SESSION 3. SUMMARY

Victor Miguel Terron Macias

21/2/2021

```
setwd("C:/Users/Victor Miguel Terron/Documents/PHASE2/PROCESAMIENTO DE DATOS CON PYTHON/SESION 3/")
```

SESION 3. PROGRAMACIÓN FUNCIONAL, OPERADORES LÓGICOS Y FUNCIONES *lambda*

OBJETIVOS

- Utilizar estructuras de datos y funciones para realizar gran parte de los procesos de un científico de datos.
- Comprender que los programas no son más que datos organizados de alguna manera y funciones aplicadas a dichos datos
- Encadenar condiciones y realizar filtros y comparaciones más complejas.

PREWORK

INTRODUCCIÓN

En la sesión pasada aprendimos sobre estructuras de datos y funciones, dos de los pilares más importantes de la programación. Ya sabemos entonces cómo organizar nuestros datos y cómo encapsular comportamiento de manera que sea reutilizable.

En esta sesión usaremos nuestros dos pilares (estructuras de datos y funciones) para crear programas hechos y derechos. Te sorprenderá saber que los conocimientos que ya tienes en este momento bastan para realizar gran parte de los procesos que realiza un científico de datos en su día a día. Un programa es básicamente:

- Datos estructurados de una cierta manera
- Funciones que sirven para transformar datos
- Cadenas de procesos que utilizan funciones para transformar estructuras de datos

PROGRAMACIÓN FUNCIONAL

A través de las décadas los programadores han ido desarrollando innumerables técnicas para hacer programas resilientes, comprensibles y fáciles de extender.

Muchos de estos conocimientos han sido reunidos en los llamados paradigmas de programación. Un paradigma de programación es básicamente un conjunto de herramientas, métodos y reglas que se reúnen cohesivamente y se utilizan para resolver problemas computacionales.

Hay diferentes “perspectivas” acerca de cuál es la mejor manera de escribir un programa. Cada una de esas “perspectivas” está representada por un paradigma de programación.

En realidad, como somos científicos de datos, no ingenieros de software, no hace falta que dominemos estos paradigmas. Pero hay algunas métodos de un paradigma llamado programación funcional que nos pueden ser de mucha utilidad para entender mejor cómo funcionan algunas de las herramientas que utilizaremos constantemente como científicos de datos.

Básicamente vamos a estudiar 2 funciones: map y filter. Estas 2 funciones nos permiten transformar nuestras estructuras de datos de una manera sumamente intuitiva y útil.

```
In [1]: numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Figure 1: IMG

MAP

Espero que ya haya quedado clara la idea básica de cómo funciona un programa: tenemos estructuras de datos, tenemos funciones, y luego aplicamos esas funciones a nuestras estructuras de datos para transformar nuestros datos.

map es una función que nos ayuda a realizar este procedimiento muy fácilmente. Vamos a ver cómo funciona.

Digamos que tenemos una estructura de datos que se ve así:

```
In [1]: numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Figure 2: IMG

Esto es una simple lista con ints dentro. Ahora, digamos que queremos multiplicar cada uno de los elementos de esta lista por 2. Una manera horrible, lenta e impráctica de hacer esto sería lo siguiente:

```
In [3]: numeros_por_dos = [numeros[0] * 2, numeros[1] * 2, numeros[2] * 2, numeros[3] * 2,
                           numeros[4] * 2, numeros[5] * 2, numeros[6] * 2, numeros[7] * 2,
                           numeros[8] * 2]

        numeros_por_dos
Out[3]: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Figure 3: IMG

¿Por qué es tan horrible esta solución?

- Para empezar, necesitamos escribir muchísimo código (y repetido). Ya aprendimos que una de las reglas de la programación es: si vas a repetir el mismo código múltiples veces, lo mejor sería encapsular ese código en una función.
- Y en segunda, ¿qué pasaría si nuestra lista numeros cambia? Por ejemplo, podríamos agregar elementos o eliminar elementos. En ese caso, el código que estamos utilizando para crear numeros_por_dos podría fallar. Si numeros tiene menos elementos, entonces numeros_por_dos intentaría acceder a un índice que ya no existe y nos lanzaría un error. Si numeros tiene ahora más elementos, entonces numeros_por_dos va a estar incompleto.

Es una muy mala idea escribir este tipo de procesos “a mano”, paso a paso. Vamos a ver ahora cómo podríamos simplificar este proceso usando map.

```
In [4]: def multiplicar_por_dos(numero):

            resultado = numero * 2
            return resultado
```

Figure 4: IMG

En primer lugar, vamos a encapsular nuestro proceso en una función:

¡Listo!

Ahora, lo que hace map es lo siguiente:

- Recibe una función que queremos aplicar a una lista.
- Recibe una lista.
- Aplica la función a la lista elemento por elemento y regresa una nueva lista que contiene los elementos de la lista anterior transformados.

Veamos:

```
In [7]: map(multiplicar_por_dos, numeros)
Out[7]: <map at 0x10dd53160>
```

Figure 5: IMG

Ok, todavía no tenemos el output que queramos, ¿verdad?

Esta función nos regresó un objeto que se llama map y alguna especie de número incomprensible. Bueno, veamos qué pasa cuando usamos otra función list y le pasamos este objeto map. list es una función que intente convertir cualquier cosa que le pases a una lista:

```
In [8]: list(map(multiplicar_por_dos, numeros))
Out[8]: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Figure 6: IMG

¡Lo logramos! ¿Ves qué diferencia? El resultado de este procedimiento puede ser asignado a una variable para ser utilizado después:

Ahora qué pasaría si nuestra lista numeros crece:

No tenemos que cambiar absolutamente nada. map aplica la función elemento por elemento, así que no le importa cuántos elementos haya. Simplemente va a recorrer todos los elementos que encuentre, transformarlos y regresarlos en una nueva lista:

¡Supongo que ya te puedes imaginar el potencial!

Ésta es una de las razones por las que establecimos que era una buena idea tener listas con un solo tipo de dato. A la hora de querer aplicar una función a toda la lista, las cosas se complican mucho cuando tenemos diversos tipos de datos en la misma estructura de datos.

