Sets

Fuente: https://recursospython.com

Un conjunto es una colección no ordenada de objetos únicos. Python provee este tipo de datos «por defecto» al igual que otras colecciones más convencionales como las listas, tuplas y diccionarios.

Los conjuntos son ampliamente utilizados en lógica y matemática, y desde el lenguaje podemos sacar provecho de sus propiedades para crear código más eficiente y legible en menos tiempo.

Creación de un conjunto

Para crear un conjunto especificamos sus elementos entre llaves:

```
s = {1, 2, 3, 4}
```

Al igual que otras colecciones, sus miembros pueden ser de diversos tipos:

```
>>> s = {True, 3.14, None, False, "Hola mundo", (1, 2)}
```

No obstante, un conjunto no puede incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos.

```
>>> s = {[1, 2]}
Traceback (most recent call last):
 ...
TypeError: unhashable type: 'list'
```

Python distingue este tipo de operación de la creación de un diccionario ya que no incluye dos puntos. Sin embargo, no puede dirimir el siguiente caso:

```
s = {}
```

Por defecto, la asignación anterior crea un diccionario. Para generar un conjunto vacío, directamente creamos una instancia de la clase set:

```
s = set()
```

De la misma forma podemos obtener un conjunto a partir de cualquier objeto iterable:

```
s1 = set([1, 2, 3, 4])
s2 = set(range(10))
```

Un *set* puede ser convertido a una lista y viceversa. En este último caso, los elementos duplicados son unificados.

```
>>> list({1, 2, 3, 4})
[1, 2, 3, 4]
>>> set([1, 2, 2, 3, 4])
{1, 2, 3, 4}
```

Elementos

Los conjuntos son objetos mutables. Vía los métodos add() y discard() podemos añadir y remover un elemento indicándolo como argumento.

```
>>> s = {1, 2, 3, 4}
>>> s.add(5)
>>> s.discard(2)
>>> s
{1, 3, 4, 5}
```

Nótese que si el elemento pasado como argumento a discard() no está dentro del conjunto es simplemente ignorado. En cambio, el método remove() opera de forma similar pero en dicho caso lanza la excepción KeyError.

Para determinar si un elemento pertenece a un conjunto, utilizamos la palabra reservada in.

```
>>> 2 in {1, 2, 3}
True
>>> 4 in {1, 2, 3}
False
```

La función clear() elimina todos los elementos.

```
>>> s = {1, 2, 3, 4}
>>> s.clear()
>>> s
set()
```

El método pop() retorna un elemento en forma aleatoria (no podría ser de otra manera ya que los elementos no están ordenados). Así, el siguiente bucle imprime y remueve uno por uno los miembros de un conjunto.

```
while s:
  print(s.pop())
```

remove() y pop() lanzan la excepción KeyError cuando un elemento no se encuentra en el conjunto o bien éste está vacío, respectivamente.

Para obtener el número de elementos aplicamos la ya conocida función len():

```
>>> len({1, 2, 3, 4})
4
```

Operaciones principales

Algunas de las propiedades más interesantes de los conjuntos radican en sus operaciones principales: *unión*, *intersección* y *diferencia*.

La unión se realiza con el caracter | y retorna un conjunto que contiene los elementos que se encuentran en al menos uno de los dos conjuntos involucrados en la operación.

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5, 6}
>>> a | b
{1, 2, 3, 4, 5, 6}
```

La intersección opera de forma análoga, pero con el operador &, y retorna un nuevo conjunto con los elementos que se encuentran en ambos.

```
>>> a & b
{3, 4}
```

La diferencia, por último, retorna un nuevo conjunto que contiene los elementos de a que no están en b.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3}
>>> a - b
{1, 4}
```

Dos conjuntos son iguales si y sólo si contienen los mismos elementos (a esto se lo conoce como *principio de extensionalidad*):

```
>>> {1, 2, 3} == {3, 2, 1}
True
>>> {1, 2, 3} == {4, 5, 6}
False
```

Otras operaciones

Se dice que B es un *subconjunto* de A cuando todos los elementos de aquél pertenecen también a éste. Python puede determinar esta relación vía el método issubset().

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3}
>>> b.issubset(a)
True
```

Inversamente, se dice que A es un *superconjunto* de B.

```
>>> a.issuperset(b)
True
```

La definición de estas dos relaciones nos lleva a concluir que todo conjunto es al mismo tiempo un subconjunto y un superconjunto de sí mismo.

