_generadora_send()
try:
    generador_send.send(5)
except TypeError as error:
    print("TypeError:", error)
# TypeError: can't send non-None value to a just-started generator
```

Una vez hecho el primer `next`, ya podemos comunicarnos con el generador:

```
next(generador_send)            # -> 0      (avanza hasta el primer yield)
generador_send.send(5)          # -> 5
next(generador_send)            # -> 5      (equivale a send(None))
generador_send.send(10)         # -> 15
```

Llamada	Qué recibe el `yield`	contador	Retorna	Se imprime
`next(g)`	nada: recién arranca	0	0	—
`g.send(5)`	5	5	5	`Hemos recibido 5` `Sumaremos 5 ...`
`next(g)`	`None` → se usa 0	5	5	`Hemos recibido None` `Sumaremos 0 ...`
`g.send(10)`	10	15	15	`Hemos recibido 10` `Sumaremos 10 ...`



comunicación bidireccional: el generador emite *y* recibe

3.4.1 Un pequeño paréntesis: ¿cuándo se asigna `valor_recibido`?

Puede surgir una duda: si al hacer `next(generador_send)` ya se ejecutó la línea `valor_recibido = yield contador`, ¿cómo es que después se guarda el `5`? La clave está en cómo Python ejecuta esa línea: **el `yield` sí se ejecuta** (entrega el valor y pausa), pero **la asignación a `valor_recibido` queda pendiente** hasta que el generador se **reanuda**. Cuando llamamos a `send(5)`, el `5` se convierte en el **resultado de la expresión** `yield contador`, y recién entonces se asigna.

En otras palabras, el `yield` actúa como una **puerta de entrada** para los valores enviados: al pausar, el generador "espera" a que le envíen algo, que se asignará a la variable cuando la ejecución continúe. Y por eso, **hacer `next` es equivalente a hacer `send(None)`**.

3.5 `yield from`: generadores a partir de un iterable

Otra forma de crear una función generadora es con `yield from`, que permite **crear un generador a partir de otro iterable** sin necesidad de un ciclo (`for` o `while`) que lo recorra. Con esto escribimos código más **limpio, modular y fácil de mantener** cuando trabajamos con generadores compuestos o anidados.

```python
def generador_simple():
    yield from range(5)

for valor in generador_simple():
    print(valor)          # 0 1 2 3 4
```

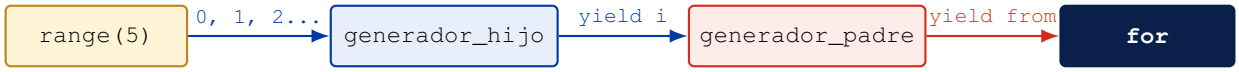
Como un generador también es un iterable, podemos **crear un generador a partir de otro generador**:

```python
def generador_hijo():
    for i in range(5):
        yield i

def generador_padre():
    yield from generador_hijo()

for valor in generador_padre():
    print(valor)          # 0 1 2 3 4
```

¿Qué hace este código? `yield from X` significa "entrega, uno por uno, todo lo que entregue `X`". Es un atajo de `for elemento in X: yield elemento`. En el segundo ejemplo, `generador_padre` **delega** su trabajo en `generador_hijo`, y los valores fluyen hacia quien recorre al padre.



3.6 ¿Qué hace que una función sea generadora?

Si una función se vuelve generadora por contener `yield`, surgen dos preguntas: ¿**toda** función que contenga `yield` es generadora?, ¿necesitamos que el `yield` **se ejecute**? Probemos con tres funciones:

```python
def funcion1():
    """yield dentro del scope principal."""
    yield 1

def funcion2():
    """yield dentro de un if falso, y ademas la funcion tiene un return."""
    if False:
        yield 1
    return 1

def funcion3():
    """Contiene un yield, pero nunca alcanza a ejecutarse."""
    return 1
    yield 1

print(type(funcion1()), type(funcion2()), type(funcion3()))
```

Salida

```
<class 'generator'> <class 'generator'> <class 'generator'>
```

Regla

Para que una función sea considerada **generadora** basta con que su cuerpo **contenga un `yield`, independiente de si este alcanza a ejecutarse o no**. Es una propiedad *sintáctica*: Python decide, al leer la definición de la función, que su llamada retornará un generador. (En ese caso, un `return valor` dentro de ella solo **termina** el generador; el valor viaja dentro de la excepción `StopIteration`, no como resultado de la llamada.)

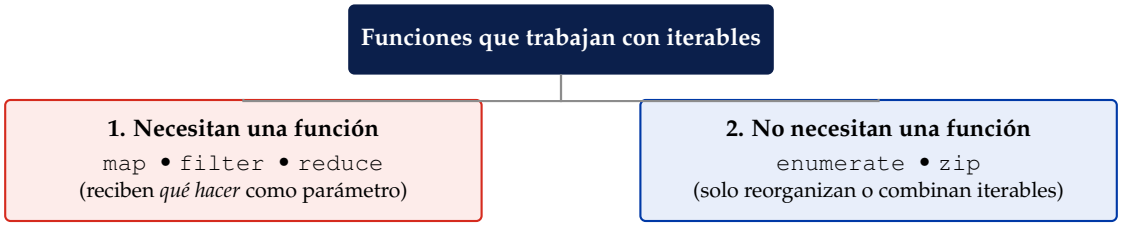
✓ Resumen express: funciones generadoras

- Una función con `yield` **retorna un generador** al llamarla; no ejecuta nada hasta que se le pide un valor.
- `yield` entrega un valor y **pausa** la función *conservando su estado local*.
- Se recorren con `for` o `next()`; al terminar levantan `StopIteration`, y no se pueden reiniciar.
- `send(v)` envía un valor al `yield` pausado (primero hay que hacer un `next`); `next` ≡ `send(None)`.
- `yield from X` delega en otro iterable o generador.

- Sirven para secuencias grandes o infinitas, y para leer archivos línea a línea.

4 Funciones que trabajan con iterables

Además de las funciones generadoras, Python trae funciones que **no necesariamente usan generadores**, pero que **aceptan cualquier iterable** (listas, tuplas, *sets*, generadores, archivos…) y suelen devolver **iteradores perezosos**. Las agrupamos en dos:



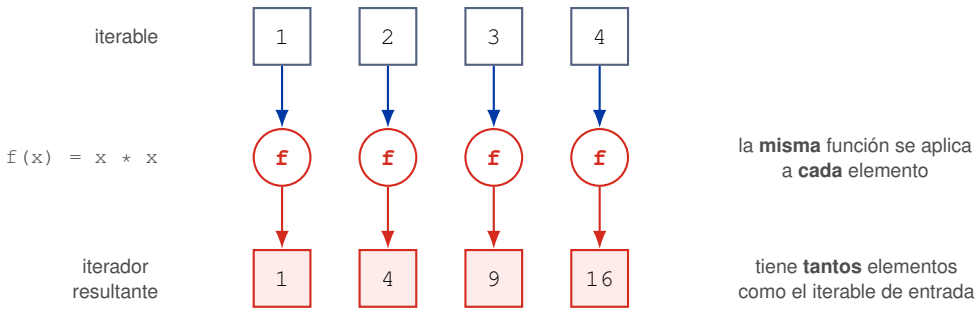
4.1 Funciones que necesitan de una función

4.1.1 `map`

`map` recibe una **función** y **al menos un iterable**, y retorna un **iterador** que resulta de **aplicar la función a cada elemento** del iterable, por separado. Es decir, `map(f, iterable)` es similar a la expresión generadora `(f(x) for x in iterable)`.

Sintaxis: `map(función, iterable1, ...)`

- `función`: una función de Python o, en general, **cualquier invocable**: funciones *built-in* o definidas por nosotros, clases, métodos de instancia y de clase, funciones `lambda`…
- `iterable`: cualquier iterable válido (lista, tupla, *set*, generador…).
- La **cantidad de iterables** debe coincidir con la **cantidad de parámetros** de la función: `map(f, it1, it2)` exige que `f` reciba dos parámetros, y equivale a `(f(x, y) for x, y in zip(it1, it2))`.



Ejemplo 1: un iterable. Queremos pasar a minúsculas una lista de *strings*:

