RESUMEN DE ESTUDIO

Clase 10: Programación Funcional

Lambda, Map, Filter, Reduce y Zip

Curso de Python - Material de Referencia

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN A LA PROGRAMACIÓN FUNCIONAL

La programación funcional es un paradigma de programación que trata la computación como la evaluación de funciones matemáticas. A diferencia de la programación imperativa (que usamos con for loops), la programación funcional se enfoca en describir QUÉ queremos obtener, no CÓMO hacerlo paso a paso.

Ventajas de la Programación Funcional:

- **Código más conciso:** Menos líneas de código para lograr el mismo resultado
- **Menos errores:** Al no mutar variables, hay menos efectos secundarios
- **Más fácil de probar:** Funciones puras siempre dan el mismo resultado
- **Mejor para procesar datos:** Ideal para transformar colecciones
- **Código más legible:** Una vez que dominas la sintaxis

Comparación: Imperativo vs Funcional

Imperativo (for loop):

```
numeros = [1, 2, 3, 4, 5] resultado = [] for n in numeros: resultado.append(n * 2) #
resultado = [2, 4, 6, 8, 10]
```

Funcional (map):

```
numeros = [1, 2, 3, 4, 5] resultado = list(map(lambda x: x * 2, numeros)) # resultado = [2,
4, 6, 8, 10]
```

2. FUNCIONES LAMBDA

Una función lambda es una función anónima (sin nombre) que se define en una sola línea. Son ideales para operaciones simples que se usan una sola vez.

Sintaxis:

```
lambda argumentos: expresión
```

Ejemplos de Lambda:

```
# Lambda con un argumento cuadrado = lambda x: x ** 2 print(cuadrado(4)) # 16 # Lambda con múltiples argumentos suma = lambda a, b: a + b print(suma(3, 5)) # 8 # Lambda con condicional mayor = lambda a, b: a if a > b else b print(mayor(10, 5)) # 10 # Lambda para verificar condición es_par = lambda n: n % 2 == 0 print(es_par(7)) # False # Lambda para calcular porcentaje porcentaje = lambda valor, pct: valor * pct / 100 print(porcentaje(200, 15)) # 30.0
```

Características de Lambda:

- Solo puede contener UNA expresión
- No necesita la palabra return (está implícito)
- Puede recibir cualquier número de argumentos
- Se puede asignar a una variable
- Ideal para usar como argumento de otras funciones

3. LA FUNCIÓN MAP()

map() aplica una función a CADA elemento de un iterable (lista, tupla, etc.) y devuelve un nuevo iterable con los resultados transformados.

Sintaxis:

```
map(función, iterable)
```

Ejemplos:

```
# Duplicar cada número numeros = [1, 2, 3, 4, 5] duplicados = list(map(lambda x: x * 2, numeros)) print(duplicados) # [2, 4, 6, 8, 10] # Convertir a mayúsculas palabras = ["hola", "mundo", "python"] mayusculas = list(map(lambda p: p.upper(), palabras)) print(mayusculas) # ["HOLA", "MUNDO", "PYTHON"] # Con función definida def cuadrado(x): return x ** 2 cuadrados = list(map(cuadrado, [1, 2, 3, 4])) print(cuadrados) # [1, 4, 9, 16]
```

Map con Múltiples Iterables:

```
# Sumar dos listas elemento por elemento lista1 = [1, 2, 3, 4] lista2 = [10, 20, 30, 40] sumas = list(map(lambda a, b: a + b, lista1, lista2)) print(sumas) # [11, 22, 33, 44] # Calcular promedio de tres listas notas1 = [85, 90, 78] notas2 = [88, 92, 80] notas3 = [90, 85, 82] promedios = list(map(lambda a, b, c: (a + b + c) / 3, notas1, notas2, notas3)) print(promedios) # [87.67, 89.0, 80.0]
```

IMPORTANTE: map() retorna un objeto map, NO una lista. Debes usar list() para convertir el resultado a lista.

4. LA FUNCIÓN FILTER()

filter() selecciona solo los elementos de un iterable que cumplen una condición (es decir, para los cuales la función retorna True).