```
>>> a = {1, 2, 3, 4}
>>> a.issubset(a)
True
>>> a.issuperset(a)
True
```

La *diferencia simétrica* retorna un nuevo conjunto el cual contiene los elementos que pertenecen a alguno de los dos conjuntos que participan en la operación pero no a ambos. Podría entenderse como una unión exclusiva.

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5, 6}
>>> a.symmetric_difference(b)
{1, 2, 5, 6}
```

Dada esta definición, se infiere que es indistinto el orden de los objetos:

```
>>> b.symmetric_difference(a)
{1, 2, 5, 6}
```

Por último, se dice que un conjunto es *disconexo* respecto de otro si no comparten elementos entre sí.

```
>>> a = {1, 2, 3}
>>> b = {3, 4, 5}
>>> c = {5, 6, 7}
>>> a.isdisjoint(b)
False  # No son disconexos ya que comparten el elemento 3.
>>> a.isdisjoint(c)
True   # Son disconexos.
```

En otras palabras, dos conjuntos son disconexos si su intersección es el conjunto vacío, por lo que puede ilustrarse de la siguiente forma:

```
>>> def isdisjoint(a, b):
...     return a & b == set()
...
>>> isdisjoint(a, b)
False
>>> isdisjoint(a, c)
True
```

Conjuntos inmutables

frozenset es una implementación similar a set pero inmutable. Es decir, comparte todas las operaciones de conjuntos provistas en este artículo a excepción de aquellas que implican alterar sus elementos (add(), discard(), etc.). La diferencia es análoga a la existente entre una lista y una tupla.

```
>>> a = frozenset({1, 2, 3})
>>> b = frozenset({3, 4, 5})
>>> a & b
frozenset({3})
>>> a | b
```

```
frozenset({1, 2, 3, 4, 5})
>>> a.isdisjoint(b)
False
```

Esto permite, por ejemplo, emplear conjuntos como claves en los diccionarios:

```
>>> a = {1, 2, 3}
>>> b = frozenset(a)
>>> {a: 1}
Traceback (most recent call last):
 ...
TypeError: unhashable type: 'set'
>>> {b: 1}
{frozenset({1, 2, 3}): 1}
```

Más información: https://hetpro-store.com/TUTORIALES/set-en-python-7-colecciones/#:~:text=Un%20set%20es%20una%20colección,sets%20se%20escriben%20entre%20llaves.

Concepto de funciones - Programación estructurada

Fuente: https://www.tutorialesprogramacionya.com

Hasta ahora hemos trabajado con una metodología de programación lineal. Todas las instrucciones de nuestro archivo *.py se ejecutan en forma secuencial de principio a fin. Esta forma de organizar un programa solo puede ser llevado a cabo si el mismo es muy pequeño (decenas de líneas)

Cuando los problemas a resolver tienden a ser más grandes la metodología de programación lineal se vuelve ineficiente y compleja.
El segundo paradigma de programación que veremos es la programación estructurada.

La programación estructurada busca dividir o descomponer un problema complejo en pequeños problemas. La solución de cada uno de esos pequeños problemas nos trae la solución del problema complejo.
En Python el planteo de esas pequeñas soluciones al problema complejo se hace dividiendo el programa en funciones.
Una función es un conjunto de instrucciones en Python que resuelven un problema específico.

El lenguaje Python ya tiene incorporadas algunas funciones básicas. Algunas de ellas ya las utilizamos en conceptos anteriores como son las funciones: **print**, **len** y **range**.
Veamos ahora como crear nuestras propias funciones.
El tema de funciones en un principio puede presentar dificultades para entenderlo y ver sus ventajas ante la metodología de programación lineal que veníamos trabajando en conceptos anteriores.

Para los primeros problemas que presentaremos nos puede parecer que sea más conveniente utilizar programación lineal en vez de programación estructurada por funciones. A medida que avancemos veremos que si un programa empieza a ser más complejo (cientos de líneas, miles de líneas o más) la división en pequeñas funciones nos permitirá tener un programa más ordenado y fácil de entender y por lo tanto en mantener.

Problema 1:

Confeccionar una aplicación que muestre una presentación en pantalla del programa. Solicite la carga de dos valores y nos muestre la suma. Mostrar finalmente un mensaje de despedida del programa.
Implementar estas actividades en tres funciones.

