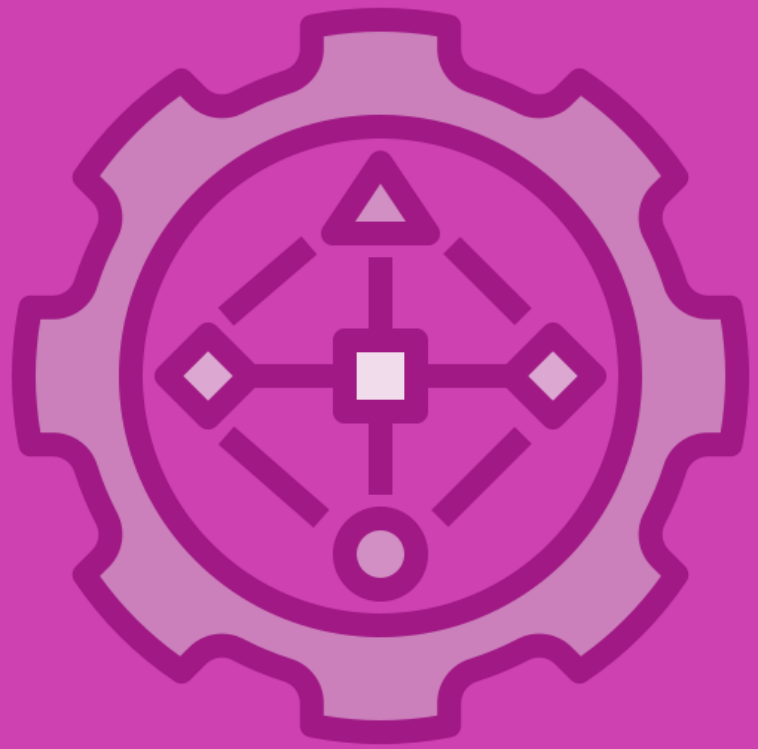
LAMBDA, MAP, FILTER Y REDUCE

Algoritmos y estructuras de datos I

Lic. Yaps David P.



FUNCIONES LAMBDA

Se refiere a una pequeña función anónima, ya que (técnicamente) carecen de nombre.

A diferencia de una función normal, no las definimos con la palabra clave *def* para crearlas. En su lugar, se definen en una sola línea que ejecuta una sola expresión. Este tipo de funciones pueden tomar cualquier número de argumentos, pero solo pueden tener una expresión que es evaluada y devuelta:

argumentos

```
lambda p1, p2, .., pn : expresión
```

Toda función lambda se puede convertir a una función normal, pero no viceversa.

EJEMPLO DE USO

Si deconstruimos una función sencilla, podemos llegar a una función lambda. Por ejemplo, podemos usar la siguiente función para obtener el doble de un valor:

```
def doble(numero):
    return numero*2
```

Esta misma podemos simplificar el código aún más. Incluso, en una sola línea:

```
def doble(numero): return numero*2
```

Esta notación simple es la que una función lambda intenta replicar. Vamos a convertir la función en una función descripta en una anónima:

```
lambda numero : numero*2
```

Por lo tanto, ya tenemos una función anónima con un solo argumento (*numero*) y el retorno de su doble (*numero * 2*):

```
lambda numero : numero*2
```

Lo único que necesitamos hacer para utilizarla es guardarla en una variable y utilizarla tal como haríamos con una función normal:

```
doble = lambda numero : numero*2

print(doble(11)) #22
doble33 = doble(33) #66
```

EJEMPLOS DE USO

Gracias a la flexibilidad del lenguaje, podemos implementar infinitas funciones simples. A continuación, podemos analizar algunos ejemplos.

Comprobar si un número es impar:

```
impar = lambda numero : numero%2 != 0
```

Invertir una cadena utilizando rebanado:

```
invertir = lambda cadena : cadena[::-1]
```

Podemos utilizar varios argumentos, por ejemplo, para sumar dos números:

```
sumar = lambda numero1, numero2 : numero1+numero2
```

Calcular la potencia de un número:

```
potencia = lambda base, exponente : base**exponente
```

Formato de moneda:

```
formatoMoneda = lambda valor : f"${valor:,.2f}"
```

Formato porcentual:

```
formatoPorcentual = lambda valor : f"{valor:.2f}%"
```

EJEMPLOS DE USO

Las funciones lambda en Python se pueden invocar inmediatamente después de su definición utilizando paréntesis para pasar argumentos.

