**Módulo de 10:**

# **Funciones lambda, orden superior, Lazy evaluation y Scope:**

# **Funciones lambda:**

* argumentos:
* expresión:

**Funciones de orden superior.**

* map
* Filter
* Reduce

**Lazy evaluation**

**Ámbitos o Scope**

**Ámbitos en Python**

* Ámbito Local (Local Scope):
* Ámbito Encerrado (Enclosing Scope)
* Ámbito Global (Global Scope):
* Ámbito de Nombre Integrado (Built-in Scope):

**Funciones lambda:**

Las funciones lambda, también conocidas como funciones anónimas, son funciones pequeñas y de una sola línea que se definen sin un nombre utilizando la palabra clave lambda.

|  |
| --- |
| En realidad si bien la comunidad recomienda no usar nombres la mayoría de los ejemplos son con nombre |

Son útiles cuando necesitas una función simple de una linea sin tener que definirla formalmente utilizando def.

La sintaxis básica de empaquetados y desempaquetados en Python es la siguiente:

lambda argumentos: expresion

**argumentos:**

Si los hay son los argumentos de la función separados por comas.

**expresión:**

Es una expresión que define el cuerpo de la función lambda.

Esta expresión se puede evalúar.

Se devuelve como resultado de la función.

#---------------------------------------------------------------------

·

|  |
| --- |
| suma = lambda a, b: a + b  print(f"{suma(3,6)=}") |
| suma(3,6)=9 |

·

|  |
| --- |
| salida = lambda a, b: a + b if a>b else a - b  print(f"{salida(3,6)=}")  print(f"{salida(6,3)=}") |
| salida(3,6)=-3  salida(6,3)=9 |

·

**Funciones de orden superior.**

En Python todos son objetos, las funciones también.

Las funciones de orden superior en Python son aquellas funciones que pueden aceptar otras funciones como argumentos de entrada, manipularlas de manera flexible dentro de otras funciones y al final devolver funciones como resultado.

|  |
| --- |
| En Python, las funciones de orden superior se benefician del hecho de que las funciones son ciudadanos de primera clase, lo que significa que se pueden tratar como cualquier otro objeto, como enteros, cadenas o listas. Esto permite realizar operaciones avanzadas con funciones, como pasarlas como argumentos a otras funciones, asignarlas a variables, almacenarlas en estructuras de datos, retornarlas como resultado de una función, etc. |

·

|  |
| --- |
| def aplicar\_operacion(funcion\_entrada, numeros):  resultado = []  for numero in numeros:  resultado.append(funcion\_entrada(numero))  return resultado  def cuadrado(x):  return x \*\* 2  def cubo(x):  return x \*\* 3  #--------------------------------------------------------  numeros = [1, 2, 3, 4, 5]  resultado\_cuadrado = aplicar\_operacion(cuadrado, numeros)  print(f"{resultado\_cuadrado=}")  resultado\_cubo = aplicar\_operacion(cubo, numeros)  print(f"{resultado\_cubo=}") |
| resultado\_cuadrado=[1, 4, 9, 16, 25]  resultado\_cubo=[1, 8, 27, 64, 125] |

·

|  |
| --- |
| def mi\_funcion(operacion):  if operacion == "suma":  def suma(a, b):  return a + b  return suma  elif operacion == "resta":  def resta(a, b):  return a - b  return resta  #--------------------------------------------------------  operacion = mi\_funcion("suma")  resultado = operacion(3, 4)  print(f"suma=",resultado) # Salida: 7  operacion = mi\_funcion("resta")  resultado = operacion(8, 4)  print(f"resta=",resultado) # Salida: 4  resultado = mi\_funcion("suma")(8, 2)  print(f"suma=",resultado) # Salida: 6 |
| suma= 7  resta= 4  suma= 10 |

·

**map**

La función map() en Python es una función de orden superior que se utiliza para aplicar una función original dada a cada elemento de un iterable (como una lista, tupla o set) y devuelve un objeto map que contiene los resultados. Es muy útil cuando necesitas aplicar una función a múltiples elementos de una colección iterable de manera eficiente y elegante de procesar datos en Python.

La sintaxis básica de la función map() es la siguiente:

map(función, iterable)

#---------------------------------------------------------------------

·

|  |
| --- |
| def cuadrado(cada\_numero):  resultado = cada\_numero \*\* 2  return resultado  #--------------------------------------------------------  numeros = [1, 2, 3, 4, 5]  regreso = map(cuadrado, numeros)  print (f"{regreso=}")  print (f"{type(regreso)=}")  print (f"{list(regreso)=}") |
| regreso=<map object at 0x000001B673BD5E70>  type(regreso)=<class 'map'>  list(regreso)=[1, 4, 9, 16, 25] |

Se define la función estándar con el nombre cuadrado(parámetro) con un return parámetro al cuadrado (\*\*2)

Map aplica esta función a cada elemento de la lista números. Los regresos se coleccionan en un objeto map. Para ver su contenido hacemos un list(objeto map) o un for de cada dato map

·

|  |
| --- |
| palabras = ["Python", "es", "genial"]  # len como función original  longitudes = map(len, palabras)  print (f"{longitudes=}")  print (f"{type(longitudes)=}")  for palab, longi in zip (palabras,longitudes):  print (f"palabra:'{palab}' longitud-len= {longi}") |
| longitudes=<map object at 0x00000221058A5D20>  type(longitudes)=<class 'map'>  palabra:'Python' longitud-len= 6  palabra:'es' longitud-len= 2  palabra:'genial' longitud-len= 6 |

·

Len es una función built-in incorporada para calcular la longitud de un string.

Map() se aplica a la función len() y se genera un iterable con las longitudes correspondientes.

Es importante tener en cuenta que map() devuelve un objeto map que es un iterable. Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

**Filter**

filter() es una función de orden superior que se utiliza para filtrar elementos de un iterable (como una lista, tupla o set) según una función original de filtrado dada.

Es importante tener en cuenta que filter() devuelve un objeto filter que es un iterable. Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

La sintaxis básica de la función filter() es la siguiente:

filter(función, iterable)

#---------------------------------------------------------------------

En este caso la función debe ser un filtro que solo regrese un True o un False.

Filter aplica a cada elemento del iterable de la colección y True se incluirá en el objeto filter mientras que los False serán excluido.

iterable: Es el objeto iterable (como una lista, tupla, conjunto) del que se desean filtrar los elementos.

La función filter() es útil cuando necesitas seleccionar elementos específicos de un iterable según una condición. Proporciona una forma concisa y eficiente de filtrar datos en Python.

·

|  |
| --- |
| def es\_par(numero):# original  return numero % 2 == 0  #--------------------------------------------------------  numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  solo\_pares = filter(es\_par, numeros)  print (f"{solo\_pares=}")  print (f"{type(solo\_pares)=}")  for cada\_numero in (solo\_pares):  print (f"\t\tnúmero {cada\_numero=}") |
| solo\_pares=<filter object at 0x000001C3A6DC5D20>  type(solo\_pares)=<class 'filter'>  número cada\_numero=2  número cada\_numero=4  número cada\_numero=6  número cada\_numero=8  número cada\_numero=10 |

·

**Reduce**

|  |
| --- |
| Es necesario recordar que reduce necesita ser importado de la librería functools |

reduce() es una función de orden superior que se utiliza para aplicar una función acumulativa original a los elementos de un iterable y reducirlos a un solo valor.

Esta función se encuentra en el Módulo functools y requiere ser importada antes de su uso.

La sintaxis básica de la función reduce() es la siguiente:

from functools import reduce

reduce(función, iterable)

reduce(función, iterable, [initializer])

#---------------------------------------------------------------------

Al ser una función acumulativa que se aplicará a los elementos del iterable. Esta función original debe tomar dos argumentos y devolver el resultado de la operación acumulativa.

iterable: Es el objeto iterable (como una lista, tupla, conjunto) que se desea reducir a un solo valor.

initializer (opcional): Es un valor inicial que se utiliza como el primer argumento en la primera llamada a la función acumulativa. Si no se proporciona, el primer elemento del iterable se utiliza como valor inicial.

Es importante tener en cuenta que reduce() requiere al menos dos elementos en el iterable. Si el iterable está vacío y no se proporciona un valor inicial (initializer), se generará un error.

La función reduce() es útil cuando necesitas realizar una operación acumulativa en los elementos de un iterable y reducirlos a un solo valor. Algunos ejemplos comunes de uso incluyen la suma de todos los elementos, el cálculo del producto de todos los elementos, la concatenación de cadenas, entre otros.