```
def lower(string: str) -> str:
    return string.lower()

strings = ['Señores pasajeros', 'Disculpen', 'mi', 'IntencIÓN', 'no', 'Es',
    'MolEstar']
mapeo = map(lower, strings)

print(', '.join(mapeo))
```

Salida

```
señores pasajeros, disculpen, mi, intención, no, es, molestar
```

Ejemplo 2: varios iterables. A partir de dos o más secuencias de números, calculamos otro valor. Cada función recibe **un elemento de cada iterable**, en la misma posición:

```python
def polinomio_1(x: int, y: int) -> int:
    return x ** 2 + y ** 2

def polinomio_2(x: int, y: int, z: int) -> int:
    return x + y ** 2 + z ** 3

a = [1, 2, 3, 4]
b = [17, 12, 11, 10]
c = (-1, -4, 5, 9)          # los iterables pueden ser de tipos distintos

mapeo_1 = map(polinomio_1, a, b)
mapeo_2 = map(polinomio_2, a, b, c)

print(list(mapeo_1))
print(list(mapeo_2))
```

Salida

```
[290, 148, 130, 116]
[289, 82, 249, 833]
```

! `map` se detiene en el iterable más corto

La cantidad de elementos que procesa `map` corresponde al largo del **iterable más pequeño**:

```python
def suma(x: int, y: int) -> int:
    return x + y

a = range(1, 2233)          # 2232 elementos
b = [100, 101, 102]         # 3 elementos
list(map(suma, a, b))       # -> [101, 103, 105]
```

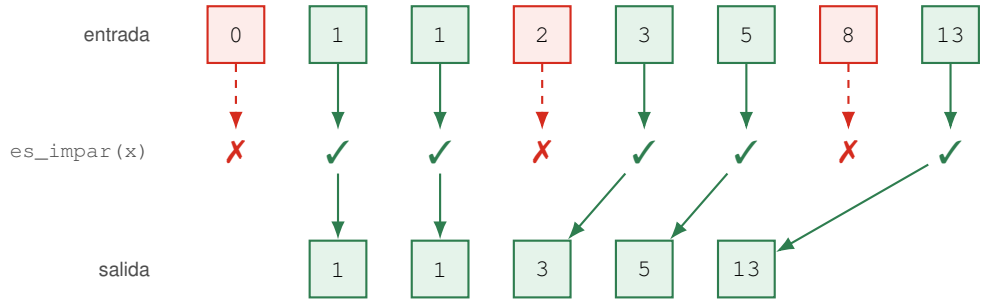
Además, `map` retorna un **iterador perezoso**: no calcula nada hasta que lo consumimos (con `list`, un `for`, `join`...), y **solo se puede consumir una vez**.

★ Analogía de fonda

`map` es como pasar **toda la fila de empanadas por la misma máquina de pintar**: a cada una se le hace exactamente lo mismo, y a la salida hay una empanada transformada por cada empanada que entró.

4.1.2 `filter`

`filter(f, iterable)` recibe una función que retorna `True` o `False` (función **booleana** o *predicado*) y un iterable. Retorna un generador que entrega **solo los elementos para los cuales `f` retorna `True`**. Es equivalente a `(x for x in iterable if f(x))`.




```python
def fibonacci(limite: int) -> Generator:
    a, b = 0, 1
    for _ in range(limite):
        yield a
        a, b = b, a + b

def es_impar(numero: int) -> bool:
    return numero % 2 != 0

def es_par(numero: int) -> bool:
    return numero % 2 == 0

filtrado_impares = filter(es_impar, fibonacci(10))
print(list(filtrado_impares))

filtrado_pares = filter(es_par, fibonacci(10))
print(list(filtrado_pares))
```

Salida

```
[1, 1, 3, 5, 13, 21]
[0, 2, 8, 34]
```

¿Qué hace este código? `fibonacci(10)` genera los diez primeros números de Fibonacci (`0, 1, 1, 2, 3, 5, 8, 13, 21, 34`). `filter` le entrega cada uno a la función booleana y **deja pasar únicamente** aquellos que dan `True`. Como `filter` acepta cualquier iterable, podemos pasarle directamente un generador.

Otro ejemplo, entregándole un *set*:

```python
def menor_que_10(numero: int) -> bool:
    return numero < 10

set_filtrado = filter(menor_que_10, {100, 1, 5, 9, 91, 1})
print(list(set_filtrado))       # contiene 1, 5 y 9 (el orden depende del
    set)
```

★ Analogía de fonda

`filter` es **el portero de la ramada**: cada persona (elemento) se presenta, el portero aplica una regla (la función booleana) y solo deja entrar a quienes la cumplen. Nadie sale transformado: los que pasan entran *tal como llegaron*.

4.1.3 `reduce`

Expliquemos la idea con un cálculo manual. Supongamos que tenemos una secuencia de números y queremos su suma, pero nos incomoda sumar más de dos números a la vez:

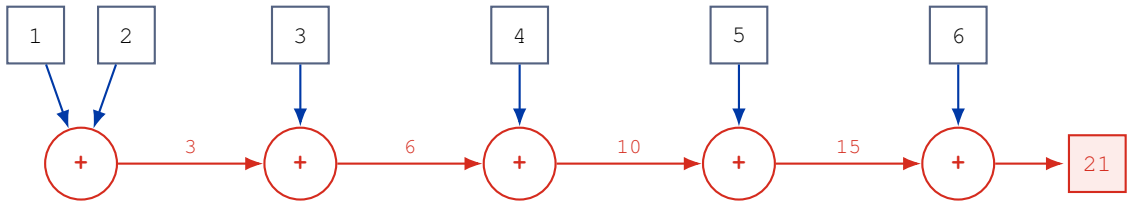
```
lista = [1, 2, 3, 4, 5, 6]
```

Lo natural sería sumar los dos primeros, guardar el resultado, sumarlo con el siguiente, y así sucesivamente: $1+2=3, 3+3=6, 6+4=10, 10+5=15, 15+6=21$. El resultado final es **21**.

Ahora supongamos que no queremos necesariamente sumar, sino aplicar una función cualquiera f *de a pares*:

$$f(1,2)=a,\ f(a,3)=b,\ f(b,4)=c,\ f(c,5)=d,\ f(d,6)=e \Rightarrow f\Big(f(f(f(f(1,2),3),4),5),6\Big)=e$$

Esa es exactamente la idea de `reduce`: aplicar sucesivamente una función $f(x,y)$, donde `x` es el **resultado acumulado** e `y` es el **siguiente elemento** de la secuencia. Así, el iterable se *reduce* a un único resultado.



$f(f(f(f(f(1,2),3),4),5),6)$ con $f=+$: el **acumulador** va de 3 a 6, 10, 15 y termina en 21

Entonces, `reduce(f, iterable)` recibe una función de **dos** valores y un iterable, y retorna el resultado de aplicar `f` al iterable `[s1, s2, s3, ..., sn]` de la forma `f(f(f(f(s1, s2), s3), s4), s5), ...`. Está en el módulo `functools`:

```
from functools import reduce

def suma(x: int, y: int) -> int:
    return x + y

reduce(suma, lista)      # -> 21
```

El inicializador. Podemos agregar un **inicializador** como tercer argumento: será el **primer valor que la función procese**, es decir, el primer valor del acumulador. En este ejemplo construimos una representación en *string* de una lista y usamos el inicializador para agregar una etiqueta:

```
def concatenar_textos(x, y) -> str:
    return f'{x}->{y}'

lista = [1, 2, 3, 4, 5]

reduce_sin_inicializador = reduce(concatenar_textos, lista)
reduce_con_inicializador = reduce(concatenar_textos, lista, '[Inicio]')

print(f'Sin inicializador: {reduce_sin_inicializador}')
print(f'Con inicializador: {reduce_con_inicializador}')
```

Salida

```
Sin inicializador: 1->2->3->4->5
Con inicializador: [Inicio]->1->2->3->4->5
```

Ejemplos aplicados. **1) Aplanamiento de listas.** Una lista con listas dentro, que queremos juntar en una sola gran lista (manteniendo el orden):

```python
def concatenar_listas(x: list, y: list) -> list:
    return x + y

lista_con_listas = [[1, 2], [3, 4], [5, 6], [7, 8, 9]]
lista_aplanada = reduce(concatenar_listas, lista_con_listas)
# -> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2) Unión o intersección de varios *sets*.

```python
def operacion_union(x: set, y: set) -> set:
    return x | y

def operacion_interseccion(x: set, y: set) -> set:
    return x & y

conjuntos = [{3, 5, 1}, {4, 3, 1}, {1, 2, 5}, {9, 5, 4, 1}]

resultado_union = reduce(operacion_union, conjuntos)
resultado_interseccion = reduce(operacion_interseccion, conjuntos)

print("Union:", resultado_union)
print("Interseccion:", resultado_interseccion)
```

Salida

```
Union: {1, 2, 3, 4, 5, 9}
Interseccion: {1}
```

3) Duplicación de elementos. Crear una cantidad de elementos que depende del valor de cada uno:

```python
def duplicar_elementos(x, y: int) -> tuple:
    # El * se usa aqui para desempaquetar valores
    return *x, *(y for _ in range(y))

reduce(duplicar_elementos, lista, [])
# -> (1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5)
```

¿Qué hace este código? El acumulador parte como la lista vacía `[]` (el inicializador). En cada paso, `*x` desempaqueta todo lo acumulado y `*(y for _ in range(y))` agrega `y` copias del elemento actual. Por eso el `3` aparece tres veces, el `4` cuatro veces, etc. Fíjate que el inicializador **es imprescindible** aquí: sin él, el primer acumulador sería el entero `1`, y no se podría desempaquetar.

4) Mínimos y máximos. Se puede obtener el máximo de una colección con `reduce`, por ejemplo sobre la unión de conjuntos que acabamos de obtener:

```python
# Hecho asi solo como ejemplo pedagogico
def maximo(x: int, y: int) -> int:
    if x >= y:
```

```
        return x
    return y

reduce(maximo, resultado_union)      # -> 9
```

No obstante, es **mucho más limpio** usar directamente las funciones `max` o `min` que ya nos entrega Python: `max(resultado_union)`.

★ Analogía de fonda

`reduce` es **cuadrar la caja al final del día**: vas sumando venta tras venta en un solo total que crece, hasta quedar con un único número. El inicializador es el "sencillo" con el que abres la caja en la mañana.

Precauciones con `reduce`.

! 1) Secuencias de un solo elemento, o vacías

Si la secuencia tiene **un único elemento**, `reduce` retorna ese elemento **sin aplicar la función**. Y si la secuencia está **vacía** y no hay inicializador, se levanta un `TypeError`. Un buen inicializador suele ser el **elemento neutro** de la operación (para la suma, el `0`):

```
reduce(suma, [1])          # -> 1       (no se aplica suma)
reduce(suma, [1], 0)       # -> 1       (suma(0, 1))

try:
    reduce(suma, [])
except TypeError as error:
    print('TypeError:', error)
# TypeError: reduce() of empty iterable with no initial value

reduce(suma, [], 0)        # -> 0
```

! 2) Operaciones no conmutativas

En ciertas operaciones el resultado **depende del orden** de los elementos. En la división, $\frac{x}{y} \neq \frac{y}{x}$, mientras que en la suma $x + y = y + x$. Con una operación sensible al orden, invertir la secuencia cambia el resultado:

```
def division(x: int, y: int) -> float:
    return x / y

numeros = [3, 5, 7, 9, 11]
reduce(division, numeros)          # -> 0.0008658008658008659
reduce(division, numeros[::-1])    # -> 0.01164021164021164
```

! 3) Reducir *sets* u otros iterables sin orden

Si la operación depende del orden y el iterable **no tiene un orden fijo** (como un *set*, o un iterador que mezcla sus elementos), el resultado puede no ser el esperado. Aquí tenemos un iterador que mezcla las palabras antes de recorrerlas (como los iteradores personalizados del resumen anterior); al concatenarlas, el orden final cambia en **cada ejecución**:

```
from random import shuffle
from typing import Self
```

```python
class IteradorListaPalabrasDesordenadas:
    def __init__(self, iterable: list) -> None:
        self.iterable = iterable.copy()

    def __iter__(self) -> Self:
        shuffle(self.iterable)
        return self

    def __next__(self) -> str:
        if not self.iterable:
            raise StopIteration("Llegamos al final")
        return self.iterable.pop(0)

def concatenar_textos(x: str, y: str) -> str:
    return f"{x} {y}"

for i in range(1, 6):
    palabras = IteradorListaPalabrasDesordenadas(['casa', 'mar', '
   ventana', 'roca', 'piso'])
    print(f'{i}. {reduce(concatenar_textos, palabras)}')
# Cinco ejecuciones = cinco ordenes (probablemente) distintos
```

4.1.4 Funciones `lambda` o funciones anónimas

Hasta ahora, para cada `map`, `filter` o `reduce` hemos tenido que **definir una función con `def`**, aunque fuera de una sola línea. Las funciones `lambda` nos permiten usar estas herramientas **sin definir funciones con `def`**. Pero antes necesitamos entender **cómo trata Python a las funciones**.