La forma de organizar nuestro programa cambia en forma radical.
El programa ahora no empieza a ejecutarse en la línea 1.
El programa principal comienza a ejecutarse luego del comentario "programa principal":

Primero declaramos las tres funciones llamadas **presentacion**, **carga_suma** y **finalizacion**. La sintaxis para declarar una función es mediante la palabra clave **def** seguida por el nombre de la función (el nombre de la función no puede tener espacios en blanco, comenzar con un número y el único caracter especial permitido es el **_**)

Luego del nombre de la función deben ir entre paréntesis los datos que llegan, si no llegan datos como es el caso de nuestras tres funciones solo se disponen paréntesis abierto y cerrado. Al final disponemos los **:**
Todo el bloque de la función se indenta cuatro espacios como venimos trabajando cuando definimos estructuras condicionales o repetitivas.
Dentro de una función implementamos el algoritmo que pretendemos que resuelva esa función, por ejemplo, la función **presentacion** tiene por objetivo mostrar en pantalla el objetivo del programa:

Si no hacemos las llamadas a las funciones los algoritmos que implementan las funciones nunca se ejecutarán.
Cuando en el bloque del programa principal se llama una función hasta que no finalice no continua con la llamada a la siguiente función:

Problema 2:

Confeccionar una aplicación que solicite la carga de dos valores enteros y muestre su suma.
Repetir la carga e impresión de la suma 5 veces.
Mostrar una línea separadora después de cada vez que cargamos dos valores y su suma.

Hemos declarado dos funciones, una que permite cargar dos enteros sumarlos y mostrar el resultado.

Ahora nuestro bloque principal del programa, recordemos que estas líneas son las primeras que se ejecutarán cuando iniciemos el programa:

```
# programa principal

for x in range(5):
    carga_suma()
    separacion()
```

Como vemos podemos llamar a la función carga_suma() y separación() muchas veces en nuestro caso en particular 5 veces.
Lo nuevo que debe quedar claro es que la llamada a las funciones desde el bloque principal de nuestro programa puede hacerse múltiples veces (esto es lógico, recordemos que print es una función ya creada en Python y la llamamos múltiples veces dentro de nuestro algoritmo)

Funciones: parámetros

Vimos en el concepto anterior que una función resuelve una parte de nuestro algoritmo. Tenemos por un lado la declaración de la función por medio de un nombre y el algoritmo de la función seguidamente. Luego para que se ejecute la función la llamamos desde el bloque principal de nuestro programa.
Ahora veremos que una función puede tener parámetros para recibir datos. Los parámetros nos permiten comunicarle algo a la función y la hace más flexible.

Problema 1:

Confeccionar una aplicación que muestre una presentación en pantalla del programa. Solicite la carga de dos valores y nos muestre la suma. Mostrar finalmente un mensaje de despedida del programa.

Ahora para resolver este pequeño problema hemos planteado una función llamada **mostrar_mensaje** que recibe como parámetro un string (cadena de caracteres) y lo muestra en pantalla.
Los parámetros van seguidos del nombre de la función encerrados entre paréntesis (y en el caso de tener más de un parámetro los mismos deben ir separados por coma):

Un parámetro podemos imaginarlo como una variable que solo se puede utilizar dentro de la función.
Ahora cuando llamamos a la función mostrar_mensaje desde el bloque principal de nuestro programa debemos pasar una variable string o un valor de tipo string:
mostrar_mensaje("El programa calcula la suma de dos valores ingresados por teclado.")

El string que le pasamos: "El programa calcula la suma de dos valores ingresados por teclado." lo recibe el parámetro de la función.
Una función con parámetros nos hace más flexible la misma para utilizarla en distintas circunstancias. En nuestro problema la función mostrar_mensaje la utilizamos tanto para la presentación inicial de nuestro programa como para mostrar el mensaje de despedida. Si no existieran los parámetros estaríamos obligados a implementar dos funciones como el concepto anterior.

Problema 2:

Confeccionar una función que reciba tres enteros y nos muestre el mayor de ellos. La carga de los valores hacerlo por teclado.

Es importante notar que un programa en Python no ejecuta en forma lineal las funciones definidas en el archivo *.py sino que arranca en la zona del bloque principal. En nuestro ejemplo se llama primero a la función "**cargar()**", esta función no tiene parámetros.
La función cargar solicita el ingreso de tres enteros por teclado y llama a la función mostrar_mayor y les pasa a sus parámetros las tres variables enteras valor1, valor2 y valor3.
La función mostrar_mayor recibe en sus parámetros v1, v2 y v3 los valores cargados en las variables valor1, valor2 y valor3.