Esto se conoce como una función lambda *inline* y se puede hacer porque en Python, las funciones lambda se pueden definir y llamar en cualquier lugar donde se pueda usar una expresión.

```
#Suma de tres números
resultado = (lambda x, y, z: x + y + z)(1, 2, 3)
print(f"Resultado: {resultado}")  #6

#Concatenación de cadenas
resultado = (lambda a, b: a + b)("Hola, ", "mundo!")
print(f"Resultado: {resultado}")  #Hola, mundo!

#Comparación de dos números
resultado = (lambda x, y: max(x, y))(7, 3)
print(f"Resultado: {resultado}") #7

#Elevar un número a otro número
resultado = (lambda base, exp: base ** exp)(2, 3)
print(f"Resultado: {resultado}")  #8
```

FUNCIONES LAMBDA CON CLÁUSULA ELSE

Es posible utilizar la cláusula *else* en una expresión *lambda*, siempre que incluya una expresión condicional (*if*), conocida como expresión ternaria. La sintaxis sería la siguiente:

argumentos

lambda *p1, .., pn* : *{valor condición verdadera}* if condición else *{valor condición falsa}*

Por ejemplo, la siguiente expresión lambda devuelve el cuadrado de un número si es positivo, y devuelve *None* si el número es negativo:

```
cuadradoSiEsPositivo = lambda numero : numero**2 if numero > 0 else None
print(cuadradoSiEsPositivo(3)) #9
print(cuadradoSiEsPositivo(-1)) #None
```

Cabe destacar que, aunque se puede utilizar *else* en una expresión *lambda*, si la lógica es demasiado compleja, puede ser más legible utilizar una función regular en su lugar.

CONCLUSIONES DEL USO DE LAMBDA

Al igual que ocurre al utilizar compresión de listas, lo que hemos logrado es escribir nuestro código en una sola línea y quitar la sintaxis innecesaria.

Siguiendo el ejemplo de la parte inferior, en lugar de usar *def* para definir nuestra función, hemos utilizado la palabra clave lambda; a continuación, escribimos *numero* como argumentos de la función, y *numero**0.5* como expresión. Además, se omite la palabra clave *return*, condensando aún más la sintaxis.

Por último, y aunque la definición es anónima, la almacenamos en la variable *raizCuadrada* para poder llamarla desde cualquier parte del código, de no ser así tan solo podríamos hacer uso de ella en la línea donde la definamos.

```
def raizCuadrada(numero):
    return numero**0.5


print(raizCuadrada(4))
print(raizCuadrada(41))
resultado = raizCuadrada(33)
print(resultado)
```

```
raizCuadrada = lambda numero: numero**0.5

print(raizCuadrada(4))
print(raizCuadrada(41))
resultado = raizCuadrada(33)
print(resultado)
```

¿CUÁNDO USAR LAMBDA EN LUGAR DE DEF?

Las funciones lambda son soluciones rápidas, algo así como usar una navaja suiza para una pequeña tarea. Son perfectas cuando necesitamos algo rápido y simple, pero si el problema se vuelve más complicado o vamos a reutilizar la herramienta, es mejor usar una herramienta dedicada, es decir, una función *def*.

Utilizaremos funciones lambda para tareas pequeñas y sencillas, especialmente cuando la función será de corta duración o se usará solo una vez o en pocas ocasiones.

👍 Concisión

Las funciones lambda son breves y pueden hacer que el código sea más legible al evitar definiciones largas.

👍 Uso en línea

Pueden definirse y utilizarse en el mismo lugar donde se necesitan, sin necesidad de hacer una definición formal previa.

👎 Simplicidad

Solo pueden contener una expresión, por lo que no son adecuadas para tareas más grandes o complejas.

👎 Legibilidad

Si se usan en exceso o se complican demasiado, pueden hacer que el código sea más difícil de entender.

USO DE FUNCIONES COMO PARÁMETROS

Son funciones que pueden recibir otras funciones como argumentos. Este enfoque es muy útil cuando necesitas aplicar lógica variable dentro de otra función.

Si tenemos una tienda en línea y queremos aplicar diferentes tipos de descuentos a tus productos, podemos crear una función genérica que reciba una función de descuento como parámetro y la aplique al precio de un producto.