Funciones de primera clase. En Python, las funciones son **de primera clase** (*first-class functions*): se tratan como cualquier otra variable (objeto). No en todos los lenguajes es así (por ejemplo, en Java no). Esto tiene consecuencias:

■ Consecuencia 1: una función se puede asignar a una variable

```python
def suma(x: int, y: int) -> int:
    return x + y

adicion = suma

# Ambas son LA MISMA funcion
print(adicion)                  # <function suma at 0x...>
print(suma)                     # <function suma at 0x...>
print(suma(3, 5), adicion(3, 5))    # 8 8
```

■ Consecuencia 2: una función se puede pasar como parámetro a otra función

```python
from typing import Callable

def saludar_senora(nombre: str) -> str:
    return "Señora " + nombre

def saludar_joven(nombre: str) -> str:
```

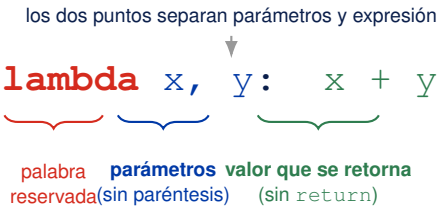
```
    return "Joven " + nombre

def saludar_tarde(funcion_saludo: Callable, nombre: str) -> str:
    return "Buenas tardes " + funcion_saludo(nombre)

print(saludar_tarde(saludar_senora, "Valeria"))    # Buenas tardes Señ
    ora Valeria
print(saludar_tarde(saludar_joven, "Bon"))         # Buenas tardes Joven
     Bon
```

Justamente eso hacemos con `map`, `filter` y `reduce`: les pasamos una función como argumento.

La sintaxis `lambda`. Una función `lambda` es una **forma alternativa de definir funciones**. Aparte de su nombre griego, no tiene nada de intimidante:



La sintaxis es `lambda <parametros>: <valor a retornar>`. No se necesita `return`: la expresión a la derecha de los dos puntos **es** el valor que se devuelve. Veamos funciones que **retornan** una `lambda`, junto a su equivalente con `def` anidado:

```
# Funcion que retorna una lambda
def retornar_sucesor():
    return lambda x: x + 1

sucesor = retornar_sucesor()

# Equivalente con def:
def sumar_uno():
    def sucesor(x):
        return x + 1
    return sucesor

sucesor_funcion = sumar_uno()

print("2 + 1 =", sucesor(2))             # 2 + 1 = 3
print("2 + 1 =", sucesor_funcion(2))     # 2 + 1 = 3
```

```
def restar_por(n: int):
    return lambda x: x - n

restar_3_a = restar_por(3)

# Equivalente con def:
def disminuir_por(n: int):
    def disminuir(x):
        return x - n
    return disminuir

disminuir_3_a = disminuir_por(3)
```

```
print("5 - 3 =", restar_3_a(5))          # 5 - 3 = 2
print("5 - 3 =", disminuir_3_a(5))       # 5 - 3 = 2
```

¿Qué hace este código? `restar_por(3)` devuelve una función que **"recuerda"** el valor `n = 3` del contexto donde fue creada (a esto se le llama *clausura* o *closure*). Es una **fábrica de funciones**: `restar_por(10)` nos daría otra función que resta 10.

Anónimas. Lo que distingue a las `lambda` es que pueden ser **anónimas**: no reciben un nombre propio. Si consultamos el atributo `__name__`:

```
sucesor.__name__              # -> '<lambda>'
restar_3_a.__name__           # -> '<lambda>'
disminuir_3_a.__name__        # -> 'disminuir'   (definida con def, si tiene
    nombre)
```

En resumen, las `lambda` son una forma de definir funciones **sin declararlas como variable** (no hace falta `def`), que **requiere menos código** y es **anónima**. Combinan perfecto con `map`, `filter` y `reduce`. Reescribamos algunos ejemplos anteriores:

```
strings = ['Señores pasajeros', 'Disculpen', 'mi', 'IntencIÓN', 'no', 'Es',
    'MolEstar']
mapeo = map(lambda x: x.lower(), strings)
print(', '.join(mapeo))

set_filtrado = filter(lambda x: x < 10, {100, 1, 5, 9, 91, 1})
print(list(set_filtrado))

lista = [1, 2, 3, 4, 5]
reduce_sin_inicializador = reduce(lambda x, y: f'{x}->{y}', lista)
reduce_con_inicializador = reduce(lambda x, y: f'{x}->{y}', lista, '[Inicio
    ]')
```

★ Analogía de fonda

Una `lambda` es **una receta anotada en una servilleta**: sirve para un uso puntual, no lleva nombre y no vale la pena archivarla en el recetario (`def`). Si la receta se pone larga o la vas a usar en muchos lugares, mejor escribirla con `def` y darle un buen nombre.

★ Buenas prácticas con `lambda`

Una `lambda` solo puede contener **una expresión** (no sentencias como `for`, `try` o asignaciones). Si la lógica es larga, si necesitas reutilizarla, o si un buen nombre ayudaría a entenderla, es preferible una función con `def`. Por la misma razón, una función que retorna otra (como `restar_por`) ya es un caso de *orden superior*: recibe o retorna funciones.

4.2 Funciones que no necesitan de una función

4.2.1 `enumerate`

`enumerate()` entrega una especie de generador que retorna **tuplas** `(indice, elemento)`: el primer objeto de cada tupla es el índice, y el segundo, el ítem original. Si queremos iterar una lista y necesitamos tanto el índice como el valor, una forma poco *pythonic* sería:

```
lista = ["a", "b", "c", "d"]

for indice in range(len(lista)):
```

```
    elemento = lista[indice]
    print(f"{indice}: {elemento}")
```

La función `enumerate` hace exactamente lo mismo, pero de forma más elegante:

```
for indice, elemento in enumerate(lista):
    print(f"{indice}: {elemento}")
```

Salida

```
0: a
1: b
2: c
3: d
```

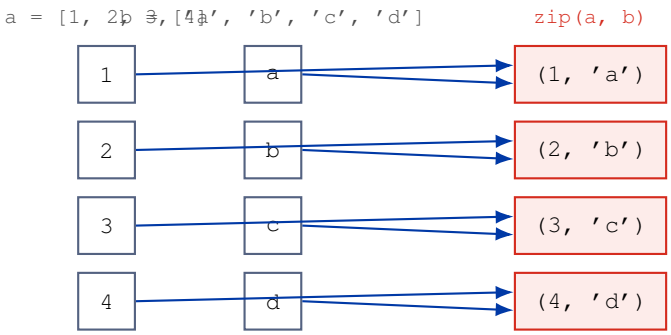
¿Qué hace este código? `enumerate(lista)` retorna un objeto de tipo `enumerate` que se comporta como un generador (se puede usar con `next`, y por eso funciona en un `for`). En cada vuelta entrega una tupla que el `for` **desempaqueta** directamente en `indice, elemento`. Ya no hay que calcular índices a mano.

