UNIVERSIDAD CATOLICA
DE TEMUCO

Lab 2: Lambda, map, filter, reduce

Programación II

Desarrollado por:

Fernando Valdés

Joaquin Cantero

Profesor: Guido Mellado

Índice

1. Recordemos contenido

En Python, los generadores son una forma eficiente y elegante de crear iteradores (objetos que se pueden recorrer elemento por elemento, como en un bucle for) sin tener que almacenar todos los valores en memoria al mismo tiempo.

Ventaja	Explicación
Ahorro de memoria	No guarda todos los datos en memoria, los genera uno por uno.
Eficiencia	Ideal para manejar grandes volúmenes de datos o flujos infinitos.
Simplicidad	El código se ve limpio y claro comparado con clases iteradoras.

1.1. Funcion yield

El comando `yield` se usa dentro de una función para convertirla en un **generador**. En lugar de devolver todos los valores a la vez, va generando uno por uno cada vez que se le llama. Esto permite ahorrar memoria y manejar grandes cantidades de datos de forma eficiente.

1.1.1. Ejemplo 1: Generar cuadrados de números

- **Con yield**

```python
def contador(n):
    for i in range(n):
        yield i ** 2

for num in contador(5):
    print(num)
```

- **Sin yield**

```python
def contador(n):
    resultado = []
    for i in range(n):
        resultado.append(i ** 2)
    return resultado

for num in contador(5):
    print(num)
```

1.1.2. Ejemplo 2: Leer líneas de un archivo grande

- **Con yield**

```python
def leer_lineas(nombre_archivo):
    with open(nombre_archivo, 'r') as archivo:
        for linea in archivo:
            yield linea.strip()

for linea in leer_lineas('datos.txt'):
    print(linea)
```

- **Sin yield**

```python
    def leer_lineas(nombre_archivo):
    with open(nombre_archivo, 'r') as archivo:
        return [linea.strip() for linea in archivo]

for linea in leer_lineas('datos.txt'):
    print(linea)
```

1.2. Funcion iter()

En Python, **iter()** se utiliza para obtener un iterador a partir de un objeto iterable (como listas, tuplas, diccionarios, strings, etc.) o a partir de una función y un valor centinela.

- Sintaxis básica:

```python
    iter(n)
```

- Devuelve un iterador, que puedes recorrer con next() hasta que se agoten los elementos.
- Puntos importantes
  - Cualquier objeto que tenga el método **__iter__()** es iterable.
  - **next(iterador)** devuelve el siguiente elemento del iterador.
  - Cuando se acaban los elementos, **next()** lanza **StopIteration.**

1.2.1. Ejemplo 1: Contador simple

- **Con iter()**

```python
class Contador:
    def __init__(self, inicio, fin):
        self.actual = inicio
        self.fin = fin

    def __iter__(self):
        return self  # el propio objeto es el iterador

    def __next__(self):
        if self.actual <= self.fin:
            valor = self.actual
            self.actual += 1
            return valor
        else:
            raise StopIteration

for numero in Contador(3):
    print(numero)
```

- **Sin iter()**

```python
class Contador:
    def __init__(self, inicio, fin):
        self.inicio = inicio
        self.fin = fin

    def obtener_lista(self):
        resultado = []
        for i in range(self.inicio, self.fin + 1):
            resultado.append(i)
        return resultado

# Uso:
for numero in Contador(1, 5).obtener_lista():
    print(numero)
```

1.2.2. Ejemplo 2: Generador de cuadrados

- **Con iter()**

```
    class Cuadrados:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        for i in range(1, self.n + 1):
            yield i**2

for c in Cuadrados(5):
    print(c)
```

- **Sin iter()**

```
class Cuadrados:
    def __init__(self, n):
        self.n = n

    def obtener_lista(self):
        resultado = []
        for i in range(1, self.n + 1):
            resultado.append(i**2)
        return resultado

for c in Cuadrados(5).obtener_lista():
    print(c)
```

1.3. listas por comprensión

Las listas por comprensión (list comprehensions) son una forma compacta y elegante de crear listas en Python a partir de iterables, aplicando expresiones y filtros.