Nuestra función puede ser tan complicada como queramos. Por ejemplo, mira esta función que transforma todos los datos nones en 0s, mientras que los datos pares los regresa sin transformarlos:

Veamos qué pasa si le aplicamos esta función a otra lista:

¡Qué genial!

```
In [9]: numeros_por_dos = list(map(multiplicar_por_dos, numeros))

        numeros_por_dos

Out[9]: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Figure 7: IMG

```
In [11]: numeros

Out[11]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Figure 8: IMG

```
In [12]: numeros_por_dos = list(map(multiplicar_por_dos, numeros))

         numeros_por_dos

Out[12]: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40]
```

Figure 9: IMG

```
In [14]: def si_non_regresar_0(numero):

             if numero % 2 == 0:
                 return numero
             else:
                 return 0
```

Figure 10: IMG

```
In [15]: numeros = [6, 78, 3, 89, 5, 63, 42, 8, 76, 1, 245, 9, 877, 465, 34, 77, 80]

         numeros_non_en_cero = list(map(si_non_regresar_0, numeros))

         numeros_non_en_cero

Out[15]: [6, 78, 0, 0, 0, 0, 42, 8, 76, 0, 0, 0, 0, 0, 34, 0, 80]
```

Figure 11: IMG

```
In [19]: def formato_de_dinero(numero):

             formateado = f'${numero} MXN'
             return formateado
```

Figure 12: IMG

También podemos transformar de un tipo de datos a otro. Por ejemplo, mira esta función que toma un número y lo regresa en forma de string con el signo de dinero añadido y la unidad MXN:

Vamos a verla en acción:

```
In [21]: numeros = [6, 78, 3, 89, 5, 63, 42, 8, 76]

         numeros_formato_dinero = list(map(formato_de_dinero, numeros))

         numeros_formato_dinero

Out[21]: ['$6 MXN',
          '$78 MXN',
          '$3 MXN',
          '$89 MXN',
          '$5 MXN',
          '$63 MXN',
          '$42 MXN',
          '$8 MXN',
          '$76 MXN']
```

Figure 13: IMG

¿Cuánto a que te sientes poderoso? ¡Y apenas estamos empezando! Veamos nuestra siguiente función.

FUNCIÓN filter

El nombre de la función filter explica exactamente lo que la función hace: filtrar. ¿Filtrar qué? Pues elementos en una lista. Veamos cómo lo hace.

Tenemos una lista con números (perdón por estar duro y dale con las listas con números):

```
In [1]: numeros = [5, 8, -2, -34, 9, 78, 45, -67, 12, 48, -94, -76, -3, 25, 47]
```

Figure 14: IMG

Ahora digamos que lo que queremos hacer con esta lista es filtrar todos los valores positivos. Esto significa que la lista resultante solamente va a contener números positivos.

¿Cómo vamos a hacer eso? Pues primero necesitamos una función que nos "avise" cuando un número es positivo:

```
In [2]: def numero_es_positivo(numero):

            if numero >= 0:
                return True
            else:
                return False
```

Figure 15: IMG

La función numero_es_positivo checa si numero es mayor o igual a 0 (if numero >= 0); si esta condición se cumple regresa True (que significa "efectivamente, es positivo"). Si la condición no se cumple regresa False (que es un "no, no es un número positivo").

Ahora veamos qué pasa si usamos filter con esta función y nuestra lista numeros:

Interesante

¿Qué está pasando aquí? filter funciona de la siguiente manera:

```
In [4]: list(filter(numero_es_positivo, numeros))
Out[4]: [5, 8, 9, 78, 45, 12, 48, 25, 47]
```

Figure 16: IMG

1. Recibe una función que regrese True o False.
2. Recibe una lista.
3. Va recorriendo la lista elemento por elemento y le aplica la función a cada elemento de la lista.
4. Cada vez que la función regresa True, filter agrega ese elemento a una nueva lista (la que vamos a obtener de regreso). Cada vez que la función regresa False, filter descarta ese elemento y no lo agrega a la nueva lista.

Es por eso que en nuestro ejemplo ya solamente tenemos los elementos que son positivos. Veamos qué pasaría si quisiéramos ahora filtrar los números negativos:

```
In [5]: def numero_es_negativo(numero):

            if numero < 0:
                return True
            else:
                return False

In [6]: list(filter(numero_es_negativo, numeros))
Out[6]: [-2, -34, -67, -94, -76, -3]
```

Figure 17: IMG

Ahora nuestra función regresa True cuando el número es negativo (if numero < 0). Por lo tanto, filter agrega a la nueva lista todos los números negativos y deja fuera los positivos.

Ahora, muchas veces vamos a necesitar usar más de un criterio al mismo tiempo para filtrar nuestros datos. ¿Cómo podríamos hacer algo así? Para lograr esto vamos a aprender tres nuevos operadores que utilizaremos muchísimo a través de todo el módulo: los operadores lógicos.

OPERADORES LÓGICOS

Los operadores lógicos complementan las funcionalidades de los operadores de comparación. Podríamos decir que "extienden" las funcionalidades. Los operadores lógicos toman una o dos sentencias de comparación y regresan un valor (booleano) que depende del resultado de las comparaciones y del operador lógico que se esté usando. En Python tenemos 3 de estos operadores: and, or y not. Veamos cómo funcionan.

AND

and une dos sentencias de comparación y regresa True sólo cuando ambas sentencias regresen True. En este primer ejemplo, estamos reemplazando las sentencias por simples booleanos, pero sólo lo hacemos para simplificar la lógica y hacer muy evidente el funcionamiento de and. En esta tablita los 1s representan True y los 0s representan False:

Veamos ahora qué pasa si usamos True con algunas comparaciones:

En este ejemplo, ambas comparaciones son verdaderas (True); por lo tanto la sentencia and regresa True. Veamos ahora unas comparaciones que resultan en False. En esta primera, la primera comparación es True pero la segunda es False:

Ahora, la primera es False mientras que la segunda es True:

```
In [14]: print(f'{("")  :8}|{("True"):8}|{("False"):8}')
         print(f'{("True"):8}|{(True and True):8}|{(True and False):8}')
         print(f'{("False"):8}|{(False and True):8}|{(False and False):8}')

                  |True    |False
         True     |       1|       0
         False    |       0|       0
```

Figure 18: IMG

```
In [15]: (5 > 2) and (10 == 10)
Out[15]: True
```

Figure 19: IMG

Ahora, las dos son False:

Veamos entonces cómo podríamos utilizar el operador and en la función filter. Tenemos la siguiente lista:

Lo que queremos es filtrar todos los números que sean nones y menores o iguales a 50. Eso quiere decir que queremos quedarnos sólo con números que sean pares y mayores a 50. Primero tenemos que hacer dos funciones para identificar los números que queremos que permanezcan:

Ahora, como filter sólo recibe una sola función tenemos que hacer una nueva función que utilice nuestras dos funciones anteriores para regresar True sólo si un número es a la vez par y mayor a 50:

¡Estamos listos! Ahora sí, a filtrar nuestra lista:

Tenemos ahora una lista con puros números pares y mayores a 50. ¡Qué cosas maravillosas podremos hacer con esto!

Vayamos ahora a nuestro segundo operador.

OR

or une dos sentencias de comparación y regresa True si una de las dos o ambas sentencias regresen True. Es decir, si hay True en nuestra sentencia, or regresa True:

En la siguiente sentencia, la primera comparación es True, mientras que la segunda es False:

Si ambas comparaciones son False es la única forma en la que el or puede regresar False:

Veamos éste operador usado en nuestra función filter. Vamos a usar las mismas funciones que usamos en el ´and´ pero ahora vamos a usar or para unirlas. Eso quiere decir que nuestro filter va regresar todos los valores que sean pares o que sean mayores a 50:

Esto quiere decir que filtramos los valores que eran menores a 50 y nones.