★ Empezar desde otro número

`enumerate` acepta un segundo argumento opcional, `start`, para que la numeración parta desde otro valor: `enumerate(lista, start=1)` entrega `(1, "a"), (2, "b"), ...`. Es ideal para mostrar posiciones "humanas" que parten en 1.

4.2.2 zip

`zip` toma **dos o más** secuencias o iterables y retorna un iterador que entrega **tuplas**, donde cada tupla contiene los elementos **i-ésimos** de cada iterable. La cantidad de elementos que entrega es igual al **menor** de los largos. Igual que `enumerate`, retorna un objeto de tipo `zip` que se comporta como un generador: ocupa poco espacio y genera los elementos a medida que se necesitan.



A modo de ejemplo, tenemos una tupla con los *headers* (nombres de columnas) de un CSV y una tupla con los datos de una persona. Queremos una lista de tuplas que asocie cada *header* con su valor:

```
columnas = ("nombre", "apellido", "email")
persona = ("Juan", "Perez", "jp1@hotmail.com")

list(zip(columnas, persona))
```

Salida

```
[('nombre', 'Juan'), ('apellido', 'Perez'), ('email', 'jp1@hotmail.com')
    ]
```

Si ampliamos el ejemplo a una lista de personas, podemos usar el **operador** `*` para **desempaquetar** la lista como argumentos separados:

```
columnas = ("nombre", "apellido", "email")
personas = [
    ("Juan", "Perez", "jp1@hotmail.com"),
    ("Gonzalo", "Aldunate", "gan@gmail.com"),
    ("Alberto", "Gomez", "agomez@yahoo.com")
]

# Si personas = [p1, p2, p3], esto equivale a zip(columnas, p1, p2, p3)
list(zip(columnas, *personas))
```

Salida

```
[('nombre', 'Juan', 'Gonzalo', 'Alberto'),
 ('apellido', 'Perez', 'Aldunate', 'Gomez'),
 ('email', 'jp1@hotmail.com', 'gan@gmail.com', 'agomez@yahoo.com')]
```

! `zip` corta en el iterable más corto

Si un iterable tiene más elementos que otro, los sobrantes **se descartan**. Si quieres que `zip` considere el iterable **más largo**, usa `zip_longest` del módulo `itertools`, que rellena con `None` (o con el valor que le indiques):

```
from itertools import zip_longest

columnas = ("nombre", "apellido", "email")
persona = ("Juan", "Perez", "jp1@hotmail.com", "+56123123??")

list(zip(columnas, persona))
# [('nombre', 'Juan'), ('apellido', 'Perez'), ('email', 'jp1@hotmail.
    com')]
list(zip_longest(columnas, persona))
# [..., (None, '+56123123??')]
```

`zip` como inversa de sí misma. Usando `zip` junto al operador `*` (que desempaqueta listas o tuplas en argumentos de una función), podemos **deshacer** un `zip`:

```
a = [1, 2, 3, 4]
b = ["a", "b", "c", "d"]

zipped = list(zip(a, b))
print(zipped)       # [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

unzipped = list(zip(*zipped))
print(unzipped)     # [(1, 2, 3, 4), ('a', 'b', 'c', 'd')]
```

¿Qué hace este código? Al ejecutar `zip(*zipped)`, el desempaquetamiento entrega a `zip` **las cuatro tuplas por separado**: `(1, 'a')`, `(2, 'b')`, `(3, 'c')`, `(4, 'd')`. Cada una tiene largo

2, así que `zip` junta **todos los primeros elementos** `(1, 2, 3, 4)` y luego **todos los segundos** `('a', 'b', 'c', 'd')`: volvimos a las listas originales.

★ Analogía de fonda

`zip` es **armar las parejas de cueca**: el primer bailarín con la primera bailarina, el segundo con la segunda, y así. Si de un lado sobra gente, los sobrantes se quedan sin pareja y **no bailan** (`zip` corta en el más corto). Y `zip(*parejas)` es volver a separar a los dos coros.

✓ Resumen express: funciones sobre iterables

- `map(f, it, ...)` **transforma** cada elemento; `filter(f, it)` **selecciona** los que cumplen; `reduce(f, it[, ini])` **acumula** todo en un único valor (`from functools import reduce`).
- `map`, `filter`, `enumerate` y `zip` retornan **iteradores perezosos** (se consumen una sola vez); `reduce` retorna un valor.
- Las funciones son **de primera clase**: se asignan, se pasan como argumento y se retornan. `lambda` es una función anónima de una sola expresión.
- `enumerate` agrega el **índice**; `zip` **combina** iterables posición a posición, y `zip(*x)` lo deshace.

5 Aplicaciones

Con generadores, `map`, `filter`, `reduce`, `lambda`, `zip` y `enumerate` ya tenemos las piezas. Ahora veremos herramientas que Python trae "de fábrica" para simplificar cálculos sobre iterables, y casos completos donde todo se combina.

5.1 Funciones *built-in* de Python

Python trae muchas funciones ya implementadas, principalmente para **simplificar y abstraer** cálculos que se pueden aplicar a objetos de clases distintas (*duck typing*). Puedes revisarlas todas en la documentación oficial de funciones de Python. Veamos algunas.

5.1.1 `reversed`

`reversed()` recibe una **secuencia** cualquiera y entrega sus elementos **en orden inverso**. Podemos personalizar su comportamiento haciendo *overriding* del método `__reversed__` en nuestra clase. Si no lo hacemos, se usa el mecanismo *built-in*, que recorre el objeto **hacia atrás** usando `__len__` y `__getitem__` (es decir, accede a los índices `len - 1, len - 2, ..., 0`).

Como ejemplo, definimos un tipo especial de lista que **sobrescribe** `__reversed__`: en vez de invertir el orden, **intercambia la primera mitad con la segunda**:

```
from functools import reduce

class ListaReversaMitad(list):
    def __init__(self, *args) -> None:
        super().__init__(args)

    def __reversed__(self) -> list:
        mitad = len(self) // 2
        return self[mitad:] + self[:mitad]

lista = [1, 2, 3, 4, 5, 6]
```

```python
for secuencia in lista, ListaReversaMitad(*lista):
    print(f"Clase {type(secuencia).__name__}: ", end="")
    print(reduce(lambda x, y: f'{x}, {y}', reversed(secuencia)))
```

Salida

```
Clase list: 6, 5, 4, 3, 2, 1
Clase ListaReversaMitad: 4, 5, 6, 1, 2, 3
```

¿Qué hace este código? El mismo llamado `reversed(secuencia)` se comporta distinto según la clase del objeto: para una `list` normal invierte el orden, y para `ListaReversaMitad` usa **su propio** `__reversed__`. Es *duck typing* en acción: `reversed` no pregunta el tipo, solo "pregunta" si el objeto sabe revertirse.

