Lectura

Funciones Lambda y de Orden Superior

Python Intermedio

Introducción

En esta primera unidad se abordarán dos herramientas fundamentales de la programación en Python:

- **Funciones lambda**, que permiten definir operaciones sencillas en una sola línea.
- **Funciones de orden superior** y su aplicación mediante herramientas como map, filter, reduce y sorted, muy útiles para transformar y procesar colecciones de datos.

El uso de estas herramientas no solo **fortalece tu conocimiento del lenguaje**, sino que también **desarrolla un estilo de programación más estructurado y profesional.** Su comprensión te permitirá escribir código más compacto, legible y reutilizable, optimizando tareas repetitivas y reduciendo errores comunes.

Además, estas funciones son ampliamente empleadas en la práctica profesional, especialmente en áreas como análisis de datos, procesamiento de colecciones y desarrollo de aplicaciones modernas, donde la eficiencia y la claridad del código son clave. Al dominar estos conceptos, estarás preparado para resolver problemas de programación con un enfoque más analítico, práctico y cercano a los estándares utilizados en proyectos reales.

Función lambda

Función **anónima** y **corta** definida con la palabra **reservada *lambda***. Acepta uno o más parámetros y devuelve una expresión evaluada. No admite múltiples sentencias ni bloques de código complejos. Sus características son:

- **No tienen nombre** (a menos que se asignen a una variable).
- Pueden recibir cualquier número de parámetros, pero solo permiten **una expresión** (no bloques de código).

- Son útiles cuando se necesita una función **rápida y sencilla**, sin tener que usar def.
- Se utilizan con gran frecuencia junto con funciones de orden superior (map, filter, sorted).

Su sintaxis es:

```
lambda argumentos: expresión
```

A continuación se presentan algunos ejemplos:

- El siguiente ejemplo muestra una función **lambda básica:**

```python
# Función que suma 10 a un número
suma10 = lambda x: x + 10

print(suma10(5))  # 15
```

➔ lambda x: x + 10 crea una función que recibe x, y devuelve x + 10.

➔ Se asigna a la variable suma10, que ahora funciona como si fuera una función normal.

- El siguiente ejemplo muestra una función **lambda con 2 parámetros:**

```python
multiplicar = lambda a, b: a * b

print(multiplicar(4, 5))  # 20
```

➔ Aquí la función lambda recibe **dos argumentos** (a y b).

➔ Devuelve su producto.

Para saber más

Los casos de uso comunes de funciones lambda son:

- Funciones rápidas y simples en una sola línea.
- Ordenar listas con criterios personalizados.
- Aplicar operaciones a listas con map, filter, reduce.
- Evitar escribir funciones completas con def, cuando solo se necesitan una vez.

Funciones de orden superior

Son funciones que pueden recibir como parámetro a otras funciones y/o devolver una función como resultado. Este concepto es fundamental en la **programación funcional**, ya que permite tratar las funciones como datos.

A continuación se presenta un ejemplo:

```
1   def aplicar_funcion(f, valor):
2       return f(valor)
3
4   def cuadrado(x):
5       return x * x
6
7   print(aplicar_funcion(cuadrado, 5))  # 25
```

➔ aplicar_funcion es una función de orden superior porque recibe como parámetro otra función (cuadrado).

➔ Luego la aplica al valor 5.

Map()

La función **map()** aplica una función a **cada elemento de un iterable** (lista, tupla, conjunto, etc.) y devuelve un **objeto map** (que normalmente convertimos en lista para ver el resultado).

Su sintaxis es:

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

- ➔ **funcion** → lo que se aplicará a cada elemento.
- ➔ **iterable** → lista, tupla u otro conjunto de datos.

A continuación se presentan algunos ejemplos:

- El siguiente ejemplo muestra cómo **duplicar números:**

```python
numeros = [1, 2, 3, 4]

duplicados = list(map(lambda x: x * 2, numeros))
print(duplicados)
# [2, 4, 6, 8]
```

- ➔ map() recorre cada número de la lista numeros.
- ➔ La función lambda x: x * 2 multiplica cada valor por 2.
- ➔ Convertimos el resultado en lista con list().

- El siguiente ejemplo muestra cómo convertir **a mayúsculas:**

```python
palabras = ["hola", "python", "intermedio"]

mayusculas = list(map(lambda x: x.upper(), palabras))
print(mayusculas)
# ['HOLA', 'PYTHON', 'INTERMEDIO']
```

➔ Cada palabra de la lista se pasa a mayúsculas con el método .upper().

➔ map() aplica esa transformación a todos los elementos de forma automática.

- El siguiente ejemplo muestra cómo **calcular longitudes de palabras:**

```python
palabras = ["uva", "sandía", "kiwi", "manzana"]

longitudes = list(map(lambda x: len(x), palabras))
print(longitudes)
# [3, 6, 4, 7]
```

➔ La función lambda x: len(x) devuelve la longitud de cada palabra.