Espero que esta explicación aclare cómo funciona la función reduce() como una función de orden superior en Python. Si tienes más preguntas, no dudes en hacerlas.

|  |
| --- |
| #--------------------------------------------------------  from functools import reduce  def sumar(acumulador, valor\_nuevo ):  print (f"{acumulador=} {valor\_nuevo=}")  return valor\_nuevo + acumulador  #--------------------------------------------------------  numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  resultado = reduce(sumar, numeros)  print (f"{resultado=}")  print (f"{type(resultado)=}") |
| acumulador=1 valor\_nuevo=2  acumulador=3 valor\_nuevo=3  acumulador=6 valor\_nuevo=4  acumulador=10 valor\_nuevo=5  acumulador=15 valor\_nuevo=6  acumulador=21 valor\_nuevo=7  acumulador=28 valor\_nuevo=8  acumulador=36 valor\_nuevo=9  acumulador=45 valor\_nuevo=10  resultado=55  type(resultado)=<class 'int'> |

En este ejemplo, se define la función sumar(acumulador, valor\_nuevo) que toma dos argumentos y devuelve su suma return valor\_nuevo + acumulador.

Esto se itera para cada elemento de la colección y acumula sucesivamente los elementos del return

Pueden comparar con print (sum( numeros))

·

Funciones lambda + funciones de orden superior.

|  |
| --- |
| lambda <parámetro> :expresión  Funciones anónimas que en Python pueden llevar nombre :(   * + lambda + map   + lambda + filter   + lambda + reduce (import functools) |

·

|  |
| --- |
| string = "Python es el mejor lenguaje de programación"  palabras = string.split()  print (f"{palabras=}")  longitud\_mayor\_a\_5 = filter(lambda palabra: len(palabra) >= 5, palabras)  print(f"{list(longitud\_mayor\_a\_5)=}") |
| palabras=['Python', 'es', 'el', 'mejor', 'lenguaje', 'de', 'programación']  list(longitud\_mayor\_a\_5)=['Python', 'mejor', 'lenguaje', 'programación'] |

·

En este ejemplo, se utiliza una función lambda para verificar si la longitud de una cadena es mayor o igual a 5. filter() se aplica a la función lambda y a cada elemento de la lista de palabras, y se generan objeto tipo filter que es un iterable con las palabras con longitud >= a 5.

Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

·

|  |
| --- |
| todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  resultado = filter(lambda x: x % 2 == 0, todos)  print(f" la salida de filter es {resultado}")  print(f" type {type(resultado)}")  print(f" la lista de la salida de filter es {list(resultado)}") |
| la salida de filter es <filter object at 0x000001E57FA8F790>  type <class 'filter'>  la lista de la salida de filter es [2, 4, 6, 8, 0] |

·

La función filter() es útil cuando necesitas seleccionar elementos específicos de un iterable según una condición de filtrado. Puedes utilizar una función definida por el usuario o una función lambda para especificar la condición de filtrado. Esto te permite realizar operaciones más complejas de selección y filtrado en tus datos.

·

|  |
| --- |
| todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  pares = [2, 4, 6, 8, 10]  resultado = filter(lambda xy: xy[0] in pares and xy[1] in pares, zip(todos, pares))  print(f" la salida de filter es {resultado}")  print(f" type {type(resultado)}")  print(f" la lista de la salida de filter es {list(resultado)}") |
| la salida de filter es <filter object at 0x000001A994805720>  type <class 'filter'>  la lista de la salida de filter es [(2, 4), (4, 8)] |

·

Una función lambda para verificar si los elementos de números y pares están presentes en la lista pares. filter() se aplica a la función lambda y a los iterables numeros y pares, y se generan los elementos que cumplen con la condición.

**Lazy evaluation:**

La evaluación perezosa es una estrategia de evaluación utilizada en algunos lenguajes de programación que consiste en posponer la evaluación de una expresión hasta que sea necesaria o se requiera su resultado. En lugar de calcular todos los valores de una expresión de forma inmediata, la evaluación perezosa permite evaluar solo los valores necesarios en un momento dado.

La evaluación perezosa es especialmente útil cuando se trabaja con estructuras de datos potencialmente grandes o cuando se tienen expresiones complejas donde no todos los valores son necesarios en todos los casos. Al posponer la evaluación, se puede evitar el cálculo innecesario de valores que no se utilizarán.

En Python, la evaluación perezosa se puede lograr utilizando generadores, que son objetos que generan una secuencia de valores bajo demanda. Los generadores se definen utilizando la sintaxis de comprensión de listas pero con paréntesis en lugar de corchetes.

|  |
| --- |
| todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  cuadrados\_en\_reserva = (x \*\* 2 for x in todos)  # os primeros 3 cuadrados  print(f"primer valor:",next(cuadrados\_en\_reserva)) # 1  print(f"segundo valor:",next(cuadrados\_en\_reserva)) # 4  print(f"tercer valor:",next(cuadrados\_en\_reserva)) # 9 |
| primer valor: 1  segundo valor: 4  tercer valor: 9 |

·

En este ejemplo, en lugar de generar una lista completa con los cuadrados de los números, se crea un generador cuadrados\_en\_reserva que produce los cuadrados bajo demanda. Al llamar a la función next(cuadrados\_en\_reserva ), se obtiene el siguiente valor del generador.