5.1.2 Otras funciones *built-in* útiles sobre iterables

Función	Qué hace
`sum(it)`	**Suma** todos los elementos del iterable.
`min(it)` / `max(it)`	Retornan el **menor** / el **mayor** elemento del iterable.
`all(it)`	`True` si **todos** los elementos son verdaderos (o si el iterable está vacío).
`any(it)`	`True` si **al menos uno** de los elementos es verdadero.

Las primeras tres tienen un enfoque **matemático** (`sum`, `min`, `max`) y las últimas dos, **booleano** (`all`, `any`). Son especialmente útiles en dos situaciones:

1) Obtener la cantidad de elementos de un generador, sin transformarlo en otra estructura. Los generadores **no** tienen el método `__len__`, así que `len` falla:

```python
generador = (i for i in range(10))
try:
    largo = len(generador)
except TypeError as error:
    print('TypeError:', error)
# TypeError: object of type 'generator' has no len()
```

Pero con `sum` sí podemos calcular su "tamaño": sumamos un `1` por cada elemento que entrega.

```python
generador = (i for i in range(10))

largo = sum(1 for _ in generador)
print(f'Largo generador: {largo}')      # Largo generador: 10
```

! Contar consume el generador

Para contar, `sum` tiene que **recorrer todo** el generador, y como un generador se agota, **después de contar ya no quedan elementos**. Si además necesitas los valores, o cuentas y consumes al mismo tiempo, o bien trabajas con una estructura en memoria.

2) Verificar si todos o alguno de los elementos cumplen una condición.

```
iterable = {1, 3, 5, 11, 8, 9}

todos_son_impares = all(map(lambda x: x % 2 == 1, iterable))
contiene_un_par = any(map(lambda x: x % 2 == 0, iterable))

print(f'Todos los elementos son impares: {todos_son_impares}')    # False
print(f'El iterable contiene al menos un par: {contiene_un_par}')    # True
```

¿Qué hace este código? `map` transforma cada número en un booleano ("¿es impar?", "¿es par?") y `all`/`any` resumen esa secuencia en un solo valor. Como hay un `8` en el *set*, no todos son impares, y sí existe al menos un par.

★ Dos detalles útiles

(1) `all` y `any` **dejan de revisar apenas conocen la respuesta** (`all` se detiene en el primer falso; `any`, en el primer verdadero), así que con generadores pueden no consumir todo. (2) En el *notebook* original de este ejemplo, las condiciones estaban al revés respecto al nombre de las variables (`todos_son_impar` preguntaba por pares); acá están corregidas para que el nombre coincida con lo que calculan.

5.2 Librerías *built-in*

Python también incluye librerías que facilitan el trabajo funcional. Dos de las más útiles son `itertools` y `collections`.

5.2.1 `itertools`

La librería `itertools` tiene funciones especializadas en **trabajar con iteradores**. Sus funciones se pueden agrupar así:

Grupo	Función	Qué entrega
Iteradores infinitos	`count(start, step)`	Cuenta desde `start` aumentando de a `step` (por defecto, desde `0` de a `1`).
	`cycle(it)`	Repite el iterable una y otra vez, en ciclo.
	`repeat(x, n)`	Repite `x` (infinitas veces, o `n` veces si se indica).
Combinatoria	`product(a, b, ...)`	Producto cartesiano: todos los pares (tuplas) entre los iterables.
	`permutations(it, r)`	Todas las ordenaciones de `r` elementos (importa el orden).
	`combinations(it, r)`	Todas las combinaciones de `r` elementos (**no** importa el orden).
Otras	`zip_longest(...)`	Como `zip`, pero hasta el iterable **más largo**.
	`tee(it, n)`	Divide un iterador en `n` iteradores independientes (permite recorrerlo más de una vez).

Y hay muchas más: te invitamos a investigar la documentación y evaluar en qué casos pueden servirte. Veamos algunos ejemplos.

`count()` : un generador infinito que cuenta; podemos cambiar el valor inicial (`start`) y el incremento (`step`).

```python
from itertools import count

contador = count(22, 0.5)

for _ in range(6):
    print(next(contador))      # 22, 22.5, 23.0, 23.5, 24.0, 24.5
```

`combinations()` : recibe un iterable y un entero `r`, y entrega un generador con las **combinaciones de tamaño `r`** de sus elementos.

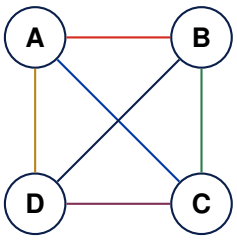
```python
from itertools import combinations

combinaciones_grupos_2 = combinations('ABCD', 2)
combinaciones_grupos_4 = combinations('ABCD', 4)

print(list(combinaciones_grupos_2))
print(list(combinaciones_grupos_4))
```

Salida

```
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')
    ]
[('A', 'B', 'C', 'D')]
```



`combinations('ABCD', 2)` = **6 pares**.
Cada línea es una combinación:
`('A','B')` y `('B','A')`
son **la misma** (no importa el orden).
Con `permutations` serían 12.

Otros ejemplos rápidos.

```
from itertools import permutations, product, cycle, repeat

list(permutations('ABC', 2))
# [('A','B'), ('A','C'), ('B','A'), ('B','C'), ('C','A'), ('C','B')]

list(product('AB', [1, 2, 3]))
# [('A',1), ('A',2), ('A',3), ('B',1), ('B',2), ('B',3)]

ciclo = cycle('ab')                    # a b a b a b ... (infinito)
list(repeat('x', 3))                   # ['x', 'x', 'x']
```

5.2.2 `collections`

La librería `collections` nos entrega **alternativas** a `dict`, `list`, `set` y `tuple`. Entre lo que ya hemos visto está:

- `namedtuple`: tuplas cuyos campos tienen **nombre** (acceso con `tupla.campo`).
- `deque`: cola de doble extremo, eficiente para agregar y quitar por ambos lados.
- `defaultdict`: diccionario que crea un **valor por defecto** para llaves inexistentes.