➔ map() aplica esa función a cada elemento de la lista.

- El siguiente ejemplo muestra **cómo usar *map()*** con funciones definidas con ***def*.**

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

numeros = [1, 2, 3, 4]

resultado = list(map(cuadrado, numeros))
print(resultado)
# [1, 4, 9, 16]
```

➔ Aquí no usamos lambda, sino una función normal def cuadrado.

➔ map() recibe la referencia a la función (cuadrado) y la aplica a cada número.

Para saber más

Los casos de uso prácticos para la función **map()** son:
- Transformar listas completas en una sola línea.
- Normalizar datos (ej. pasar todos los textos a minúsculas).
- Aplicar cálculos repetitivos a muchos elementos (ej. convertir de Celsius a Fahrenheit).

Filter()

Filtra los elementos de un **iterable**, según una función que devuelve **True** o **False**.

Su sintaxis es:

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

En donde:

- ➔ **funcion**→ recibe un elemento y devuelve True o False.
- ➔ **iterable** → lista, tupla, conjunto, etc.
- ➔ Devuelve un objeto filter (normalmente lo convertimos en list()).

A continuación se presentan algunos ejemplos:

- El siguiente ejemplo muestra **cómo filtrar números pares:**

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

pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)
# [2, 4, 6]
```

- ➔ La función lambda x: x % 2 == 0 devuelve True si el número es par.
- ➔ filter() recorre la lista y mantiene solo los que cumplen esa condición.

- El siguiente ejemplo muestra **cómo filtrar palabras largas:**

```
palabras = ["uva", "sandía", "kiwi", "manzana"]

largas = list(filter(lambda x: len(x) > 4, palabras))
print(largas)
# ['sandía', 'manzana']
```

- ➔ La función lambda x: len(x) > 4 devuelve True si la palabra tiene más de 4 letras.
- ➔ filter() conserva solo "sandía" y "manzana".

- El siguiente ejemplo muestra **cómo filtrar valores positivos:**

```
numeros = [-5, -1, 0, 3, 7, -2, 10]

positivos = list(filter(lambda x: x > 0, numeros))
print(positivos)
# [3, 7, 10]
```

➔ Aquí la condición x > 0 selecciona solo los números mayores que cero.

➔ Resultado: [3, 7, 10].

- El siguiente ejemplo muestra **cómo utilizar la función *filter()* con una función normal:**

```
def es_vocal(letra):
    return letra.lower() in "aeiou"

letras = ["a", "b", "c", "e", "i", "x"]

solo_vocales = list(filter(es_vocal, letras))
print(solo_vocales)
# ['a', 'e', 'i']
```

➔ La función es_vocal devuelve True si la letra está en "aeiou".

➔ filter() recorre la lista y deja únicamente las vocales.

Para saber más

Los casos de uso prácticos de la función *filter()* son:

- Filtrar datos válidos en una lista (ej. solo valores positivos, solo cadenas no vacías).
- Seleccionar registros que cumplen ciertas condiciones (ej. estudiantes aprobados con nota >= 70).
- Limpiar información antes de procesarla (ej. quitar None o valores negativos).

Reduce()

La función **reduce()** está en el módulo functools y sirve para **reducir** una lista o iterable a **un solo valor**, aplicando una función acumuladora de **dos parámetros**.

Su sintaxis es:

```
from functools import reduce

reduce(funcion, iterable[, valor_inicial])
```

En donde:

- ➔ **funcion** → recibe dos argumentos: el acumulador y el siguiente valor.
- ➔ **iterable** → la lista o tupla a procesar.
- ➔ **valor_inicial** *(opcional)* → punto de partida del acumulador.

A continuación se presentan algunos ejemplos:

- El siguiente ejemplo **suma todos los números:**

```python
from functools import reduce

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

suma = reduce(lambda x, y: x + y, numeros)
print(suma)
# 15
```

➔ reduce() empieza con los dos primeros números: 1 + 2 = 3.

➔ Luego toma ese resultado y el siguiente: 3 + 3 = 6.

➔ Después 6 + 4 = 10, y 10 + 5 = 15.

➔ Al final queda un único valor: 15.

- El siguiente ejemplo muestra **cómo multiplicar todos los números:**

```python
from functools import reduce

numeros = [1, 2, 3, 4]

producto = reduce(lambda x, y: x * y, numeros)
print(producto)
# 24
```

➔ Empieza con 1 * 2 = 2.

➔ Luego 2 * 3 = 6.

➔ Después 6 * 4 = 24.

➔ Resultado final: 24.