La evaluación perezosa tiene la ventaja de ahorrar recursos y tiempo de ejecución al evitar cálculos innecesarios. Sin embargo, es importante tener en cuenta que los generadores son un tipo de iteradores y solo pueden ser recorridos una vez. Una vez que se agotan los valores del generador, no se pueden obtener más elementos.

En resumen, es una estrategia que permite posponer la evaluación de expresiones hasta que sea necesario, lo que puede ser beneficioso en términos de eficiencia y rendimiento en ciertos escenarios

**Ámbitos o Scope**

En Python, los ámbitos o scopes determinan la accesibilidad y visibilidad de los distintos tipos de objetos en diferentes partes del código.

Un objeto en ciertos ámbitos puede ser modificado reescrito, con funciones o con métodos de los objetos - Accesibilidad total.

Un objeto en ciertos ámbitos solo puede mostrar su contenido – Visibilidad y utilizar solo los métodos y atributos que no modifica el objeto.

Un objetos con un solo dato creado en la raiz ( root - sobre el margen) tipo str, int, float, bool, puede ser leídos y no modificados si no se les marca como globales.

Un objetos múltiple – Colección con un conjunto de datos creado en la raiz ( root - sobre el margen) tipo list, set, dict,etc (no tuplas o frozensets), puede ser leídos y modificados sin necesidad de marcarlo como globales.

Si los objetos se crean en una función o método no existen en otros ámbitos - otras funciones o métodos

|  |
| --- |
| El siguiente contenido se tomo de bibliografía para los que requieren mas información pero no son parte de este curso. |

|  |
| --- |
| **Ámbitos en Python** |
| Ámbito Local (Local Scope):  El ámbito local se refiere al ámbito dentro de una función o método.  Las variables definidas dentro de una función solo son accesibles dentro de esa función.  Si se intenta acceder a una variable local fuera de su función, se producirá un error. |
| Ámbito Encerrado (Enclosing Scope)  El ámbito encerrado se refiere al ámbito que se encuentra alrededor de una función anidada.  Ocurre cuando hay una función definida dentro de otra función.  Las variables definidas en el ámbito encerrado son accesibles tanto dentro de la función anidada como en la función externa. |
| Ámbito Global (Global Scope):  El ámbito global se refiere al ámbito fuera de cualquier función o clase.  Las variables definidas en el ámbito global son accesibles desde cualquier parte del código, incluyendo dentro de funciones y clases. |
| Ámbito de Nombre Integrado (Built-in Scope):  El ámbito de nombre integrado se refiere a las funciones y variables predefinidas que están disponibles en Python sin necesidad de importar ningún módulo.  Incluye funciones y objetos integrados como print(), len(), range(), etc.  Estas funciones y variables están disponibles en cualquier parte del código.  Es importante tener en cuenta que hay una jerarquía en la resolución de nombres en Python, lo que significa que Python busca una variable en el ámbito local primero, luego en el ámbito encerrado, seguido del ámbito global y finalmente en el ámbito de nombre integrado. Si no se encuentra la variable en ningún ámbito, se genera un error de "NameError". |

En este ejemplo, la variable x tiene un ámbito global y es accesible en todas partes del código. La variable y tiene un ámbito local dentro de la función func(), mientras que la variable z tiene un ámbito encerrado dentro de la función nested\_func(). Las variables x, y y z son accesibles dentro de sus respectivos ámbitos.

Espero que esto aclare los conceptos de los ámbitos de los distintos objetos en funciones en Python. Si tienes más preguntas, no dudes en hacerlas.

|  |
| --- |
| # Ámbito global  x = 10  y = "Sin Datos root"  z = "Sin Datos root"  def func():  # Ámbito local  y = 20  def nested\_func():  # Ámbito encerrado  z = 30  print(f"dentro de función anidada: {x=} {y=} {z=}") # Accede a las variables x, y y z  nested\_func()  print(f"dentro de función: {x=} {y=} {z=}") # Accede a las variables x e y  func()  print(f"root: {x=} {y=} {z=}") # Accede a la variable x |
| dentro de función anidada: x=10 y=20 z=30  dentro de función: x=10 y=20 z='Sin Datos root'  root: x=10 y='Sin Datos root' z='Sin Datos root' |

·