Y además cuenta con otros objetos, como `Counter`, una **subclase de diccionario** que **cuenta los elementos de un iterable**: cada elemento queda como **llave** y su recuento como **valor**. Ojo: para que `Counter` funcione, los elementos del iterable deben ser **hasheables**.

```
from collections import Counter
from random import randint

iterable = [randint(1, 3) for _ in range(20)]
contador = Counter(iterable)
contador        # p. ej.: Counter({2: 9, 3: 6, 1: 5})    (varia en cada
    ejecucion)
```

También sirve para encontrar los **elementos más repetidos**, con `most_common`. Recuerda que los *strings* también son iterables:

```
texto = 'supercalifragilisticoespialidoso'
contador = Counter(texto)
contador.most_common(3)
```

Salida

```
[('i', 6), ('s', 4), ('a', 3)]
```

★ Analogía de fonda

`Counter` es **el conteo de ventas de la fonda**: le muestras la lista de todo lo que se vendió (terremotos, empanadas, anticuchos…) y te devuelve cuántas unidades hubo de cada cosa; `most_common(3)` te dice el *top* tres del día.

5.3 Casos aplicados

Para cerrar, veamos casos donde se combinan los contenidos. Trabajaremos con datos de **lenguas extintas o en peligro de extinción**. Cada registro tiene: ID, nombre de la lengua, país donde se habla, código del país, grado de peligro y cantidad de hablantes. Lo modelamos con una `namedtuple` y un generador:

```
from collections import namedtuple
from typing import Generator

LenguaExtinta = namedtuple(
    'LenguaExtinta', 'id, name, country, country_code, endangerment,
   speakers')

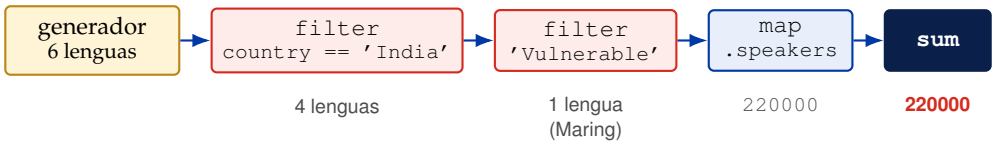
def generador_lenguas_extintas() -> Generator:
    yield LenguaExtinta(349, 'Juang', 'India', 'IND', 'Definitely
   endangered', 230000)
    yield LenguaExtinta(350, 'Tiwa', 'India', 'IND', 'Definitely endangered
   ', 230000)
    yield LenguaExtinta(351, 'Embera (Panama)', 'Panama', 'PAN', '
   Vulnerable', 224850)
    yield LenguaExtinta(352, 'Ediamat', 'Guinea-Bissau', 'GNB', 'Vulnerable
   ', 220250)
    yield LenguaExtinta(353, 'Mara', 'India', 'IND', 'Definitely endangered
   ', 220000)
    yield LenguaExtinta(354, 'Maring', 'India', 'IND', 'Vulnerable',
   220000)
```

5.3.1 Consultas anidadas

Una gran característica de la programación funcional es que podemos **anidar consultas**, siempre que cada una retorne un generador u otro iterable. Por ejemplo: *"¿cuántos hablantes totales tienen las lenguas de India que están en estado vulnerable?"*

```
lenguas_india = filter(lambda lengua: lengua.country == 'India',
                       generador_lenguas_extintas())
lenguas_vulnerables = filter(lambda lengua: lengua.endangerment == '
   Vulnerable',
                             lenguas_india)
hablantes_totales = sum(map(lambda lengua: lengua.speakers,
   lenguas_vulnerables))

print('Los hablantes totales son:', hablantes_totales)    # 220000
```



¿Qué hace este código? Cada consulta recibe un iterable y entrega otro, así que se pueden encadenar como una **tubería**: primero nos quedamos con las de India (4 lenguas), luego con las vulnerables (queda solo *Maring*), después extraemos sus hablantes con `map`, y `sum` entrega el total: `220000`. Como todo es perezoso, **nada se calcula** hasta que `sum` pide valores: entonces cada registro recorre *toda* la tubería antes de pedir el siguiente.

También podemos **unir consultas**, ya sea con condiciones combinadas en una `lambda` o como argumentos anidados:

```
hablantes_totales = sum(
    map(lambda lengua: lengua.speakers,
        filter(lambda lengua: (lengua.country == 'India')
                              and (lengua.endangerment == 'Vulnerable'),
               generador_lenguas_extintas())))
print('Los hablantes totales son:', hablantes_totales)    # 220000
```

5.3.2 Unir generadores

A veces necesitamos **combinar dos o más generadores** según ciertas condiciones. Aquí buscamos todos los pares (uno de cada generador) cuya suma sea múltiplo de 5:

```
from itertools import product
from random import randint

generador_1 = (randint(0, 20) for _ in range(10))
generador_2 = (randint(20, 50) for _ in range(10))

combinaciones = product(generador_1, generador_2)

# 'product' entrega tuplas (a, b): para operar sobre ellas debemos
# indexarlas (x[0], x[1]) o usar funciones que operen sobre iterables.
pares_multiplos_de_5 = filter(lambda x: (x[0] + x[1]) % 5 == 0,
    combinaciones)

print(list(pares_multiplos_de_5))
```

¿Qué hace este código? `product` entrega las $10 \times 10 = 100$ combinaciones posibles (un elemento de cada generador), y `filter` se queda solo con los pares cuya suma es divisible por 5. Los números son aleatorios, así que el resultado cambia en cada ejecución.

★ Un detalle sobre `product`

Para poder combinar todo con todo, `product` necesita **recorrer por completo** cada iterable de entrada (los guarda internamente). Es decir, con generadores muy grandes o infinitos **no es perezoso en sus entradas**: úsalo con conjuntos de tamaño razonable.

5.3.3 Agrupar información con diccionarios

Aprovechemos la lista de lenguas y agrupemos la **cantidad de lenguas por país**. Hay varias maneras de lograrlo:

1) Con estructuras por comprensión. Como aquí recorremos los datos más de una vez, primero los pasamos a lista (en programación funcional "pura" se podría usar `tee` de `itertools`):

```
paises = map(lambda lengua: lengua.country, generador_lenguas_extintas())
lista_paises = list(paises)

contador = {pais: lista_paises.count(pais) for pais in set(lista_paises)}
print(contador)      # {'India': 4, 'Panama': 1, 'Guinea-Bissau': 1}  (orden
    puede variar)
```

2) Con el objeto `Counter`.

```
from collections import Counter

paises = map(lambda lengua: lengua.country, generador_lenguas_extintas())
contador = Counter(paises)
print(contador)      # Counter({'India': 4, 'Panama': 1, 'Guinea-Bissau':
   1})
```

3) Con `reduce`.

```
from functools import reduce

def funcion_auxiliar(diccionario, lengua) -> dict:
    pais = lengua.country

    if pais not in diccionario:
        diccionario[pais] = 0

    diccionario[pais] += 1
    return diccionario

contador = reduce(funcion_auxiliar, generador_lenguas_extintas(), {})
print(contador)      # {'India': 4, 'Panama': 1, 'Guinea-Bissau': 1}
```

¿Qué hace este código? El inicializador `{}` es el diccionario acumulador. En cada paso, `funcion_auxiliar` recibe el diccionario acumulado y la lengua actual, suma 1 al país correspondiente y **retorna** el diccionario para el siguiente paso. La versión con `Counter` es la más corta y legible; la de `reduce` es la más "funcional" en espíritu, aunque (ojo) **muta** el diccionario acumulador, por lo que no es estrictamente pura.