Este patrón es muy flexible y escalable. Si en el futuro necesitamos agregar más tipos de descuentos, simplemente creamos una nueva función de descuento sin necesidad de modificar la lógica de *aplicarDescuento*.

```
def aplicarDescuento(funcionDescuento, precio):
    return funcionDescuento(precio)

def descuentoTemporada(precio):
    return precio * 0.90  # 10% de descuento

def descuentoClienteFrecuente(precio):
    return precio * 0.85  # 15% de descuento


precio = 100

print(f"Precio original: ${precio}")
print("Precio con descuento de temporada:")
print(f"${aplicarDescuento(descuentoTemporada, precio)}")
print(f"Precio con descuento por cliente frecuente:")
print(f"${aplicarDescuento(descuentoClienteFrecuente, precio)}")
```

```
enteros = [1, 2, 4, 7]
cuadrados = []

for numero in enteros:
    cuadrados.append(numero ** 2)

print(cuadrados) #[1, 4, 16, 49]
```

FUNCIÓN MAP

Por medio de la función *map()* podemos aplicar una función a cada uno de los elementos de un iterable y obtener un lote de datos ya transformados:

map (*función, iterable*)

Si tenemos una lista de enteros y queremos obtener una nueva lista con el cuadrado de cada uno de ellos. Seguramente, una forma de resolverlo como venimos trabajando hasta el momento será similar a la del ejemplo.

FUNCIÓN MAP

Sin embargo, podemos usar una función lambda en combinación con *map()* para obtener el mismo resultado de una manera mucho más simple.

Se debe tener en cuenta que el resultado de uso de la función *map()* retorna un objeto del tipo *map*. Para que podamos trabajar con dicho resultado, es que lo combinamos con el uso de la función de conversión para el tipo de datos original, en caso de que así se requiera.

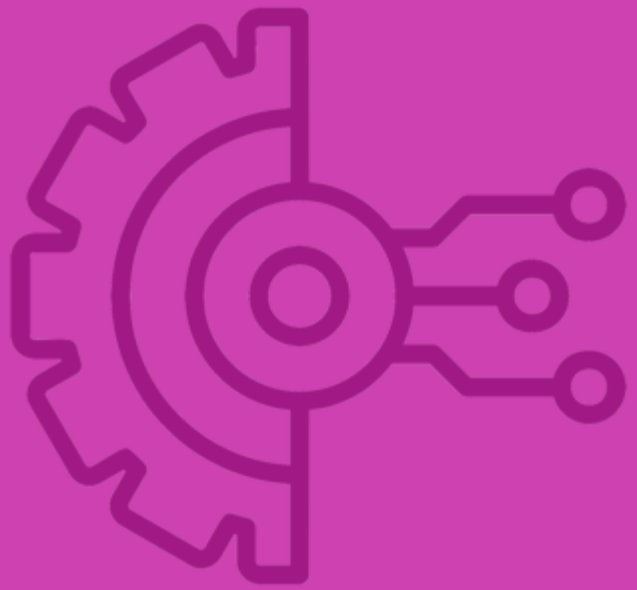
```
#Usando map con una lista
enteros = [1, 2, 4, 7]
resultado = list(map(lambda numero : numero ** 2, enteros))
print(resultado) #[1, 4, 16, 49]

#Usando map con una tupla
enteros = (1, 2, 4, 7)
resultado = tuple(map(lambda numero : numero ** 2, enteros))
print(resultado) #(1, 4, 16, 49)

#Usando map con un conjunto
enteros = {1, 2, 4, 7}
resultado = set(map(lambda numero : numero ** 2, enteros))
print(resultado)  #{16, 1, 4, 49}

#Usando map con un rango
rango = (numero for numero in range(0, 6))
resultado = list(map(lambda numero : numero ** 2, rango))
print(resultado)  #[0, 1, 4, 9, 16, 25]
```

Lic. Yaps David P.



FUNCIÓN MAP

A continuación, analizaremos el uso de *map* con una función que hayamos creado anteriormente:

```
def elevarCuadrado(numero):
    return numero**2

numeros = [1, 3, 5, 8]

resultado = list(map(elevarCuadrado, numeros))
print(resultado) #[1, 9, 25, 64]
```

La asignación de la variable resultado con *map* sería equivalente al siguiente código donde se utiliza una iteración para completar la lista:

```
resultado = []
for numero in numeros:
    resultado.append(elevarCuadrado(numero))

print(resultado) #[1, 9, 25, 64]
```

FUNCIÓN FILTER

La función *filter()* retorna un lote de elementos que cumplen una determinada condición (*True*) en base a la utilización de una función para tal fin.

filter (*función, iterable*)

Supongamos que queremos obtener de una lista de números solo los valores pares. Una alternativa podría ser la siguiente.

```
valores = [1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = []

for valor in valores:
    if valor % 2 == 0:
        pares.append(valor)

print(pares) #[2, 4, 6, 8]
```

```python
#Usando filter con una lista
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
resultado = list(filter(lambda numero : numero % 2 == 0, lista))
print(resultado) #[2, 4, 6, 8]

#Usando filter con una tupla
tupla = (1, 2, 3, 4, 5, 6, 7, 8, 9)
resultado = tuple(filter(lambda numero : numero % 2 == 0, tupla))
print(resultado) #(2, 4, 6, 8)

#Usando filter con un conjunto
conjunto = {1, 2, 3, 4, 5, 6, 7, 8, 9}
resultado = set(filter(lambda numero : numero % 2 == 0, conjunto))
print(resultado)  #{8, 2, 4, 6}

#Usando filter con un rango
rango = (numero for numero in range(1, 10))
resultado = list(filter(lambda numero : numero % 2 == 0, rango))
print(resultado)  #[2, 4, 6, 8]
```

FUNCIÓN FILTER CON LAMBDA

Sin embargo, podemos hacer uso de la función *filter()* y una función *lambda* para obtener el mismo resultado con una sola línea de código.