- El siguiente ejemplo muestra **cómo concatenar palabras:**

```python
from functools import reduce

palabras = ["Python", "es", "genial"]

frase = reduce(lambda x, y: x + " " + y, palabras)
print(frase)
# "Python es genial"
```

➔ Primero concatena "Python" + "es" → "Python es".

➔ Luego "Python es" + "genial" → "Python es genial".

➔ Resultado: una sola cadena formada con todas las palabras.

Sorted()

Ordena elementos de un **iterable** de forma ascendente (o descendente con **reverse=True**). La función **key** define el **criterio de ordenación**. Las características principales de esta función son:

- No modifica el iterable original, siempre genera una copia ordenada.
- Por defecto, ordena de menor a mayor (ascendente).
- Se puede invertir el orden con el parámetro reverse=True.
- El parámetro key permite personalizar el criterio de ordenación.
    - Ejemplo: key=len ordena por longitud de elementos.
      También puede usarse una función lambda para mayor flexibilidad.
    - Soporta ordenación de datos numéricos, cadenas de texto y estructuras más complejas, como listas de tuplas o diccionarios.
- Si se aplican criterios mixtos (ej. números y cadenas en la misma lista),

puede generar error porque Python no puede compararlos directamente.

Su sintaxis es:

```
sorted(iterable, *, key=None, reverse=False)
```

A continuación se presentan algunos ejemplos:

- El siguiente ejemplo **ordena una lista de números usando sorted():**

```
numeros = [4, 1, 7, 3, 9, 2]

ordenados = sorted(numeros)
print(ordenados)
# [1, 2, 3, 4, 7, 9]
```

➔ La lista numeros está desordenada.

➔ sorted(numeros) devuelve una nueva lista con los elementos en orden ascendente (de menor a mayor).

➔ La lista original no cambia, porque sorted() no modifica, sino que crea una copia ordenada.

- El siguiente ejemplo **ordena palabras alfabéticamente:**

```
palabras = ["zorro", "perro", "gato", "elefante"]

ordenadas = sorted(palabras)
print(ordenadas)
# ['elefante', 'gato', 'perro', 'zorro']
```

  - ➔ Cuando se trabaja con texto, sorted() ordena las palabras en orden alfabético.
  - ➔ Aquí, "elefante" aparece primero porque empieza con "e", luego "gato", después "perro" y finalmente "zorro".

- El siguiente ejemplo muestra **cómo ordenar por longitud de palabras:**

```
palabras = ["zorro", "perro", "gato", "elefante"]

ordenadas = sorted(palabras, key=len)
print(ordenadas)
# ['gato', 'zorro', 'perro', 'elefante']
```

  - ➔ El parámetro key permite definir **cómo** queremos ordenar.
  - ➔ key=len significa que se usará la **longitud de cada palabra** como criterio.
  - ➔ Resultado: primero las palabras cortas, luego las más largas.

- El siguiente ejemplo muestra **cómo obtener un ordenamiento descendente:**

```
numeros = [4, 1, 7, 3, 9, 2]

ordenados_desc = sorted(numeros, reverse=True)
print(ordenados_desc)
# [9, 7, 4, 3, 2, 1]
```

- ➔ El parámetro reverse=True cambia el orden de la lista para que sea de mayor a menor.
- ➔ Es muy útil cuando queremos ver primero los valores más grandes.

Conclusión

Las funciones lambda y de orden superior son herramientas esenciales en Python, las cuales optimizan el código, haciéndolo más legible y fácil de mantener. Permiten realizar transformaciones, filtrados y reducciones de datos de manera eficiente, mejorando la claridad y el rendimiento del desarrollo de programas. Su dominio fomenta un estilo de programación profesional, capacitando al desarrollador para crear soluciones limpias y reutilizables, lo cual constituye una ventaja clave en la resolución de problemas de programación reales.

El aprendizaje de estas funciones constituye una base sólida para enfrentar temas más complejos, como el manejo avanzado de estructuras de datos y la programación orientada a objetos, que se abordarán en las siguientes unidades. La práctica constante con distintos ejemplos y el proponer variaciones propias, ayudará a consolidar el conocimiento adquirido, garantizando que estas herramientas se conviertan en parte natural del estilo de programación del estudiante.

Elaboró: Enrique Quezada Próspero

Contenido: Enrique Quezada Próspero

DI: Génesis Equihua

Referencias:

1. Campesato, O. (2023). *Intermediate Python*. Mercury Learning and Information.
2. Python Software Foundation. (2025a). *Built-in Functions*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/library/functions.html
3. Python Software Foundation. (2025b). *functools — Higher-order functions and operations on callable objects*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/library/functools.html
4. Python Software Foundation. (2025c). *itertools — Functions creating iterators for efficient looping*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/library/itertools.html
5. Python Software Foundation. (2025d). *The Python Tutorial – Functional Programming*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions