Operadores encadenados:

Este termino se refiere a encadenar expresiones en las que por ejemplo se use un and, y contraerlas de la siguiente manera:

1<2 and 2<3

Esto sabemos que es True and True, que devuelve True, pero en lugar de hacer esto también podemos hacer 1<2<3 y devolverá True. Lo que se hace es encontrar un elemento que sea común entre las dos funciones o expresiones y “omitirlo” y que el resultado sea el mismo, de esta manera “omitimos” un and.

Comprensión de listas:

Metodo Tradicional:

lista = [] **for** letra **in** ‘casa’: lista.append(letra)

print(lista)

>>>[‘c’,’a’,’s’,’a’]

Metodo con comprensión de listas:

lista = [letra **for** letra **in** ‘casa’]

print(lista)

>>>[‘c’,’a’,’s’,’a’]

Ahora bien, esto funciona bien cuando simplemente recorremos la lista con un for , pero que pasa cuando el for tiene una condición, como por ejemplo cuando vamos a hacer una lista con todos los múltiplos de un número determinado y tenemos que usar la condicional **numero % 2 ==**0 por ejemplo:

lista = [numero **for** numero **in** range (0, 11) **if** numero % 2 == 0]

print(lista)

>>>[ 0, 2, 4, 8, 10 ]

Aquí como vemos la sintaxis en la siguiente: primero se agrega la variable que va a ser parte de la lista, o sea la que se declara en el for, luego se declara el for con el iterable del cual vamos a formar una lista, y por ultimo se [ podemos obtener todos los números que sean pares pero que además sean las potencias del uno al 10, de esta manera:

lista = [numero **for** numero **in [**numero \*\*2 **for** numero **in** range(0,11)**]if** numero % 2 == 0]

print(lista)

>>>[ 0, 4, 16, 36, 64, 100 ]

En esta línea, se resumen 7 líneas.

Funciones Decoradoras:

***Local y Global:***

Una cosa a tener en cuenta al hacer esto es de la manera de ala que funcionan las funciones y las variables en python, estas pueden ser **locales** o **globales**. Algo curioso que podemos hacer es obtener todas las variables y funciones tanto locales como globales de la siguiente forma:

locals() para las funciones y variables locales

globals() para las funciones y de variables globales

Esto va a devolvernos unos diccionarios uno para las locales y otro para las globales, y como diccionarios al fin, podemos hacer con estos todo lo que podemos hacer con los diccionarios normales, iterar sobre ellos, extraer las claves, extraer los valores, o buscar una clave en el, este diccionario también posee las localizaciones de dichos objetos y funciones en la memoria.

Ahora, cuando tenemos una función dentro de otra función, y la función principal esta llamando a la función local, con un return, no con un print o algo así, o sea que la función principal devuelve una función local, por supuesto, esto por pantalla simplemente va a imprimir la ubicación de este objeto de tipo función, entonces haríamos esto:

def hola ():

def bienvenido(): return “Hola!”

return bienvenido

variable\_temporal = funcion\_local()

funcion\_global = variable\_temporal()

Esto se hace porque en este caso, variable\_temporal es de hecho la ubicación de este objeto de tipo funcion, y si lo llamamos con un paréntesis detrás, entonces si se ejecutara la función, por lo que también podríamos resumir esto de la manera siguiente:

funcion\_global()()

Esto esta justificado por la misma sintaxis de python, esto lo hemos hecho en otras ocasiones cuando por ejemplo estamos llamando a un elemento que este contenido en una lista anidada de listas de la siguiente forma: lista[0][0].

Funciones decoradoras:

Es una funcion que envuelve la ejecución de otra funcion y permite extender su comportamiento, están pensadas para reutilizarlas en una sintaxis de ejecución más sencilla.

O sea, básicamente a una funcion se le pasa otra funcion como **argumento**.

Se puede hacer uso de una funcion decoradora una vez que esta ya esta definida sobre otra funcion, sin tener que pasarle ningún argumento, simplemente se asume que dentro de esta solamente hay una funcion, por tanto, podemos escribir un ‘@’ delante del decorador y debajo en la siguiente línea escribir la funcion a decorar. Esto de decorar puede significar que queremos escribir una línea al final de la ejecución de la funcion, o al principio, en ambas partes.

@decorador\_ya\_definido

Funcion\_a\_decorar\_en\_linea\_siguiente():

Funciones Generadoras e Iteradores:

Normalmente cuando una función es creada, al final esta devuelve un valor cuando se hace el return, eso es lo que devuelve la funcion, pero las funciones generadoras como por ejemplo la funcion range() devuelven mas de uno. Esto es super bueno porque de esta manera no tenemos que almacenar en la memoria toda la lista, sino que la vamos generando sobre la marcha.

En este caso lo que nos interesa es hacer por nosotros mismos una funcion generadora ¿pero cٕómo hacemos eso? Bueno simplemente usamos otra funcion generadora y en base a esta, la modificamos o hacemos comprobaciones y tomamos de esta los valores que realmente nos interesa generar o no.