Debemos tener en cuenta, que el retorno de la función *filter()* es un objeto *filter*. Para que lo podamos utilizar, es necesario convertirlo en el tipo de datos conveniente para la lógica que estemos aplicando.

FUNCIÓN FILTER

No necesariamente debemos hacer uso de una función *lambda*, sino que también podemos hacer uso de una función que hayamos desarrollado para un fin determinado.

Al utilizar una función personalizada en lugar de una *lambda*, podemos tener un código más legible y reutilizable. Además, nos brinda la flexibilidad de implementar lógicas de filtrado más complejas que podrían ser difíciles de expresar con una lambda simple.

```
def esElegible(edad):
    if edad < 18:
        return False
    elif 18 <= edad < 30:
        return True  #Jóvenes adultos son elegibles
    elif 30 <= edad < 50:
        return True  #Adultos de mediana edad son elegibles
    else:
        return False  #Personas mayores no son elegibles


edades = [11, 22, 33, 28, 45, 60, 18]

edadesElegibles = list(filter(esElegible, edades))

print(edadesElegibles)
```

FUNCIÓN REDUCE

Es una función que toma como argumento cualquier objeto iterable y lo "reduce" a un único valor. Cómo se obtiene ese único valor a partir de la colección pasada como argumento dependerá de la función aplicada (que debe tener dos parámetros definidos).

opcional

reduce (*función, iterable, valor inicial*)

Para utilizar la función, se debe importarla desde el módulo *functools*.

Por ejemplo, el siguiente código reduce la lista [1, 2, 3, 4] al número 10 aplicando la función suma(a, b), que retorna la suma de sus argumentos.

```
from functools import reduce

def suma(a, b):
    return a + b

print(reduce(suma, [1, 2, 3, 4]))  #10
```

FUNCIÓN REDUCE

Como se mencionó anteriormente, la función pasada como primer argumento debe tener dos parámetros. *Reduce()* se encargará de llamarla de forma acumulativa (es decir, preservando el resultado de llamadas anteriores) de izquierda a derecha.

De modo que el código anterior es similar a:

```
print(suma(suma(suma(1, 2), 3), 4))
```

Es decir, la operación realizada es ((1 + 2) + 3) + 4, de la que resulta 10.

```
from functools import reduce

def suma(a, b):
    return a + b

print(reduce(suma, [1, 2, 3, 4]))  #10
```

EJEMPLOS DE FUNCIÓN REDUCE

La función *reduce* es una herramienta versátil que permite realizar operaciones acumulativas sobre los elementos de un iterable, transformando múltiples valores en un solo resultado.

Su uso se extiende desde cálculos matemáticos simples, como sumar o multiplicar números, hasta tareas más complejas, como la concatenación de cadenas y la filtración de datos.

Gracias a su integración con funciones *lambda*, reduce permite escribir código más conciso y legible.

```
from functools import reduce

#Usamos reduce con una función lambda para encontrar el máximo
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
resultadoMaximo = reduce(lambda x, y: x if x > y else y, numeros)
print(resultadoMaximo) #9

#Usamos reduce con una función lambda para concatenar las cadenas
cadenas = ["¡", "Hola", " ", "mundo", "!"]
resultadoConcatenacion = reduce(lambda x, y: x + y, cadenas)
print(resultadoConcatenacion) #¡Hola mundo!

#Usamos reduce con una función lambda para sumar las longitudes,
comenzando desde 0, usándose como el primer valor de x, por lo que x es un
entero desde el principio y len(y) también es un entero, lo que permite la suma
sin errores.
cadenas = ["La", "ciencia", "es", "un", "arte"]
resultadoLongitud = reduce(lambda x, y: x + len(y), cadenas, 0)
print(resultadoLongitud) #17
```

Lic. Yaps David P.

AUTOEVALUACIÓN

La autoevaluación es crucial para identificar mejoras y fortalezas de forma objetiva, facilitando la fijación de metas y la toma de acciones para alcanzarlas.