Sintaxis:

```
filter(función_condición, iterable)
```

Ejemplos:

```
# Filtrar números pares numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] pares =
list(filter(lambda x: x % 2 == 0, numeros)) print(pares) # [2, 4, 6, 8, 10] # Filtrar
palabras largas (más de 5 letras) palabras = ["hola", "programación", "python", "sol",
"desarrollo"] largas = list(filter(lambda p: len(p) > 5, palabras)) print(largas) #
["programación", "python", "desarrollo"] # Filtrar productos caros productos = [ {"nombre":
"Laptop", "precio": 800}, {"nombre": "Mouse", "precio": 25}, {"nombre": "Monitor",
"precio": 300} ] caros = list(filter(lambda p: p["precio"] > 100, productos)) print(caros)
# [{"nombre": "Laptop"...}, {"nombre": "Monitor"...}] # Filtrar números positivos numeros =
[-5, 3, -2, 8, -1, 0, 7] positivos = list(filter(lambda x: x > 0, numeros))
print(positivos) # [3, 8, 7]
```

IMPORTANTE: La función que se pasa a filter() debe retornar True o False. También retorna un objeto filter que debe convertirse a list().

5. LA FUNCIÓN REDUCE()

reduce() aplica una función de forma acumulativa a los elementos de un iterable, reduciéndolos a un solo valor. Necesita importarse desde functools.

Sintaxis:

```
from functools import reduce reduce(función, iterable) reduce(función, iterable, valor_inicial)
```

Cómo funciona reduce:

La función recibe dos argumentos: el acumulador (resultado parcial) y el elemento actual. Procesa la lista de izquierda a derecha.

```
# Ejemplo paso a paso: sumar [1, 2, 3, 4, 5] # reduce(lambda acc, x: acc + x, [1, 2, 3, 4, 5]) # Paso 1: acc=1, x=2 -> 1+2=3 # Paso 2: acc=3, x=3 -> 3+3=6 # Paso 3: acc=6, x=4 -> 6+4=10 # Paso 4: acc=10, x=5 -> 10+5=15 # Resultado: 15
```

Ejemplos:

```
from functools import reduce # Sumar todos los números numeros = [1, 2, 3, 4, 5] suma = reduce(lambda acc, x: acc + x, numeros) print(suma) # 15 # Multiplicar todos producto = reduce(lambda acc, x: acc * x, numeros) print(producto) # 120 # Encontrar el máximo maximo = reduce(lambda a, b: a if a > b else b, [45, 12, 89, 34]) print(maximo) # 89 # Con valor inicial suma_con_inicial = reduce(lambda a, b: a + b, [1, 2, 3], 100) print(suma_con_inicial) # 106 (100+1+2+3) # Concatenar strings palabras = ["Hola", "mundo", "Python"] frase = reduce(lambda a, b: a + " " + b, palabras) print(frase) # "Hola mundo Python"
```

IMPORTANTE: reduce() debe importarse: from functools import reduce

6. LA FUNCIÓN ZIP()

zip() combina múltiples iterables en uno solo, emparejando elementos por posición. Devuelve un iterador de tuplas.

Sintaxis:

```
zip(iterable1, iterable2, ...)
```

Ejemplos:

```
# Combinar dos listas nombres = ["Ana", "Luis", "María"] edades = [25, 30, 28] combinados =
list(zip(nombres, edades)) print(combinados) # [("Ana", 25), ("Luis", 30), ("María", 28)] #
Iterar sobre pares for nombre, edad in zip(nombres, edades): print(f"{nombre} tiene {edad}
años") # Crear diccionario con zip claves = ["nombre", "edad", "ciudad"] valores = ["Ana",
25, "Madrid"] diccionario = dict(zip(claves, valores)) print(diccionario) # {"nombre":
"Ana", "edad": 25, "ciudad": "Madrid"} # Zip con múltiples listas notas1 = [85, 90, 78]
notas2 = [88, 92, 80] notas3 = [90, 85, 82] for n, p1, p2, p3 in zip(nombres, notas1,
notas2, notas3): promedio = (p1 + p2 + p3) / 3 print(f"{n}: {promedio:.1f}")
```

IMPORTANTE: Si las listas tienen diferente longitud, zip() se detiene en la más corta.