5.3.4 Eficiencia: generadores vs. estructuras que usan memoria

Finalmente, comparemos hacer consultas **solo con generadores** contra usar **estructuras en memoria** (`list`, `dict`, `tuple`, `set`...). Los tiempos de estos códigos están acotados, así que el *notebook* muestra un extracto de la implementación y los resultados de ejecutarlos con `python -B -m unittest -v` (el `-B` evita usar caché, para tener valores más comparables), sobre el archivo de lenguas extintas.

1) Leer un archivo línea por línea vs. leerlo completo. Podemos revisar el contenido de a poco (`for line in file`) o cargarlo todo (`file.readlines()`):

```
def cargar_archivo_generador() -> Generator:
    with open(ruta_archivo, encoding='utf-8') as file:
        for line in file:
            yield line.strip().split(',')
```

```python
def cargar_archivo_memoria() -> Generator:
    with open(ruta_transacciones, encoding='utf-8') as file:
        for line in file.readlines():
            yield line.strip().split(',')
```

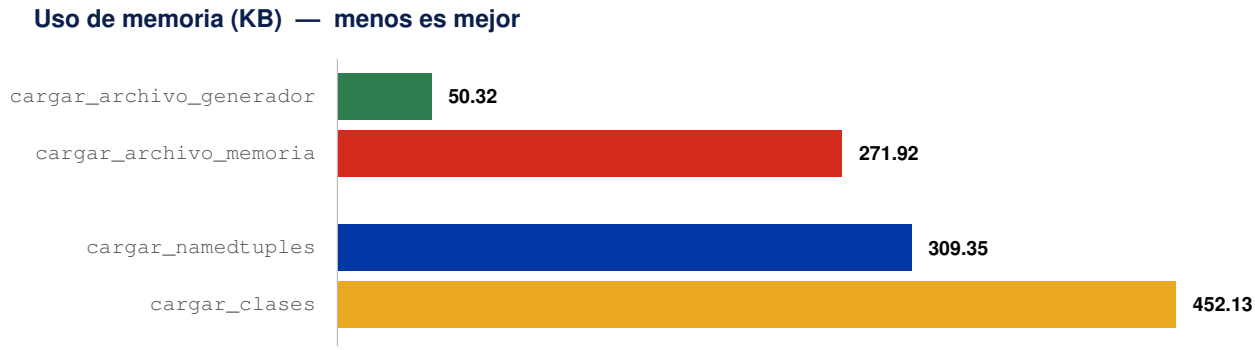
Los tiempos totales son muy similares, pero el uso de memoria no:

2) Cargar los datos como `namedtuple` vs. como clases. Otro punto importante con grandes cantidades de datos es decidir si se cargan como `namedtuple` o como **clases personalizadas**:

```python
def cargar_namedtuples() -> Generator:
    with open(ruta_archivo, encoding='utf-8') as file:
        file.readline()
        for line in file:
            info = (int(x) for x in line.strip().split(',')[::5])
            yield NamedTupleLengua(*info)

def cargar_clases() -> Generator:
    with open(ruta_archivo, encoding='utf-8') as file:
        file.readline()
        for line in file:
            info = (int(x) for x in line.strip().split(',')[::5])
            yield ClaseLengua(*info)
```

Nuevamente los tiempos son muy similares, pero la `namedtuple` ocupa bastante menos memoria que la clase.



✓ Qué aprendimos sobre eficiencia

Al trabajar con generadores conviene pensar el código en términos de:

- **Eficiencia**: *cómo se carga* la información (línea a línea vs. todo junto) y *cómo se maneja* (`namedtuple` vs. clases).
- **Comportamiento**: *qué hace* cada función y *qué retorna*.

A igual tiempo de ejecución, el enfoque perezoso usa **mucha menos memoria**, y la diferencia se vuelve más notoria (también en tiempo) mientras más grandes son los datos.

6 Hoja de referencia

Herramienta	Qué hace	Retorna	Equivale a
`yield`	Entrega un valor y pausa la función	(función → generador)	—
`send(v)`	Envía `v` al `yield` pausado	siguiente valor del generador	`next` = `send(None)`
`yield from X`	Delega en otro iterable	(dentro de un generador)	`for e in X: yield e`
`map(f, it)`	Aplica `f` a cada elemento	iterador	`(f(x) for x in it)`
`map(f, a, b)`	Aplica `f(x, y)` en paralelo	iterador (largo del menor)	`(f(x,y) for x,y in zip(a,b))`
`filter(f, it)`	Deja los que cumplen `f`	iterador	`(x for x in it if f(x))`
`reduce(f, it, ini)`	Acumula todo con `f(acum, x)`	**un valor**	`f(f(f(ini, s1), s2), s3)...`
`lambda x: expr`	Función anónima de una expresión	función	`def f(x): return expr`
`enumerate(it)`	Agrega el índice	iterador de `(i, x)`	`zip(count(), it)`
`zip(a, b)`	Junta por posición	iterador de tuplas	corta en el más corto
`zip(*pares)`	Deshace un `zip`	iterador de tuplas	—
`sum / min / max`	Suma, mínimo, máximo	un valor	`reduce` con `+`, `min`, `max`
`all / any`	Todos / alguno verdadero	`bool`	—
`Counter(it)`	Cuenta elementos hasheables	dict especial	`reduce` con dict acumulador

! Errores frecuentes

- **Consumir dos veces** un iterador o generador (`map`, `filter`, `zip`, expresión generadora): la segunda vez está vacío.
- **Olvidar consumir** un `map` o un `filter`: por sí solos no ejecutan nada hasta que se recorren (`list`, `for`, `sum`...).
- Usar `reduce` sobre una secuencia **vacía** sin inicializador (`TypeError`), o con una operación **no conmutativa** sobre un iterable sin orden.
- Hacer `send` a un generador **antes del primer** `next` (`TypeError`).
- Poner `print`, modificar globales o mutar argumentos en funciones que deberían ser puras,

y luego usarles `@cache`.

- Creer que `zip` conserva los elementos sobrantes: corta en el iterable más corto (para lo contrario, `zip_longest`).

✓ Checklist de código funcional

1. ¿La función depende **solo** de sus parámetros y no toca nada fuera de ella?
2. ¿Estoy **transformando** datos (retornando valores nuevos) en vez de mutar los existentes?
3. ¿Necesito **todos** los datos a la vez, o me basta un generador que los entregue de a uno?
4. ¿Puedo expresar el ciclo como `map`, `filter` o `reduce` (o con `Counter`, `sum`, `any`...) de forma **más clara** que con un `for`?
5. Si una `lambda` se pone larga o repetida, ¿mejor una `def` con nombre?

Fin del resumen • ¡Felices fiestas y a programar funcional!