NOT

El último operador es mucho más sencillo. Lo único que hace este operador es regresar el valor booleano opuesto al que recibió:

```
In [16]: (5 > 2) and (10 < 8)
Out[16]: False
```

Figure 20: IMG

```
In [17]: (5 != 5) and (10 > 2)
Out[17]: False
```

Figure 21: IMG

```
In [18]: (5 != 5) and (10 == 3)
Out[18]: False
```

Figure 22: IMG

```
In [19]: numeros = [3, 6, 78, 54, 36, 55, 32, 11, 37, 89, 87, 88, 90, 63, 74, 5, 97, 6, 3, 225, 46, 79]
```

Figure 23: IMG

```
In [20]: def numero_es_par(numero):

             if numero % 2 == 0:
                 return True
             else:
                 return False
```

```
In [21]: def numero_es_mayor_a_50(numero):

             if numero > 50:
                 return True
             else:
                 return False
```

Figure 24: IMG

```
In [22]: def numero_es_par_y_mayor_a_50(numero):

             if numero_es_par(numero) and numero_es_mayor_a_50(numero):
                 return True
             else:
                 return False
```

Figure 25: IMG

```
In [23]: list(filter(numero_es_par_y_mayor_a_50, numeros))
Out[23]: [78, 54, 88, 90, 74]
```

Figure 26: IMG

```
In [24]: print(f'{("")  :8}|{("True"):8}|{("False"):8}')
         print(f'{("True"):8}|{(True or True):8}|{(True or False):8}')
         print(f'{("False"):8}|{(False or True):8}|{(False or False):8}')
```

```
        |True    |False
True    |       1|       1
False   |       1|       0
```

Figure 27: IMG

```
In [25]: (4 < 19) or (12 > 34)
Out[25]: True
```

Figure 28: IMG

```
In [26]: (3 > 34) or (10 == 22)
Out[26]: False
```

Figure 29: IMG

```
In [28]: list(filter(numero_es_par_o_mayor_a_50, numeros))
Out[28]: [6, 78, 54, 36, 55, 32, 89, 87, 88, 90, 63, 74, 97, 6, 225, 46, 79]
```

Figure 30: IMG

```
In [30]: print(f'{("True"):8}|{(not True):8}')
         print(f'{("False"):8}|{(not False):8}')

         True    |       0
         False   |       1
```

Figure 31: IMG

Es como decir "dame el valor opuesto al que te estoy dando". ¿Para qué querríamos hacer algo así? Por ejemplo, nosotros ya tenemos una función que regresa True cuando el valor que recibe es par:

Si la usamos en nuestra lista numeros, vamos a obtener una nueva lista donde todos los números son pares:

Pero, ¿qué pasa si queremos filtrar ahora los números nones y obtener una lista sólo con números nones? Podríamos escribir otra función que regrese True cuando nuestro número sea non. Pero también podríamos hacer algo mucho más sencillo, usar la función que ya tenemos y simplemente "revertir" el valor que regresa:

¡Hey, hey, hey! ¿Qué está pasando ahí? ¿"Lambda" quién? Esto que acabamos de hacer requirió una nueva herramienta que será nuestro último tema de la sesión de hoy: funciones lambda. Vamos a entender cómo funcionan y luego regresamos a este último ejemplo y lo analizamos.

FUNCIONES LAMBDA

Las funciones lambda son simplemente maneras simplificadas de escribir las funciones que ya conocemos tan bien. No necesitamos entender ningún nuevo concepto, sólo aprender una nueva sintaxis. La sintaxis de una función lambda es la siguiente. Primero escribimos la palabra lambda:

Ahora, agregamos los nombres de nuestro parámetro (o parámetros):

En este caso tenemos un sólo parámetro llamado x.

Nuestro siguiente paso es agregar dos puntos (:). Todo lo que esté escrito después de los dos puntos será nuestro "bloque" de la función:

Como las funciones lambda son funciones simplificadas, no podemos escribir como tal todo un bloque completo para nuestra función. Lo único que podemos hacer es escribir una única sentencia que sería la que estaría escrita después de nuestro return. Es decir, la única sentencia de nuestra función lambda es la que regresa el valor final de nuestra función:

Como puedes ver, las funciones lambda sólo sirven si queremos realizar procesos muy sencillos. En este caso, nuestra función recibe un solo parámetro x y regresa x * 100. Cualquier función que podamos escribir en una sóla línea podría ser una función lambda.

Ahora, ¿cómo es que usamos una de estas funciones lambda? Podemos pasarlas directamente a nuestras funciones map y filter. Si nuestra función es muy sencilla, podemos ahorrarnos tiempo y espacio y definir nuestra función ahí mismo en lugar de tener que declararla primero usando def. Por ejemplo:

```
In [20]: def numero_es_par(numero):

             if numero % 2 == 0:
                 return True
             else:
                 return False
```

Figure 32: IMG

```
In [31]: list(filter(numero_es_par, numeros))
Out[31]: [6, 78, 54, 36, 32, 88, 90, 74, 6, 46]
```

Figure 33: IMG

```
In [35]: list(filter(lambda x: not numero_es_par(x), numeros))
Out[35]: [3, 55, 13, 37, 89, 87, 63, 5, 97, 3, 225, 79]
```

Figure 34: IMG

```
In [ ]: lambda
```

Figure 35: IMG

```
In [ ]: lambda x
```

Figure 36: IMG

```
In [ ]: lambda x
```

Figure 37: IMG

```
In [ ]: lambda x: x * 100
```

Figure 38: IMG

```
In [38]: numeros = [1, 2, 3, 4, 5, 6, 7, 8]
         list(map(lambda x: x * 100, numeros))
Out[38]: [100, 200, 300, 400, 500, 600, 700, 800]
```

Figure 39: IMG

¿Ves? Le hemos pasado nuestra función lambda al map y obtenido de regreso una lista con todos nuestros valores multiplicados por 100.

Veamos otro ejemplo usando map:

```
In [39]: list(map(lambda x: x * (x + 1), numeros))
Out[39]: [2, 6, 12, 20, 30, 42, 56, 72]
```

Figure 40: IMG

Esta función lambda nos regresa el número x multiplicado por la suma de x + 1.

Ahora veamos cómo usarlas con filter:

```
In [42]: list(filter(lambda x: x > 5, numeros))
Out[42]: [6, 7, 8]
```

Figure 41: IMG

Aquí estamos filtrando todos los números iguales o menores a 5 con una sola línea.

Veamos ahora qué pasaba en nuestro ejemplo de arriba:

```
In [35]: list(filter(lambda x: not numero_es_par(x), numeros))
Out[35]: [3, 55, 13, 37, 89, 87, 63, 5, 97, 3, 225, 79]
```

Figure 42: IMG

Lo que está pasando aquí es que la función lambda recibe nuestro número (x), lo pasa a la función numero_es_par y luego regresa el opuesto usando not. ¡Así podemos revertir el funcionamiento de nuestra función original en una sola línea!

