Python soporta la creación de funciones anónimas (es decir, funciones definidas sin nombre), utilizando un constructo llamado “lambda”.

La estructura general de una función lambda es:

```
lambda <args>: <expr>
```

Vamos a echar una función de Python para duplicar el valor de un escalar:

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

Por ejemplo, para utilizar esta función:

```
print(f(2))
4
```

La misma función se puede escribir como función lambda:

```
g = lambda x: x**2
print(g(2))
4
```

Como se puede ver **ambas funciones hacen exactamente lo mismo** y pueden ser **utilizados de la misma manera** .Hay que tener en cuenta las siguientes consideraciones:

- La definición de lambda no incluye una declaración de “retorno” - que siempre contiene una sola expresión que se devuelve.
- Se puede poner en cualquier lugar de una definición lambda se espera una función, y no es necesario asignarla a una variable.
- Las funciones lambda provienen de lenguajes de programación funcionales y el cálculo lambda. Lo normal es que puedan escribirse en una sola línea al simplificarse el código.
- No es exactamente lo mismo que las funciones lambda en lenguajes de programación funcionales puros como haskell o clojure, pero es un concepto muy poderoso que está bien integrado en Python.

Expresión condicional en funciones Lambda

Se puede utilizar la expresión condicional en una función lambda y tener más de un argumento de entrada.

```
f = lambda x,y: ["OK",x,y] if x>3 and y<100 else ["NOOK",x,y]
print(f(4,50))
['FAIL', 4, 200]
```

Podemos definir una estructura de lista de claves del tipo:

```
pares= [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

Escribir una función lambda para ordenar por pares de claves utilizando sus nombres. Para ordenar podemos utilizar el método list.sort () de una lista.

La función que deseen pasar al método sort () debe devolver el nombre (como una cadena) para cada par. Una vez que se pasa al método sort (), se llama exactamente una vez para cada registro de entrada.

Función map

map () es una función con dos argumentos:

```
r = map(func, seq)
```

El primer argumento *func* es el nombre de una función y la segunda una secuencia (por ejemplo, una lista). *map ()* aplica la función *func* a todos los elementos de la lista.

Devuelve una nueva lista con los elementos cambiados por *func* .

```
pares.sort(key=lambda par: par[1])
```

Vamos a definir una lista de palabras: list_words = [“python”,”spark”, ‘scala’, ‘lambda’]

Creamos una función de mapa para imprimir el número de caracteres de cada palabra:

```
print(list(map(len,list_words)))
```

Funcion filter

Como su nombre indica, *la función* filter se puede utilizar para filtrar los datos. Se prueba cada elemento de los datos de entrada y devuelve un subconjunto de la misma para que una condición dada por una función es TRUE. No modifica los datos de entrada.

Por ejemplo de un rango de números podríamos extraer sólo los números impares

```
numeros = range (-15, 15)
impares= list(filter(lambda x: x%2!=0, numeros))
print(odd_numbers)
[-15, -13, -11, -9, -7, -5, -3, -1, 1, 3, 5, 7, 9, 11, 13]
```

Función reduce

Reduce toma una función f y una matriz como entrada. La función f recibe dos parámetros de entrada que trabajan en los elementos individuales de la matriz. Reduce combina cada dos elementos de la matriz mediante la función f.

La función reduce la cogemos del paquete functools

```
# lista de enteros
enteros = [1, 4, 6, 2, 9, 10]

# Definimos un función que convierta cada x,y en una tupla
def combinar(x,y):
 return "(" + str(x) + ", " + str(y) + ")"

# Usamos reduce para aplicar la funcion combinar a la lista de numeros
from functools import reduce

print(enteros)
reduce(combinar,enteros)
```

Combinando reduce y función lambda,vemos que el código se simplifica:

```
# lista de enteros
enteros= [1, 4, 6, 2, 9, 10]

# Usamos reduce para aplicar una funcion lambda con la misma lógica de combinar
from functools import reduce

print(enteros)
reduce(lambda x,y: "(" + str(x) + ", " + str(y) + ")",enteros)
```

Convertir la temperatura de grados Celsius a Kelvin en pyspark creando un RDD con la lista de temperaturas

```
temperaturas = [10, 3, -5, 25, 1, 9, 29, -10, 5]
rdd_temperaturas = sc.parallelize(temperaturas)
rdd_temperaturas_kelvin= rdd_temperaturas.map(lambda x: x + 273.15).collect()
print(rdd_temperaturas_kelvin)
```

Salida:

Mismo ejemplo de antes pero con pyspark creando un RDD. En este caso la función reduce es propia de pyspark y no es necesario recurrir al paquete functools.

```
# lista de enteros
numeros= [1, 4, 6, 2, 9, 10]

rdd_numeros=sc.parallelize(numeros)

# Usamos reduce para aplicar la funcion combinar
rdd_reduce = rdd_numeros.reduce(lambda x,y: "(" + str(x) + ", " + str(y) + ")")
print(rdd_reduce)
```