- Sintaxis básica:

```
 [expresion for elemento in iterable if condicion]
```

- **nueva_expresion** → lo que quieres que aparezca en la lista **__iter__()** es iterable.
- **for item in iterable** → iteras sobre los elementos
- **if condicion** → opcional, filtra elementos

1.3.1. Ejemplo 1: Números pares del 1 al 20

- **Con Comprension**

```python
pares = [i for i in range(1, 21) if i % 2 == 0]
print(pares)
```

- **Sin Comprension**

```python
pares = []
for i in range(1, 21):
    if i % 2 == 0:
        pares.append(i)
print(pares)
```

1.3.2. Ejemplo 2: Cuadrados de números impares del 1 al 10

- **Con Comprension**

```python
cuadrados_impares = [i**2 for i in range(1, 11) if i % 2 != 0]
print(cuadrados_impares)
```

- **Sin Comprension**

```python
cuadrados_impares = []
for i in range(1, 11):
    if i % 2 != 0:
        cuadrados_impares.append(i**2)
print(cuadrados_impares)
```

1.3.3. Ejemplo 3: Números del 1 al 10

- **Con Comprension**

```python
numeros = [i for i in range(1, 11)]
print(numeros)
```

- **Sin Comprension**

```python
numeros = []
for i in range(1, 11):
    numeros.append(i)
print(numeros)
```

1.4. Función lambda

En Python, **lambda** se utiliza para crear funciones anónimas, cortas y de una sola expresión, útiles para operaciones rápidas sin definir una función con **def**.

- Sintaxis básica:

```
lambda argumentos: expresion
```

1.4.1. Ejemplo 1: Elevar al cuadrado

- **Con lambda**

```
cuadrado = lambda x: x**2
print(cuadrado(5))  # Output: 25
```

- **Sin lambda**

```
def cuadrado(x):
    return x**2

print(cuadrado(5))  # Output: 25
```

1.4.2. Ejemplo 2: Sumar dos números

- **Con lambda**

```
sumar = lambda x, y: x + y
print(sumar(3, 7))  # Output: 10
```

- **Sin lambda**

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

print(sumar(3, 7))  # Output: 10
```

1.5. Función map()

En Python, **map()** aplica una función a todos los elementos de un iterable, devolviendo un objeto iterable.

- Sintaxis básica:

```
map(funcion, iterable)
```

1.5.1. Ejemplo 1: Elevar al cuadrado con map

- **Con map y lambda**

```
numeros = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)  # Output: [1, 4, 9, 16, 25]
```

- **Sin map**

```
numeros = [1, 2, 3, 4, 5]
cuadrados = []
for x in numeros:
    cuadrados.append(x**2)
print(cuadrados)  # Output: [1, 4, 9, 16, 25]
```

1.5.2. Ejemplo 2: Convertir una lista de grados Celsius a Fahrenheit

- **Con map y lambda**

```
celsius = [0, 10, 20, 30]
fahrenheit = list(map(lambda x: (x * 9/5) + 32, celsius))
print(fahrenheit)  # Output: [32.0, 50.0, 68.0, 86.0]
```

- **Sin map**

```
celsius = [0, 10, 20, 30]
fahrenheit = []
for x in celsius:
    fahrenheit.append((x * 9/5) + 32)
print(fahrenheit)  # Output: [32.0, 50.0, 68.0, 86.0]
```

1.6. Función filter()

En Python, **filter()** selecciona elementos de un iterable que cumplen una condición, devolviendo un objeto iterable.

- Sintaxis básica:

```
filter(funcion, iterable)
```

1.6.1. Ejemplo 1: Filtrar números pares

- **Con filter y lambda**

```
numeros = list(range(1, 11))
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # Output: [2, 4, 6, 8, 10]
```

- **Sin filter**

```
numeros = list(range(1, 11))
pares = []
for x in numeros:
    if x % 2 == 0:
        pares.append(x)
print(pares)  # Output: [2, 4, 6, 8, 10]
```

1.6.2. Ejemplo 2: Filtrar palabras con más de 4 letras

- **Con filter y lambda**

```
palabras = ["sol", "estrella", "luna", "planeta"]
largas = list(filter(lambda x: len(x) > 4, palabras))
print(largas)  # Output: ['estrella', 'planeta']
```

- **Sin filter**

```
palabras = ["sol", "estrella", "luna", "planeta"]
largas = []
for p in palabras:
    if len(p) > 4:
        largas.append(p)
print(largas)  # Output: ['estrella', 'planeta']
```

1.7. Función reduce()

En Python, **reduce()** aplica una función acumulativa a los elementos de un iterable, reduciéndolos a un único valor. Se debe importar desde **functools**.

- Sintaxis básica:

```
from functools import reduce
reduce(funcion, iterable)
```

1.7.1. Ejemplo 1: Sumar todos los elementos

- **Con reduce y lambda**

```
from functools import reduce
numeros = [1, 2, 3, 4, 5]
suma = reduce(lambda x, y: x + y, numeros)
print(suma)  # Output: 15
```

- **Sin reduce**

```
numeros = [1, 2, 3, 4, 5]
suma = 0
for x in numeros:
    suma += x
print(suma)  # Output: 15
```

1.7.2. Ejemplo 2: Multiplicar todos los elementos de una lista

- **Con reduce y lambda**

```
from functools import reduce

numeros = [1, 2, 3, 4, 5]
producto = reduce(lambda x, y: x * y, numeros)
print(producto)  # Output: 120
```

- **Sin reduce**