Podríamos incluso rehacer nuestra función numero_es_par_o_mayor_a_50 en una sola línea:

¿No te parece genial?

Con todo lo que aprendimos el día de hoy estamos ya listos para sumergirnos en el mundo de la ciencia de datos. Asegúrate de entender todos estos temas a la perfección para que te sientas cómodo con lo que veremos a continuación. ¡Feliz aprendizaje!

WORK. PROGRAMACIÓN FUNCIONAL, OPERADORES LÓGICOS Y FUNCIONES LAMBDA

OBJETIVO

1. Usar las dos funciones más importantes en la programación funcional: **map** y **filter**
2. Utilizar **operadores lógicos** para extender la funcionalidad de nuestros **operadores de comparación**
3. Conocer la sintaxis de las funciones **lambda** para simplificar al definición de funciones

CONTENIDO

Aprendimos en el Prework que la programación funcional es un paradigma de programación. Básicamente, es un conjunto de herramientas, métodos y reglas que sirven para organizar nuestro código y darle coherencia.

Figure 43: IMG

En este curso no nos interesan los detalles de la programación funcional, pero vamos a aprender a usar dos de sus funciones más comunes: map y filter. ¿Por qué? Porque la manera como funcionan se parece mucho a la manera como programan los científicos de datos.

Entendiendo map y filter al 100 te será más fácil aproximarte a las funciones universales en numpy y pandas y a cómo funcionan sus filtros.

¡Vamos adelante!

Elegí los temas de map y filter en lugar de los ciclos porque se parecen mucho más a los paradigmas que utilizan los científicos de datos. Es poco común (y a veces incluso es mala práctica) usar ciclos junto con las librerías de pandas y numpy. Por eso me tomé la libertad impensable de omitirlos.

MAP

La primera función que vamos a aprender es la función **map**, map toma una rfunción y una lista para regresarnos una nueva lista donde la función ha sido aplicada a cada elemento de la lista original:

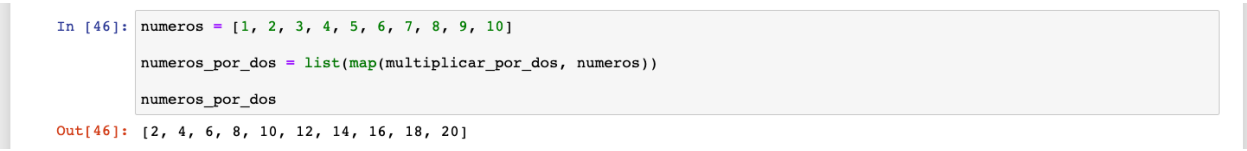

Figure 44: IMG

Veamos como funciona.

Es importante que los alumnos entiendan el concepto de aplicar una función "elemento por elemento" ("element-wise") a una lista, ya que en esencia eso es lo que sucede cuando aplicamos funciones universales o vectorizadas a un numpy.array o a una Serie de pandas.

EJEMPLO 1. MAP

OBJETIVO

- Entender cómo funciona la función map y verla aplicada en ejemplos para después reproducir su uso

DESARROLLO

Muchas veces vamos a querer aplicar funciones a cada uno de los elementos en una lista. Esto es un procedimiento muy común en la ciencia de datos. Aplicar funciones "elemento por elemento" a una lista es bastante tedioso sin utilizar la función map. Por suerte, map hace todo muy fácil e intuitivo.

Para poder usar map primero necesitamos una lista:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

LISTO!

Ahora tenemos que pensar qué proceso queremos aplicar a cada uno de los elementos de la lista. Digamos que queremos multiplicarlos por 10. Hay que escribir una función entonces que reciba un parámetro (que va a ser cada uno de nuestros números) y lo regrese multiplicado por 10:

```
def multiplicar_por_10(numero):

    return numero * 10
```

El siguente paso es aplicar nuestra función a la lista usando map:

```
list(map(multiplicar_por_10, numeros))
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

¿Por qué estamos agregando esa función list? Como viste en el Prework, list nos ayuda a convertir el resultado de map en una lista común y corriente. Podríamos guardar el resultado de este procedimiento en otra variable:

```
numeros_por_10 = list(map(multiplicar_por_10, numeros))

numeros_por_10
```

```
## [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

¡Tan fácil como eso!

Podemos aplicar una infinidad de funciones a nuestra lista usando map. Veamos algunos otros ejemplos:

1, ejemplo de conversion de valores a unidades

```
def convertir_en_string_mas_unidad(numero):

    return f'{numero} seg'

list(map(convertir_en_string_mas_unidad, numeros))
```

```
['1 seg', '2 seg', '3 seg', '4 seg', '5 seg', '6 seg', '7 seg', '8 seg', '9 seg', '10 seg']
```

2, conversión a numeros negativos

```
def convertir_a_numeros_negativos(numero):

    return numero * -1

list(map(convertir_a_numeros_negativos, numeros))
```

```
[-1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
```

3, convertir un numero a 0 si es menor a 5

```
def convertir_en_0_si_menor_a_5(numero):

    if numero < 5:
        return 0
    else:
        return numero

list(map(convertir_en_0_si_menor_a_5, numeros))
```

```
[0, 0, 0, 0, 5, 6, 7, 8, 9, 10]
```

4, convertir en verdadero si el numero es mayor a 6

```
def convertir_en_true_si_mayor_a_6(numero):

    if numero > 6:
        return True
    else:
        return False

list(map(convertir_en_true_si_mayor_a_6, numeros))
```

```
[False, False, False, False, False, False, True, True, True, True]
```

RETO 1. FUNCIÓN MAP

OBJETIVOS

- Prácticar el uso de `map` para transformar los datos en una `lista`

DESARROLLO

A. PROPORCIÓN A PORCENTAJES

Tenemos una lista que contiene proporciones:

```
proporciones = [0.45, 0.2, 0.78, 0.4, 0.77, 0.9, 0.4, 0.5, 0.67, 0.24, 0.73]
```

Queremos convertir esta lista en una lista porcentajes, donde las proporciones hayan sido convertidas a porcentajes. Termina la función proporcion_a_porcentajes y después utiliza map para convertir proporciones y asignar la lista transformada a porcentajes:

```
def proporcion_a_porcentajes(proporcion):

    return f'{proporcion*100}%'

porcentajes=list(map(proporcion_a_porcentajes,proporciones))
porcentajes
```

```
['45.0%', '20.0%', '78.0%', '40.0%', '77.0%', '90.0%', '40.0%', '50.0%', '67.0%', '24.0%', '73.0%']
```

B. STRINGS A NUMEROS

Tenemos una lista con strings que representan valores numericos:

```
numeros_como_strings = ["3", "7", "45", "89", "12", "9", "5", "89", "78", "87", "44", "45", "26", "84",
```