7. COMBINANDO FUNCIONES

El verdadero poder de la programación funcional está en encadenar operaciones como un pipeline de procesamiento de datos.

Pipeline: filter → map → reduce

```
from functools import reduce numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # Objetivo: Filtrar pares → Elevar al cuadrado → Sumar todos # Paso 1: Filtrar números pares pares = filter(lambda x: x % 2 == 0, numeros) # [2, 4, 6, 8, 10] # Paso 2: Elevar al cuadrado cuadrados = map(lambda x: x ** 2, pares) # [4, 16, 36, 64, 100] # Paso 3: Sumar todos total = reduce(lambda a, b: a + b, cuadrados) print(total) # 220
```

Ejemplo con diccionarios:

```
productos = [ {"nombre": "A", "precio": 100, "stock": 5}, {"nombre": "B", "precio": 50, "stock": 0}, {"nombre": "C", "precio": 200, "stock": 3} ] # Pipeline: Filtrar con stock > 0 → Calcular valor total → Sumar con_stock = filter(lambda p: p["stock"] > 0, productos) valores = map(lambda p: p["precio"] * p["stock"], con_stock) total = reduce(lambda a, b: a + b, valores) print(total) # 1100 (100*5 + 200*3)
```

8. TABLA COMPARATIVA: LAMBDA VS DEF

Característica	Lambda	def
Nombre	Anónima (sin nombre)	Tiene nombre
Líneas	Una sola línea	Múltiples líneas
Return	Implícito	Explícito (return)
Docstring	No soporta	Soporta docstrings
Complejidad	Simple (una expresión)	Puede ser compleja
Reutilización	Generalmente una vez	Muchas veces
Uso típico	Argumento de map/filter	Código reutilizable
Legibilidad	Concisa	Más explícita

9. TABLA COMPARATIVA: MAP VS FILTER VS REDUCE

Función	¿Qué hace?	Entrada	Salida	Cuándo usar
map()	Transforma cada elemento	N elementos	N elementos	Convertir datos
filter()	Selecciona elementos	N elementos	0 a N elementos	Filtrar por condición
reduce()	Acumula a un valor	N elementos	1 valor	Calcular totales
zip()	Combina iterables	Múltiples listas	Lista de tuplas	Unir datos paralelos

10. ENUMERATE() VS ZIP()

Función	Uso	Ejemplo
enumerate()	Cuando necesitas índice + UNA lista	`for i, x in enumerate(lista):`
zip()	Cuando necesitas combinar 2+ listas	`for a, b in zip(lista1, lista2):`

Combinando ambos:

```
nombres = ["Ana", "Luis", "María"] edades = [25, 30, 28] # Combinar enumerate + zip for i, (nombre, edad) in enumerate(zip(nombres, edades)): print(f"{i}: {nombre} tiene {edad} años") # Salida: # 0: Ana tiene 25 años # 1: Luis tiene 30 años # 2: María tiene 28 años
```

11. CUÁNDO USAR PROGRAMACIÓN FUNCIONAL

USA programación funcional cuando:

- Transformas datos simples (multiplicar, convertir, etc.)
- Filtras por condiciones claras
- Reduces a un valor único (suma, máximo, etc.)
- El código queda más legible que con for
- Procesas colecciones de datos

USA loops tradicionales cuando:

- La lógica es muy compleja
- Necesitas múltiples operaciones por elemento
- El código funcional es difícil de leer
- Necesitas break o continue
- Encadenas demasiadas operaciones (más de 3)

Regla de oro: Si tienes que explicar tu código funcional durante 5 minutos, probablemente un for loop sea mejor.