```
numeros = [1, 2, 3, 4, 5]
producto = 1
for x in numeros:
    producto *= x
print(producto)  # Output: 120
```

2. Actividad

2.1. Instrucciones Generales

A partir de los contenidos revisados en esta guía, se propone la realización de las siguientes actividades con el objetivo de consolidar los conceptos de programación funcional y orientada a objetos en Python.

Se recomienda abordar los ejercicios de manera secuencial, comprendiendo primero la teoría y posteriormente implementando cada solución en Python. Estas actividades permitirán:

- Aplicar iteradores y generadores para recorrer colecciones de datos eficientemente.
- Generar listas de manera concisa y elegante usando comprensión.
- Utilizar funciones lambda para definir operaciones rápidas y anónimas.
- Transformar y filtrar colecciones de datos usando map y filter.
- Aplicar reduce para combinar elementos de un iterable en un único resultado.

2.2. Evaluación del laboratorio

El laboratorio será evaluado considerando dos componentes principales:

1. **Calidad de los códigos (80 % de la nota final)**:
    - Se evaluará la correcta implementación de los ejercicios de cada comando: **yield**, **iter()**, **listas por comprensión**, **lambda**, **map**, **filter** y **reduce**.
    - Cada 4 ejercicios correctamente realizados y funcionales aportará 1 punto.
    - Se valorará el código limpio, bien indentado, legible y comentado adecuadamente.
2. **Entrega en GitHub (20 % de la nota final)**:
    - Cada estudiante deberá subir a un repositorio de GitHub una carpeta organizada por comando.
    - Cada carpeta deberá contener los 3 ejercicios correspondientes a dicho comando en archivos separados con extensión `.py`.

- El repositorio debe estar correctamente nombrado y accesible para la revisión por parte del docente.

2.3. Ejercicios con yield

1. Crear una función generadora que devuelva los primeros 10 números pares usando **yield** y recorrerla con un **for**.
2. Crear una función generadora que reciba una lista de números y devuelva solo los números impares usando **yield**.
3. Crear una clase con un método __iter__() que use **yield** para generar los cuadrados de los números del 1 al 10.
4. Crear una función generadora que genere la serie de Fibonacci hasta el décimo elemento usando **yield**.

2.4. Ejercicios con iter()

1. Crear un contador del 10 al 15 usando **iter()** y recorrerlo con **next()**.
2. Crear un generador de números impares del 1 al 20 usando **yield** en una clase e iterarlo con un **for**.
3. Crear una clase que genere los cuadrados de los números del 1 al 10 sin usar **iter()**, pero proporcionando un método que devuelva la lista completa.
4. Crear un iterador que recorra los elementos de una lista de cadenas y devuelva cada cadena en mayúsculas.

2.5. Ejercicios con listas por comprensión

1. Crear una lista de los números del 1 al 50.
2. Crear una lista con los cuadrados de todos los números pares del 1 al 20.
3. Crear una lista con las primeras letras de las palabras de la lista **["python", "java", Ç++", ruby"]**.
4. Crear una lista con los números divisibles por 3 del 1 al 30.

2.6. Ejercicios con lambda

1. Crear una función lambda que reciba dos números y devuelva el mayor.
2. Crear una función lambda que reciba una cadena y devuelva su longitud.
3. Crear una función lambda que reciba una lista y devuelva el primer elemento.
4. Crear una función lambda que reciba un número y devuelva su doble.

2.7. Ejercicios con map()

1. Dada la lista `[1,2,3,4,5]`, crear otra lista con cada elemento multiplicado por 10 usando **map()**.
2. Convertir la lista de grados Celsius `[0, 10, 20, 30]` a Fahrenheit usando **map()** y una función lambda.
3. Dada la lista `[üno","dos","tres"]`, crear otra lista con la longitud de cada palabra usando **map()**.
4. Dada la lista `[1,2,3,4,5]`, crear otra lista con el cuadrado de cada número usando **map()**.

2.8. Ejercicios con filter()

1. Filtrar los números pares de la lista `[1,2,3,4,5,6,7,8,9,10]` usando **filter()**.
2. Filtrar las palabras que empiezan con "p" de la lista `["perro","gato","pato","hamster"]`.
3. Filtrar los números mayores a 50 de la lista `[10,60,30,80,50,100]`.
4. Filtrar los números impares de la lista `[1,2,3,4,5,6,7,8,9,10]` usando **filter()**.

2.9. Ejercicios con reduce()

1. Sumar todos los elementos de la lista `[5,10,15,20]` usando **reduce()**.
2. Multiplicar todos los elementos de la lista `[2,3,4]` usando **reduce()**.
3. Encontrar el número mayor de la lista `[7,3,9,1,5]` usando **reduce()**.
4. Concatenar todas las cadenas de la lista `["Hola", , "Mundo", ¡"]` usando **reduce()**.