Para realizar algunos calculos estadístiucos, necesitamos que estas strings sean convertidas a ints. Escribe una función llamada string_a_int y asigna el resultado de su aplicación a numeros_como_strings a la variable numeros_como_ints:

```
def string_a_int(string):

    ## Tu código va aquí
    # ...
    # ...

    return int(string)
#RECUERDA SIEMPRE QUE LIST Y MAP TIENEN DOS PARAMETROS, FUNCION Y ORIGEN DE DATOS
numeros_como_ints =list(map(string_a_int,numeros_como_strings))
numeros_como_ints
```

```
[3, 7, 45, 89, 12, 9, 5, 89, 78, 87, 44, 45, 26, 84, 98, 46, 99, 84]
```

Pídele a tu experta la función de verificación imprimir_analisis_estadistico (encontrada en el archivo helpers.py de la carpeta donde se encuentra este Reto), pégala debajo y corre la celda para verificar tu resultado:

```
def imprimir_proporciones_en_equivalencia_a_porcentajes(proporciones, porcentajes):

    print(f'==Proporciones y su equivalencia en porcentajes de 1==\n')

    for i in range(len(proporciones)):
        print(f'- {proporciones[i]} es el {int(porcentajes[i])}% de 1.')


def imprimir_analisis_estadistico(datos):

    def mediana(datos):
        datos_sorted = sorted(datos)
        len_datos = len(datos)

        if len_datos % 2 == 0:
            mediana = (datos_sorted[int(len_datos / 2) - 1] + datos_sorted[int(len_datos / 2)]) / 2
        else:
            import math
            mediana = datos_sorted[int(math.floor(len_datos / 2))]

        return mediana

    print(f'==Análisis estadístico de los datos recibidos==\n')
    print(f'Valor mínimo: {min(datos)}')
    print(f'Valor máximo: {max(datos)}')
```

```
    print(f'Rango de valores: {max(datos) - min(datos)}')
    print(f'Promedio: {sum(datos) / len(datos)}')
    print(f'Mediana: {mediana(datos)}')



imprimir_analisis_estadistico(numeros_como_ints)
```

```
## ==Análisis estadístico de los datos recibidos==
##
## Valor mínimo: 3
## Valor máximo: 99
## Rango de valores: 96
## Promedio: 52.77777777777778
## Mediana: 45.5
```

EJEMPLO 2. FILTER

OBJETIVOS

- Entender cómo funciona la función filter y verla aplicada en ejemplos para después poder reproducir su uso

DESARROLLO

filter nos permite filtrar nuestras listas para dejar fuera elementos que no queremos. Tal vez te parezca un poco extraño esto. ¿Por qué queremos filtrar datos? Una de nuestras tareas más importantes como procesadores de datos es la de limpiar nuestros conjuntos de datos para que tengan solamente los datos que necesitamos para nuestro análisis. Una de las técnicas de limpieza más comunes es la de filtrar nuestro conjunto de datos. Vamos a aprender a hacer esto usando filter.

Como ya vimos, filter recibe una función y una lista, y regresa una nueva lista con los elementos que fueron filtrados. La función debe de regresar True o False. Cada que la función regresa True, el elemento al que le fue aplicado la función se agrega a la nueva lista. Cada que la función regresa False (o None), el elemento al que le fue aplicado la función es descartado.

Veamos esto en acción:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def numero_es_par(numero):

    if numero % 2 == 0:
        return True
    else:
        return False

list(filter(numero_es_par, numeros))
```

```
## [2, 4, 6, 8, 10]
```

Como nuestra función regresa True si el valor es par, nuestra lista resultante sólo tiene valores pares.

Veamos otro ejemplo:

```
def numero_es_mayor_a_5(numero):

    if numero > 5:
        return True

list(filter(numero_es_mayor_a_5, numeros))
```

```
## [6, 7, 8, 9, 10]
```

En este caso no agregamos el if else: return False porque Python asume que si una función no regresa nada ha regresado un None, que cuenta como False. Usamos entonces filter para quedarnos solamente con los valores que nos interesan de una lista.

Algunos ejemplos más:

```
def palabra_tiene_mas_de_5_caracteres(palabra):

    if len(palabra) > 5:
        return True

palabras = ["achicoria", "pasto", "sol", "loquillo", "moquillo", "sed", "pez", "jacaranda", "mil"]

list(filter(palabra_tiene_mas_de_5_caracteres, palabras))
```

```
## ['achicoria', 'loquillo', 'moquillo', 'jacaranda']
```

```
def numero_es_negativo(numero):

    if numero < 0:
        return True

numeros = [3, 5, -1, -7, -8, 4, -78, 5, -46, 56, 98, 9, -1, -2, -4]

list(filter(numero_es_negativo, numeros))
```

```
## [-1, -7, -8, -78, -46, -1, -2, -4]
```

```
def numero_es_divisible_entre_9(numero):

    if numero % 9 == 0:
        return True

numeros = [3, 7, 9, 34, 72, 90, 87, 34, 99, 56, 12, 18]

list(filter(numero_es_divisible_entre_9, numeros))
```

```
## [9, 72, 90, 99, 18]
```

RETO 2. FUNCIÓN FILTER

OBJETIVO

- Prácticar el uso de filter para filtrar los datos en una lista

DESARROLLO

A. LIMPIANDO DATOS NULOS

Debajo tenemos una lista que incluye datos acerca de las edades de las personas que han atendido a un curso de Cocina Medieval (ya sabes: puerco al horno, manzanas asadas, aguardiente, sangre fresca de tus enemigos). Algunas de las personas que atendieron no quisieron dar su edad. Es por eso que algunos de los elementos son None:

```
edades = [12, 16, 19, None, 21, 25, 24, None, None, 16, 17, 25, 23, 28, None, 23, 35, 59, 67, None, 34,
```

Queremos realizar una pequeña visualización (un histograma, que ya aprenderás a hacer más tarde) con nuestros datos. Pero no nos interesan los datos que vienen como None. Escribe una función llamada valor_no_es_none que reciba un valor, cheque si el valor es None, regrese False si el valor es None o regrese True si el valor no es None. Después úsala para filtrar tus datos:

```
def valor_no_es_none(valor):
    if valor==None:
      return False
    else:
      return True

edades_filtradas=list(filter(valor_no_es_none,edades))
edades_filtradas
```

```
## [12, 16, 19, 21, 25, 24, 16, 17, 25, 23, 28, 23, 35, 59, 67, 34, 21, 23, 15, 14, 18, 24, 23, 17]
```

Realizando el histograma tenemos lo siguiente:

```
library(reticulate)
py_install("seaborn")
```

```
def crear_histograma_con(datos):
    import seaborn as sns
    import numpy as np

    sns.distplot(datos, kde=False, bins=len(np.unique(datos)))