12. ERRORES COMUNES

Error 1: Olvidar convertir a lista

```
# MAL resultado = map(lambda x: x*2, [1,2,3]) print(resultado) # # BIEN resultado = list(map(lambda x: x*2, [1,2,3])) print(resultado) # [2, 4, 6]
```

Error 2: Olvidar importar reduce

```
# MAL reduce(lambda a, b: a+b, [1,2,3]) # NameError: name 'reduce' is not defined # BIEN from functools import reduce reduce(lambda a, b: a+b, [1,2,3]) # 6
```

Error 3: Lambda con múltiples líneas

```
# MAL - Lambda NO puede tener múltiples líneas # cuadrado = lambda x: # resultado = x ** 2 # return resultado # BIEN - Usar def para funciones complejas def cuadrado(x): resultado = x ** 2 return resultado
```

Error 4: Consumir el iterador dos veces

```
# MAL - Los iteradores de map/filter se consumen al usarlos resultado = map(lambda x: x*2, [1,2,3]) print(list(resultado)) # [2, 4, 6] print(list(resultado)) # [] - Ya se consumió! # BIEN - Convertir a lista primero resultado = list(map(lambda x: x*2, [1,2,3])) print(resultado) # [2, 4, 6] print(resultado) # [2, 4, 6] - Funciona
```

13. EJERCICIOS ADICIONALES

NIVEL PRINCIPIANTE

Ejercicio P1: Lambda para Celsius a Fahrenheit

Crea una función lambda que convierta una temperatura de Celsius a Fahrenheit. Fórmula: F = C × 9/5 + 32. Pruébala con: 0, 20, 100.

Ejercicio P2: Map para elevar al cubo

Dada la lista [1, 2, 3, 4, 5], usa map y lambda para crear una nueva lista con cada número elevado al cubo.

Ejercicio P3: Filter para números mayores

Dada la lista [12, 5, 8, 15, 3, 20, 7], usa filter y lambda para obtener solo los números mayores a 10.

Ejercicio P4: Map para extraer longitudes

Dada la lista ['python', 'java', 'javascript', 'c++'], usa map para crear una lista con la longitud de cada palabra.

Ejercicio P5: Filter para strings que empiezan con vocal

Dada la lista ['apple', 'banana', 'orange', 'mango', 'avocado'], usa filter para obtener solo las frutas que empiezan con vocal.

NIVEL INTERMEDIO

Ejercicio I1: Combinar map y filter

Dada la lista [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], primero filtra los números impares, luego eleva al cuadrado. Resultado esperado: [1, 9, 25, 49, 81].

Ejercicio I2: Reduce para encontrar el mínimo

Usa reduce (sin usar la función min()) para encontrar el valor mínimo de la lista [45, 12, 89, 34, 67, 5, 23].

Ejercicio I3: Zip para crear diccionarios

Dadas las listas nombres=['Ana','Luis','María'] y ciudades=['Madrid','Barcelona','Valencia'], usa zip para crear una lista de diccionarios con la estructura: {'nombre': 'Ana', 'ciudad': 'Madrid'}.

Ejercicio I4: Pipeline de procesamiento

Dada la lista de productos: productos = [{"nombre": "A", "precio": 100, "cantidad": 5}, {"nombre": "B", "precio": 50, "cantidad": 0}, {"nombre": "C", "precio": 200, "cantidad": 3}, {"nombre": "D", "precio": 75, "cantidad": 2}] Usa filter, map y reduce para: 1) Filtrar productos con cantidad > 0, 2) Calcular el valor total de cada producto (precio × cantidad), 3) Sumar todos los valores totales.

Ejercicio I5: Procesamiento de calificaciones

Dadas tres listas: nombres=['Ana','Luis','María','Carlos'], parcial1=[85,60,92,45], parcial2=[88,55,95,50]. Usa zip, map y filter para: 1) Combinar los datos 2) Calcular el promedio de cada estudiante 3) Obtener solo los estudiantes con promedio >= 70

NIVEL AVANZADO

Ejercicio A1: Analizador de texto funcional

Dado el texto: "Python es un lenguaje de programación muy popular Python" Usando solo funciones de programación funcional (map, filter, reduce), crea un programa que: 1) Divida el texto en palabras (puedes usar split()) 2) Filtre palabras con más de 3 letras 3) Convierta a minúsculas 4) Cuente cuántas veces aparece cada palabra (usa reduce con un dict como acumulador) Resultado esperado: {'python': 2, 'lenguaje': 1, 'programación': 1, 'popular': 1}

