Clase 007 — Comprehensions y generadores

Parte: 0 — Prerrequisitos · Fuente: Ramalho, Fluent Python 2e — caps. 2 (Sequences) y 17 (Iterators, Generators, Coroutines). · Duración estimada: 90 min.

Clase 007 — Comprehensions y generadores

> *Parte: 0 — Prerrequisitos · Fuente: Ramalho, Fluent Python 2e — caps. 2 (Sequences) y 17 (Iterators, Generators, Coroutines). · Duración estimada: 90 min.*

Objetivo

Que el alumno escriba código Python idiomático: list/dict/set comprehensions en vez de for+append, generadores cuando el dataset no cabe en memoria, y entienda la diferencia fundamental entre construir una lista y producir un iterable perezoso.

Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Convertir loops for+append a list/dict/set comprehensions sin perder legibilidad.
2. Usar generadores (yield y generator expressions) para procesar datos que no caben en RAM.
3. Distinguir [x for x in xs] (lista) vs (x for x in xs) (generador): memoria y consumo.
4. Encadenar generadores con itertools (chain, islice, takewhile, groupby).
5. Identificar cuándo NO usar comprehension (lógica compleja, side effects, debug difícil).

Temas

#	Tema	Por qué importa
1	List comprehension: [expr for x in xs if c	Idiomático, eficiente, legible si es simpl
2	Dict/set comprehensions	Mismo patrón, otra estructura.
3	Generator expressions: (expr for x in xs)	Perezoso, memoria O(1).
4	Funciones generadoras con yield	Reescribe procesos como streams.
5	itertools — la caja de herramientas	chain, islice, groupby, accumulate, combin
6	Comprehension vs loop: cuándo NO	Lógica >2 líneas, side effects, debug.

Definiciones y características

List comprehension

: Expresión [expr for x in iterable if cond] que construye una lista completa en memoria. Equivalente idiomático a for + append + if. Más rápida y legible si la expresión es simple.

Generator expression

: Lo mismo pero con paréntesis (expr for x in iterable if cond). Devuelve un generador perezoso: produce cada valor on-demand, memoria O(1). Solo puedes recorrerlo una vez.

Iterador

: Objeto con método __next__() que devuelve el siguiente valor o lanza StopIteration. Es lo que está detrás de un for. Características: lazy, single-pass, memoria O(1).

Generador

: Función con yield (o generator expression) que produce un iterador. Cada yield pausa la función y emite un valor; al siguiente next() retoma desde ahí. Mantiene el estado local entre llamadas.

itertools

: Librería stdlib con bloques perezosos componibles: chain (concatena), islice (slicing perezoso), groupby (agrupa consecutivos), accumulate (sum/prod corridos), takewhile, combinations, product.

Dataset / recursos

Datos sintéticos: rango grande de números (1M elementos) para mostrar diferencia memoria lista vs generador. Sin descarga.

Ejercicios

1. De for a comprehension. Toma 3 loops for+append (cuadrados, filtra pares, mapea a strings) y conviértelos.

2. Generador de Fibonacci infinito. Función con yield que produce Fibonacci. Úsala con itertools.islice para tomar los primeros 20.

3. Memoria: lista vs generador. Mide RAM (con tracemalloc) de sum([ii for i in range(10_000_000)]) vs sum(ii for i in range(10_000_000)). Reporta la diferencia.

4. Procesa CSV línea por línea. Lee un archivo grande con yield línea por línea, filtra por una condición, cuenta sin cargar todo en memoria.

5. Pivot con dict comprehension. Dada list[tuple[str, int]] (nombre, puntaje), construye dict[str, list[int]] agrupando puntajes por nombre.

Homework verificable

Notebook que: (1) reescribe 3 loops como comprehensions, (2) implementa generador Fibonacci con islice, (3) comparativa RAM lista vs generador con tracemalloc y tabla de resultados, (4) lee un CSV ≥10k filas con generador y filtra sin cargar entero.

Criterio de aceptación: La medición de RAM muestra >100× menos memoria con generador. CSV se procesa sin OOM.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Iteré un generador y la segunda vez está v	Los generadores son single-pass. Tras cons
MemoryError al hacer [expensive(x) for x i	Construyes lista completa en RAM. Fix: usa
itertools.groupby me agrupa raro	Solo agrupa elementos consecutivos con mis
Generator expression dentro de función fal	Estás haciendo gen[0] — generadores no son
List comprehension con if-else no funciona	if al final filtra; if-else va al principi

Preguntas frecuentes

¿Cuándo comprehension y cuándo for clásico?

Comprehension cuando es una expresión clara en 1 línea. For clásico cuando hay >2 statements, side effects (print, mutación), o la lógica es más legible explícita.

¿Generator expression o list?

Generator si el resultado solo se consume una vez y N es grande (RAM importa). List si necesitas recorrer 2+ veces, indexar, o hacer len(). Truco: sum(xx for x in xs) evita lista temporal vs sum([xx for x in xs]).

¿Cuánto más rápido es vs un for tradicional?

~10-30%, no mil veces más. La ganancia real es legibilidad. Si necesitas mil veces, no es comprehension lo que buscas — es numpy/vectorización (clase 015).

¿Generador infinito (while True: yield ...) es buena idea?

Sí, pero ojo: nunca lo conviertas a lista directo (list(gen)) — bucle infinito hasta OOM. Acopla con itertools.islice(gen, N) para truncar.

yield from vs yield?

yield from sub_iterable delega: yieldea todos los valores del sub-iterable. Equivale a for x in sub: yield x pero más rápido y propaga send()/throw() correctamente. Útil para componer generadores.

Referencias

- Ramalho, Fluent Python 2e — caps. 2 y 17.
- PEP 202 — List Comprehensions
- PEP 255 — Simple Generators
- itertools docs

Siguiente clase

Clase 008 — Funciones: args, kwargs, lambdas, closures

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import sys, time, tracemalloc
from itertools import chain, islice, groupby, accumulate, takewhile
print('python:', sys.version.split()[0])
```

Archivos complementarios

- notebook.ipynb