#CREANDO HISTOGRAMA
crear_histograma_con(edades_filtradas)
```

```
## C:\PROGRA~3\ANACON~1\envs\R-RETI~1\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `
##   warnings.warn(msg, FutureWarning)
```

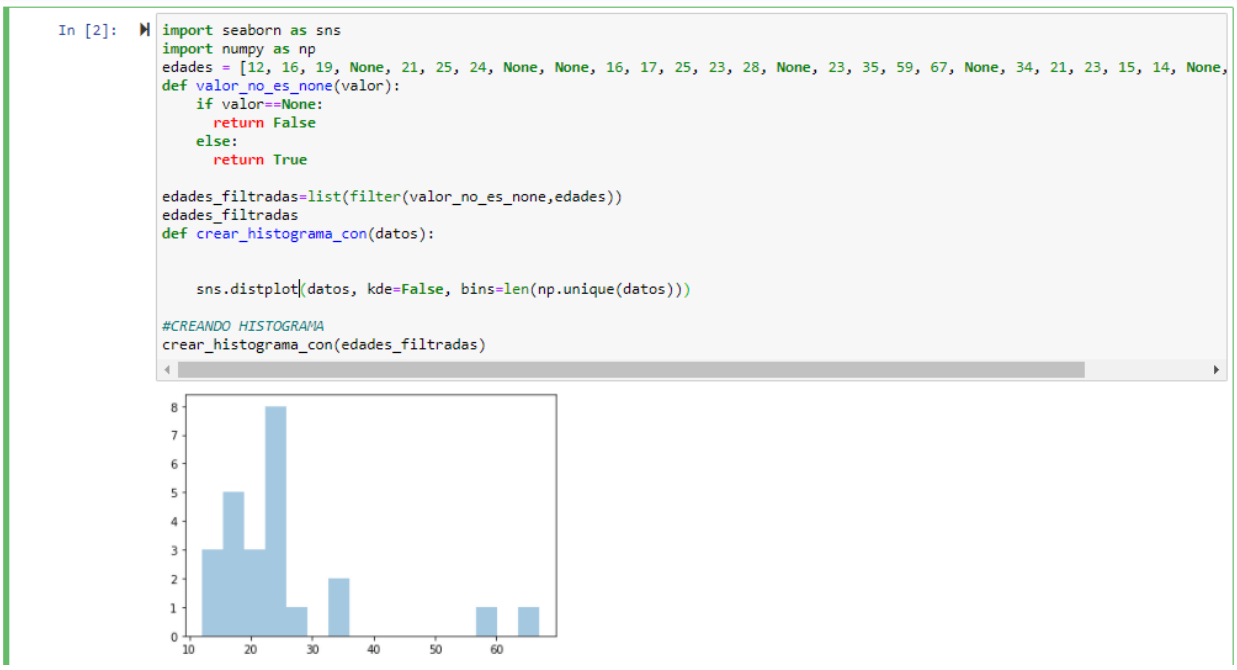

Figure 45: IMG

En R la librería de seaborn no se encuentra instalada de manera predeterminada, por ello se tiene que ejecutar en una JN:

Obteniendo el resultado anterior en cuanto a la graficación.

FILTRANDO DATOS ATIPICOS

Aquí tenemos una lista que contiene datos acerca de los sueldos (cada número representa “miles de pesos”) de los empleados de EyePoker Inc. (la empresa donde se producen los mejores picadores de ojos en todo el Hemisferio Occidental):

```
sueldos = [26, 32, 26, 30, 30, 32, 28, 30, 28, 110, 34, 30, 28, 26, 28, 30, 28, 85, 25, 30, 34, 34, 30,
```

En general todos los sueldos se encuentran en un rango bastante restringido, pero tenemos algunos datos sobre sueldos “anormalmente” grandes. Los sueldos tan grandes son los de los ejecutivos, que claramente no tienen ninguna noción de “justicia” (eso pasa cuando tus picadores de ojos son los mejores de todo el Hemisferio Occidental). Nosotros queremos usar el promedio para tener una idea de cuál es el sueldo típico en esta empresa. Nuestros valores atípicos (los sueldos anormalmente grandes) van a arruinar nuestro cálculo.

Mira cuál es el sueldo típico si no filtramos nuestros valores anormalmente grandes:

```
print(f'El sueldo "típico" en EyePoker Inc. es de {sum(sueldos) / len(sueldos)}')
```

```
## El sueldo "típico" en EyePoker Inc. es de 43.62068965517241
```

Para corregir esto haz una función llamada numero_es_menor_que_40 que descarte los números mayores de 40, y úsala para filtrar la lista sueldos, para tener un cálculo más apropiado del sueldo típico en esta empresa.

```
def numero_es_menor_que_40(numero):
  if numero>40:
    return numero

sueldos_filtrados=list(filter(numero_es_menor_que_40,sueldos))
sueldos_filtrados
```

```
## [110, 85, 120, 120, 125]
```

```
print(f'El sueldo "típico" en EyePoker Inc. es de {sum(sueldos_filtrados) / len(sueldos_filtrados)}')
```

```
## El sueldo "típico" en EyePoker Inc. es de 112.0
```

AND

OBJETIVOS

- Aprender a extender las capacidades de los operadores de comparación usando and
- Usar and para llamar filter con varios argumentos

DESARROLLO

Muchas veces una sola sentencia de comparación no va ser suficiente para filtrar los datos como queremos. En ese caso, and puede ayudarnos a unir dos sentencias. and regresa True cuando ambas sentencias regresan True.

Digamos que tenemos dos funciones que realizan una comparación y regresan True cuando la comparación se cumple:

FUNCION 1:

```
def numero_es_divisible_entre_3(numero):

    if numero % 3 == 0:
        return True
    else:
        return False
```

FUNCION 2:

```
def numero_es_menor_que_10(numero):

    if numero < 10:
        return True
    else:
        return False
```

Vamos a realizar algunas comparaciones usando ambas funciones para evaluar el mismo número:

```
#COMPARACION 1
numero_es_divisible_entre_3(9) and numero_es_menor_que_10(9)
#COMPARACION 2
```

```
## True
```

```
numero_es_divisible_entre_3(12) and numero_es_menor_que_10(12)
#COMPARACION 3
```

```
## False
```

```
numero_es_divisible_entre_3(8) and numero_es_menor_que_10(8)
#COMPARACION 4
```

```
## False
```

```
numero_es_divisible_entre_3(16) and numero_es_menor_que_10(16)
```

```
## False
```

Como puedes ver, and sólo regresa True cuando ambas comparaciones regresan True.

Veamos ahora cómo aplicarlo a un filter. Tenemos una lista que queremos filtrar:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

En esta ocasión vamos a construir una función que reúna ambas funciones que tenemos, porque filter sólo recibe una función (más adelante veremos otras opciones):

```
def numero_es_divisible_entre_3_y_menor_que_10(numero):

    return numero_es_divisible_entre_3(numero) and numero_es_menor_que_10(numero)
```

Ahora la aplicamos:

```
list(filter(numero_es_divisible_entre_3_y_menor_que_10, numeros))
```

```
## [3, 6, 9]
```

¡Genial! ¿No es así? Vayamos ahora a practicar esta herramienta.

RETO 3. FUNCION AND

OBJETIVOS

- Practicar el operador and y usarlo apra realizar filtros mas complejos

DESARROLLO

FILTRANDO VALORES ATIPICOS EN AMBOS EXTREMOS

Regresemos a nuestro ejemplo de EyePoker Inc. Esta vez tenemos un nuevo conjunto de datos con más empleados (la industria de picadores de ojos va en aumento vertiginoso). Además incluye los sueldos de algunos internos. Estos sueldos son muy bajos (simbólicos, podríamos llamarlos), como puedes ver:

```
sueldos = [26, 32, 26, 1.5, 30, 30, 1, 2, 32, 28, 30, 28, 30, 28, 27, 30, 110, 1.5, 2, 34, 30, 28,
26, 28, 2, 30, 28, 85, 25, 1.5,1.5, 30, 34, 34,30, 30, 120, 28, 2, 2, 1.5, 28, 120, 1, 1.5, 125, 2, 1.5
```

En realidad a nosotros sólo nos interesa analizar los sueldos de los empleados que tiene la empresa a largo plazo. Como tenemos bastantes internos, es muy probable que la inclusión de estos sueldos vaya a distorsionar nuestro cálculo del sueldo típico en la empresa:

```
print(f'El sueldo "típico" en EyePoker Inc. es de {sum(sueldos) / len(sueldos)}')
```

```
## El sueldo "típico" en EyePoker Inc. es de 25.36021505376344
```

Para evitar esta distorsión y calcular solamente el sueldo típico de los empleados que están contratados a largo plazo, vamos a filtrar nuestra lista.

Lo que tienes que hacer es lo siguiente:

1. Define una función que regrese True cuando el argumento sea mayor que 20.
2. Define una función que regrese True cuando el argumento sea menor que 40.
3. Define una tercera función que una las dos primeras funciones usando un operador and.
4. Filtrar la lista y asignarla a sueldos_filtrados.

```
def p1(numero):
  if numero>20:
    return True

def p2(numero):
  if(numero<40):
    return True

def p3(numero):
  if(numero>20 and numero<40):
    return True

sueldos_filtrados=list(filter(p3,sueldos))
sueldos_filtrados
```

```
## [26, 32, 26, 30, 30, 32, 28, 30, 28, 30, 28, 27, 30, 34, 30, 28, 26, 28, 30, 28, 25, 30, 34, 34, 30,
```

```
print(f'El sueldo "típico" en EyePoker Inc. es de {sum(sueldos_filtrados) / len(sueldos_filtrados)}')
```

```
## El sueldo "típico" en EyePoker Inc. es de 28.96078431372549
```

OR

or es muy parecido a and. También nos sirve para unir dos sentencias de comparacion y obtener un resultado. La diferencia es que or regresa TRUE cuando una de las dos o ambas comparaciones regresen TRUE

OBJETIVOS

- Aprender a utilizar las capacidades de los operadores de comparación utilizando or
- Usar or para llenar filter con multiples filtros

DESARROLLO

Vamos a usar las mismas funciones que definimos en nuestro ejemplo pasado:

PRIMERA FUNCIÓN:

```
def numero_es_divisible_entre_3(numero):

    if numero % 3 == 0:
        return True
    else:
        return False
```

SEGUNDA FUNCIÓN:

```
def numero_es_menor_que_10(numero):

    if numero < 10:
        return True
    else:
        return False
```

Pero ahora veamos que sucede cuando usamos or para unir dos comparaciones:

```
numero_es_divisible_entre_3(9) or numero_es_menor_que_10(9)
```

```
## True
```

```
numero_es_divisible_entre_3(12) or numero_es_menor_que_10(12)
```

```
## True
```

```
numero_es_divisible_entre_3(8) or numero_es_menor_que_10(8)
```

```
## True
```

```
numero_es_divisible_entre_3(16) or numero_es_menor_que_10(16)
```

```
## False
```

Como ves, or regresa True cuando una de las dos comparaciones regresa True o cuando ambas regresan True. En cambio, sólo regresa False cuando ambas comparaciones regresan False.

Veamos ahora qué sucede si lo aplicamos a la misma lista de la vez pasada:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
def numero_es_divisible_entre_3_o_menor_que_10(numero):

    return numero_es_divisible_entre_3(numero) or numero_es_menor_que_10(numero)
```

```
list(filter(numero_es_divisible_entre_3_o_menor_que_10, numeros))
```

```
## [1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 15, 18]
```

RETO 4. FUNCIÓN OR

OBJETIVOS

- Prácticar el operador or y usarlo para realizar filtros más complejos

DESARROLLO

FILTRANDO PALABRAS

Eres el organizador del Concurso Nacional de Deletreo "Salvador Novo". Por una bella coincidencia, este año el día del concurso cae justo el mismo día que el Día del Orgullo LGBT. Dado que Salvador Novo era homosexual, te parece muy apropiado que el concurso de deletreo funcione como una celebración de su "belicosa homosexualidad" (como la llamaba Carlos Monsiváis). Se te ocurre hacer lo siguiente:

De la lista de palabras que tenías originalmente para el concurso, vas a filtrar las palabras para que sólo tengas palabras que empiecen con "l" o o con "g" o con "b" o con "t".

Aquí está tu lista original de palabras:

```
palabras = ['cabildo', 'genocidio', 'severo', 'jarana', 'enigmático','jaguar', 'solidaridad', 'reivindi
```

INSTRUCCIONES

1. Escribe 4 funciones, para cada una de las letras del acrónimo LGBT. Las funciones van a regresar True sólo si la palabra comienza con la letra que le corresponde. Por ejemplo, la función palabra_comienza_con_l va a regresar True sólo si la palabra comienza con l.
2. Después, define una función que sea la unión de estas 4 funciones y regrese True si la palabra comienza con alguna de las letras del acrónimo LGBT.
3. Finalmente filtra la lista palabras para tener una nueva lista que será la lista usada para el concurso.

Tip #1: Las strings pueden ser accedas igual que las listas, así que si quieres acceder a la primera letra de una palabra basta con usar palabra[0], como si fuera el primer índice de una lista.