Ejercicio A2: Sistema de descuentos

Tienes una lista de compras: compras = [{"cliente": "Ana", "productos": [100, 200, 50], "vip": True}, {"cliente": "Luis", "productos": [80, 120], "vip": False}, {"cliente": "María", "productos": [300, 150, 75], "vip": True}] Crea funciones que usando programación funcional: 1) calcular_total(compra) - suma todos los productos de una compra 2) aplicar_descuento(compras) - VIP tiene 20% descuento, no-VIP tiene 10% 3) total_ventas(compras) - suma total de todas las compras con descuento aplicado No uses loops for/while explícitos.

Ejercicio A3: Procesador de logs

Tienes una lista de logs: logs = [{"timestamp": "2024-01-15 10:30", "level": "ERROR", "message": "Connection failed"}, {"timestamp": "2024-01-15 10:31", "level": "INFO", "message": "Retry attempt"}, {"timestamp": "2024-01-15 10:32", "level": "ERROR", "message": "Timeout error"}, {"timestamp": "2024-01-15 10:33", "level": "WARNING", "message": "Low memory"}, {"timestamp": "2024-01-15 10:34", "level": "INFO", "message": "Connected successfully"}] Usando solo programación funcional: 1) Filtra solo los logs de tipo "ERROR" 2) Extrae solo los mensajes de error 3) Concatena todos los mensajes en un string separado por " | " 4) Cuenta cuántos logs hay de cada nivel (usa reduce para crear un dict contador)

14. SOLUCIONES DE EJERCICIOS

SOLUCIONES NIVEL PRINCIPIANTE

Solución P1:

```
celsius_a_fahrenheit = lambda c: c * 9/5 + 32 print(celsius_a_fahrenheit(0)) # 32.0
print(celsius_a_fahrenheit(20)) # 68.0 print(celsius_a_fahrenheit(100)) # 212.0
```

Solución P2:

```
numeros = [1, 2, 3, 4, 5] cubos = list(map(lambda x: x ** 3, numeros)) print(cubos) # [1,
8, 27, 64, 125]
```

Solución P3:

```
numeros = [12, 5, 8, 15, 3, 20, 7] mayores = list(filter(lambda x: x > 10, numeros))
print(mayores) # [12, 15, 20]
```

Solución P4:

```
lenguajes = ['python', 'java', 'javascript', 'c++'] longitudes = list(map(lambda x: len(x),
lenguajes)) print(longitudes) # [6, 4, 10, 3]
```

Solución P5:

```
frutas = ['apple', 'banana', 'orange', 'mango', 'avocado'] vocales = 'aeiou' con_vocal =
list(filter(lambda f: f[0].lower() in vocales, frutas)) print(con_vocal) # ['apple',
'orange', 'avocado']
```

SOLUCIONES NIVEL INTERMEDIO

Solución I1:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] impares = filter(lambda x: x % 2 != 0, numeros) cuadrados = list(map(lambda x: x ** 2, impares)) print(cuadrados) # [1, 9, 25, 49, 81]
```

Solución I2:

```
from functools import reduce numeros = [45, 12, 89, 34, 67, 5, 23] minimo = reduce(lambda a, b: a if a < b else b, numeros) print(minimo) # 5
```

Solución I3:

```
nombres = ['Ana', 'Luis', 'María'] ciudades = ['Madrid', 'Barcelona', 'Valencia'] resultado = list(map( lambda nc: {'nombre': nc[0], 'ciudad': nc[1]}, zip(nombres, ciudades) )) print(resultado) # [{'nombre': 'Ana', 'ciudad': 'Madrid'}, # {'nombre': 'Luis', 'ciudad': 'Barcelona'}, # {'nombre': 'María', 'ciudad': 'Valencia'}]
```

Solución I4:

```
from functools import reduce productos = [ {"nombre": "A", "precio": 100, "cantidad": 5}, {"nombre": "B", "precio": 50, "cantidad": 0}, {"nombre": "C", "precio": 200, "cantidad": 3}, {"nombre": "D", "precio": 75, "cantidad": 2} ] # Pipeline con_stock = filter(lambda p: p["cantidad"] > 0, productos) valores = map(lambda p: p["precio"] * p["cantidad"], con_stock) total = reduce(lambda a, b: a + b, valores) print(total) # 1250 (500 + 600 + 150)
```

Solución I5:

```
nombres = ['Ana', 'Luis', 'María', 'Carlos'] parcial1 = [85, 60, 92, 45] parcial2 = [88, 55, 95, 50] # Combinar con zip datos = list(zip(nombres, parcial1, parcial2)) # Calcular promedios con map con_promedio = list(map( lambda d: (d[0], (d[1] + d[2]) / 2), datos )) # Filtrar aprobados aprobados = list(filter( lambda e: e[1] >= 70, con_promedio )) print(aprobados) # [('Ana', 86.5), ('María', 93.5)]
```

SOLUCIONES NIVEL AVANZADO

Solución A1:

```
from functools import reduce texto = "Python es un lenguaje de programación muy popular Python" # 1. Dividir en palabras palabras = texto.split() # 2. Filtrar palabras con más de 3 letras largas = filter(lambda p: len(p) > 3, palabras) # 3. Convertir a minúsculas minusculas = list(map(lambda p: p.lower(), largas)) # 4. Contar ocurrencias con reduce def contador(acc, palabra): acc[palabra] = acc.get(palabra, 0) + 1 return acc conteo = reduce(contador, minusculas, {}) print(conteo) # {'python': 2, 'lenguaje': 1, 'programación': 1, 'popular': 1}
```

Solución A2:

```
from functools import reduce compras = [ {"cliente": "Ana", "productos": [100, 200, 50], "vip": True}, {"cliente": "Luis", "productos": [80, 120], "vip": False}, {"cliente": "María", "productos": [300, 150, 75], "vip": True} ] # 1. Calcular total de una compra calcular_total = lambda compra: reduce(lambda a, b: a + b, compra["productos"]) # 2. Aplicar descuento def aplicar_descuento(compras): def descuento(compra): total = calcular_total(compra) descuento = 0.20 if compra["vip"] else 0.10 return { **compra, "total": total, "total_con_descuento": total * (1 - descuento) } return list(map(descuento, compras)) # 3. Total de ventas def total_ventas(compras): con_descuento = aplicar_descuento(compras) return reduce(lambda a, c: a + c["total_con_descuento"], con_descuento, 0) print(total_ventas(compras)) # 880.0
```

Solución A3:

```
from functools import reduce logs = [ {"timestamp": "2024-01-15 10:30", "level": "ERROR", "message": "Connection failed"}, {"timestamp": "2024-01-15 10:31", "level": "INFO", "message": "Retry attempt"}, {"timestamp": "2024-01-15 10:32", "level": "ERROR", "message": "Timeout error"}, {"timestamp": "2024-01-15 10:33", "level": "WARNING", "message": "Low memory"}, {"timestamp": "2024-01-15 10:34", "level": "INFO", "message": "Connected successfully"} ] # 1. Filtrar solo errores errores = filter(lambda l: l["level"] == "ERROR", logs) # 2. Extraer mensajes mensajes = list(map(lambda l: l["message"], errores)) print(mensajes) # ['Connection failed', 'Timeout error'] # 3. Concatenar concat = reduce(lambda a, b: a + " | " + b, mensajes) print(concat) # "Connection failed | Timeout error" # 4. Contar por nivel def contar_niveles(acc, log): nivel = log["level"] acc[nivel] = acc.get(nivel, 0) + 1 return acc conteo = reduce(contar_niveles, logs, {}) print(conteo) # {'ERROR': 2, 'INFO': 2, 'WARNING': 1}
```

FIN DEL RESUMEN DE ESTUDIO

Clase 10: Programación Funcional

¡Practica los ejercicios para dominar estos conceptos!

Próxima clase: List y Dict Comprehensions