Ej:

def pares (n):

**for** numero **in** range (n+1):

**if** numero % 2 == 0:

yield numero

Yield, significa ceder, ceder en este caso viene siendo como añadir a las listas, pero en su lugar lo que se hace es que se ‘ceden’ los numero por cada iteración para el objeto de clase generador o sea un iterador. Esto es muy útil también cuando estamos utilizando la comprensión de listas que vimos previamente.

Normalmente no podríamos iterar sobre una cadena de texto, o sobre una lista de números por muy raro que parezca, ya que para poder hacer esto, primero hay que pasarles la funcion **iter()** a dicho objeto de la clase str o list. Dicho esto, también existe una funcion integrada llamada **next()** que parece ser la funcion madre de funciones que vimos antes como fetchone() o readline(), ya que siempre y cuando el objeto sea un iterador, podemos usar la funcion next() de la misma forma que un readline.

Funciones Lamba:

Una funcion lambda, también puede ser definida como una funcion anónima, ya que esta no se define sino que se ejecuta al vuelo, en una sola línea como las listas comprendidas. Estas funciones sirven para poder tanto ejecutar una funcion simple en una línea o funcionar como decoradoras y pasar una funcion más compleja como argumento en un contexto en el que no serie posible pasar una funcion como argumento.

Para hacer esto primeramente debemos saber como escribir funciones sencillas en una línea, la característica que nos ayudará mas será que podemos hacerlo así:

def doblar (num):

**return** numero \* 2

def doblar (num):  **return** num \* 2

A esto se le llama funcion simple y es lo que la funcion tipo lambda replica.

lambda num: num \* 2

El output de ejecutar esto no es otro que la localización de la funcion en la memora por lo que podríamos ejecutarlo simplemente agregando paréntesis al final de esta expresión.

Aquí en la funcion lambda vemos tres elementos importantes, lo que esta delante del paréntesis, el paréntesis, y lo que esta después del paréntesis. El paréntesis define la división entre lo que es entrada y lo que es salida, y por supuesto, las expresiones delante y detrás del paréntesis determinar entrada y salida.

Esta funcion anónima también puede ser evaluada a una variable y guardarla ahí, en vez de darle un nombre y definirla como haríamos normalmente

variable = lambda num: num \* 2

Función Filter:

La funcion Filter es mas que una función un tipo de objeto en sí, es un objeto iterable por lo que el metodo next() es valido para el objeto. Para la creación de este objeto tipo Filter se necesita una funcion con una condición, ya que sin una condición no estamos filtrando nada. Es importante recordar que no es necesario establecer un true y un false, si la condición es alcanzada, devolverá true, si no lo es devolverá false, por lo que se puede perfectamente combinar con funciones lambda que devuelven un objeto del tipo funcion porque los objetos del tipo Filter necesitan un **objeto** del **tipo** **funcion** y un **argumento iterable** para pasarle a dicha funcion(**Es importante** que este argumento sea una lista o un iterable porque recordemos que lo que estamos haciendo es **FILTRAR**). Una vez hecho esto tenemos nuestro objeto iterable tipo Filter con el que podemos trabajar

def pares (lista):

**if** numero % 2 == 0:

return True

list ( **filter (lambda** numero : numero % 2 == 0, lista))

La funcion filter alcanza su verdadero potencial cuando estamos filtrando una cantidad grande de datos, y de datos mas complejos, no solo números, sino por ejemplo de un objeto tipo persona, que tendrá una nombre edad dirección etc, y de estos objetos que contienen esos datos, solamente queremos quedarnos con los mayores de edad , para esto usaríamos un **filter**, a una **lambda** que tome como **argumento** la **lista** de **objetos** **tipo persona**, y como **salida** tomaremos solamente los objetos de la **lista** de los cuales su **self.edad** sea **mayor** que **18**.

Funciones MAP:

Las funciones maps también iteran sobre listas pero en vez de solo filtrar, estas funciones también realizan acciones con las listas. Es importante destacar que los maps también son de hecho objetos, objetos iterables. Estos objetos son el resultado de la interacción de dos listas o de la modificación de una. Podemos por su puesto hacer un map con una lambda lo que parece un poco mas difuso por el uso de comas, pero solamente nos hace falta saber que el orden de ejecución para esta línea de código es el siguiente:

map (lambda input: output)

Hasta aquí parece sencillo, pero se complica cuando el output tiene más de un argumento:

Output >>> x\*y , lista1,lista2,lista3

En este caso x\*y seria la acción de la funcion como tal, no hay condiciones ya que no es un filter, sino una acción, y después de esta hay una serie de comas con demás elementos. La primera coma define la separación de la acción y de los argumentos, y las demás determinan la separación de los argumentos que en este caso serian unas listas. Por lo que este objeto map, que tiene una lambda, que multiplica tres listas se vería así:

map(lambda x, y, z: x\*y\*z , list1,list2,list3)

Luego, esto es un objeto iterable que posee varios elementos que si lo imprimimos por pantalla solo nos va a devolver la ubicación de este objeto en la memoria, para acceder a sus valores de una manera visual debemos convertir este iterable en una lista usando la funcion integrada list.