Tip #2: Hasta ahora sólo hemos usando operadores lógicos con 1 o 2 comparaciones. Juntar más de dos comparaciones es tan fácil como escribir:

```
comparacion_1 or comparacion_2 or comparacion_3 or comparacion_4
```

FUNCION PARA FILTRAR PALABRAS QUE INICIEN CON 'l':

```
def funl(lista):
  if lista[0]=="l":
    return True
  else:
    return False
```

Aplicandolo a la lista tenemos lo siguiente:

```
list(filter(funl,palabras))
```

```
## ['letargo', 'legislar', 'lloriquear', 'libélula', 'llovizna']
```

Funcion para filtrar G

```
def fung(lista):
  if lista[0]=="g":
    return True
  else:
    return False
```

Aplicandolo a la lista tenemos lo siguiente:

```
list(filter(fung,palabras))
```

```
## ['genocidio', 'gnomo', 'gargantilla']
```

Realizando la funcion de B

```
def funb(lista):
  if lista[0]=="b":
    return True
  else:
    return False
```

Aplicandolo a la lista de palabras:

```
list(filter(funb,palabras))
```

```
## ['bálsamo', 'boicotear', 'blasfemia', 'bóveda', 'basílica']
```

Haciendo el filtro de la T:

```
def funt(lista):
  if lista[0]=="t":
    return True
  else:
    return False
```

Aplicandolo a la lista de palabras:

```
list(filter(funt,palabras))
```

```
## ['tentáculo', 'tecnicismo', 'terraplén']
```

Creando una funcion con todos los datos necesarios para filtrar todas las que inicien con letras LGBT:

```
def fun_comple(letras):
  return funl or fung or funb or funt
```

Aplicando a mi lista de palabras la funcion completa tenemos lo siguiente:

```
palabras_filtradas=list(filter(fun_comple,palabras))

print(f'==Concurso Nacional de Deletreo "Salvador Novo"==\n')
```

```
## ==Concurso Nacional de Deletreo "Salvador Novo"==
```

```
print(f'Lista oficial de palabras: {palabras_filtradas}')
```

```
## Lista oficial de palabras: ['cabildo', 'genocidio', 'severo', 'jarana', 'enigmático', 'jaguar', 'soli
```

EJEMPLO 5. FUNCION NOT

OBJETIVOS

- Aprender a extender las capaciaddes de los operadores de comparación usando not

DESARROLLO

not es muy sencillo, así que simplemente vamos a ver cómo funciona usando una de nuestras funciones anteriores. not recibe un sólo booleano, así que por el momento sólo podremos usar una función al mismo tiempo:

```
def numero_es_divisible_entre_3(numero):

    if numero % 3 == 0:
        return True
    else:
        return False
```

```
not(numero_es_divisible_entre_3(9))
```

```
## False
```

```
not(numero_es_divisible_entre_3(10))
```

```
## True
```

¿Ves? not simplemente regresa True cuando la comparación es False y viceversa.

Obviamente esto sólo es útil si podemos aplicarlo a un filter. Pero para hacer eso tendremos primero que aprender sobre funciones lambda.

EJEMPLO 6. FUNCIONES LAMBDA

OBJETIVOS

- Aprender la sintaxis lambda para poderla aplicar en el map y el filter

DESARROLLO

Una función lambda se define así:

```
lambda x: x*100
```

```
## <function <lambda> at 0x000000002F401730>
```

Se usa la palabra lambda, luego se definen los parámetros, y al final se agrega el cuerpo de la función, que en este caso sólo puede incluir una sola sentencia: la sentencia return. No hace falta escribir return, lambda sabe que tiene que regresar la única línea de código que tiene.

Ahora veamos cómo se usaría para revertir nuestra comparación anterior en un filter.

```
def numero_es_divisible_entre_3(numero):

    if numero % 3 == 0:
        return True
    else:
        return False
```

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
list(filter(lambda x: not numero_es_divisible_entre_3(x), numeros))
```

```
## [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20]
```

Como puedes ver, esta sentencia nos regresa todos los números que **no** son divisibles entre 3. Revierte el funcionamiento de la función `numero_es_divisible_entre_3`. Podríamos hacerlo de esta manera, pero requiere de más código:

```
def numero_no_es_divisible_entre_3(numero):

    if not numero_es_divisible_entre_3(numero):
        return True
    else:
        return False
```

Usar una u otra de las opciones dependerá del contexto y de tu criterio. Veamos un par de `lambdas` más:

```
palabras = ["achicoria", "pasto", "sol", "loquillo", "moquillo", "sed", "pez", "jacaranda", "mil"]

list(filter(lambda x: len(x) > 5, palabras))
```

```
## ['achicoria', 'loquillo', 'moquillo', 'jacaranda']
```

```
numeros = [3, 5, -1, -7, -8, 4, -78, 5, -46, 56, 98, 9, -1, -2, -4]

list(filter(lambda x: x < 0, numeros))
```

```
## [-1, -7, -8, -78, -46, -1, -2, -4]
```

```
list(filter(lambda x: not(x < 0), numeros))
```

```
## [3, 5, 4, 5, 56, 98, 9]
```

RETO FINAL DE LA SESION 03

OBJETIVOS

- Practicar la sintaxis y uso ed las funciones lambda para poder aplicarlas a filter

DESARROLLO

CONTEO DE VOTOS

Eres el líder estudiantil de la H. Universidad Unida de Las Américas (sí, todos los países de América del Norte y América del Sur se han unido en un sólo país llamado Las Américas; ¡yei por la disolución de las fronteras!). Acabas de realizar una votación para decidir el Proyecto Comunitario que realizarán en conjunto todos los estudiantes de la universidad en el próximo año escolar. Las 2 opciones fueron:

2. Ética en Inteligencia Artificial (Código: AI)
3. Cambio Climático (Código: CC)

Los resultados de la votación fueron los siguientes:

```
votos = ['AI', 'CC', 'CC', 'CC', 'CC', 'CC', 'CC', 'CC', 'CC', 'CC', 'CC', 'CC', 'CC', 'AI', 'CC', 'CC'
```

¡Ha llegado el gran momento de contar los votos! Tu reto es el siguiente:

1. **Crea una función llamada voto_por_ai que regrese True si el voto fue "AI". Usa esa función para filtrar tus votos y asigna ese resultado a una variable llamada votos_por_ai.**
2. **Usando esa misma función, utiliza una función lambda y el operador not para filtrar de nuevo la lista votos y obtener una nueva lista llamada votos_por_cc.**

```
def voto_por_ai(lista):
  return lista=="AI"


votos_por_ai=list(filter(voto_por_ai,votos))
votos_por_cc=list(filter(lambda elemento: not voto_por_ai(elemento),votos))
```

Corre la siguiente celda para obtener el número total de votos y saber cuál fue el proyecto ganador:

```
print(f'== Resultados de la votación para el Proyecto Comunitario 2025 ==\n')
print(f'{("Proyecto"):25}   Conteo')
print(f'----------------------------------------')
print(f'{("- Ética en AI"):25} | {len(votos_por_ai)}')
print(f'{("- Cambio Climático"):25} | {len(votos_por_cc)}')
print(f'----------------------------------------')
print(f'{("- Total"):25} | {len(votos_por_ai) + len(votos_por_cc)}')
```

```
== Resultados de la votación para el Proyecto Comunitario 2025 ==

Proyecto                   Conteo

----------------------------------------

- Ética en AI              | 123

- Cambio Climático         | 323

----------------------------------------

- Total                    | 446
```