Los parámetros son la forma que nos permite comunicar la función cargar con la función mostrar_mayor.
Dentro de la función mostrar_mayor no podemos acceder a las variable valor1, valor2 y valor3 ya que son variables locales de la función cargar.
Otra cosa importante notar que en la sección del programa principal solo llamamos a la función cargar, es decir que en esta zona no es obligatorio llamar a cada una de las funciones que definimos.

Problema 3:

Desarrollar un programa que permita ingresar el lado de un cuadrado. Luego preguntar si quiere calcular y mostrar su perímetro o su superficie.

Definimos dos funciones que calculan y muestran el perímetro por un lado y por otro la superficie:

La tercera función permite cargar el lado del cuadrado e ingresar un string que indica que cálculo deseamos realizar si obtener el perímetro o la superficie. Una vez que se ingresó la variable respuesta procedemos a llamar a la función que efectúa el cálculo respectivo pasando como dato la variable local "la" que almacena el valor del lado del cuadrado.
Los parámetros son la herramienta fundamental para pasar datos cuando hacemos la llamada a una función.

Funciones: retorno de datos

Hemos comenzado a pensar con la metodología de programación estructurada. Buscamos dividir un problema en subproblemas y plantear algoritmos en Python que los resuelvan.
Vimos que una función la definimos mediante un nombre y que puede recibir datos por medio de sus parámetros.
Los parámetros son la forma para que una función reciba datos para ser procesados.
Ahora veremos otra característica de las funciones que es la de devolver un dato a quien invocó la función (recordemos que una función la podemos llamar desde el bloque principal de nuestro programa o desde otra función que desarrollemos)

Problema 1:

Confeccionar una función que le enviemos como parámetro el valor del lado de un cuadrado y nos retorne su superficie.

Aparece una nueva palabra clave en Python para indicar el valor devuelto por la función:
return
La función retornar_superficie recibe un parámetro llamado lado, definimos una variable local llamada sup donde almacenamos el producto del parámetro lado por sí mismo.
La variable local sup es la que retorna la función mediante la palabra clave return:

Hay que tener en cuenta que las variables locales (en este caso sup) solo se pueden consultar y modificar dentro de la función donde se las define, no se tienen acceso a las mismas en el bloque principal del programa o dentro de otra función.

Problema 2:

Confeccionar una función que le enviemos como parámetros dos enteros y nos retorne el mayor.

Cuando una función encuentra la palabra return no sigue ejecutando el resto de la función sino que sale a la línea del programa desde donde llamamos a dicha función.

Funciones: parámetros de tipo lista

Hasta ahora hemos resuelto problemas enviando datos simples como enteros, float y cadenas de caracteres. En este concepto veremos que una función puede recibir tanto datos simples como estructuras de datos.
La estructura de datos vista hasta este momento y que podemos enviarle a una función es la lista.

Problema 1:

Definir por asignación una lista de enteros en el bloque principal del programa. Elaborar tres funciones, la primera recibe la lista y retorna la suma de todos sus elementos, la segunda recibe la lista y retorna el mayor valor y la última recibe la lista y retorna el menor.

La función sumarizar recibe un parámetro de tipo lista, dentro de la misma lo recorremos mediante un for y accedemos a sus elementos por medio de un subíndice.

Desde el bloque principal de nuestro programa llamamos a sumarizar enviando el dato retornado a la función print para que lo muestre.

Problema 2:

Crear y cargar por teclado en el bloque principal del programa una lista de 5 enteros. Implementar una función que imprima el mayor y el menor valor de la lista.

En el bloque principal de nuestro programa cargamos dentro de un for 5 valores enteros y los añadimos a la estructura de datos "lista".
Fuera del for procedemos a llamar a la función mayormenor y procedemos a enviarle la lista para que pueda consultar sus datos.
Es importante notar que la función mayormenor no retorna nada sino que ella es la que tiene el objetivo de mostrar el mayor y menor valor de la lista.

Funciones: retorno de una lista

Hemos avanzado y visto que una función puede recibir como parámetros tipos de datos simples como enteros, flotantes etc. y estructuras de datos tipo lista.

También hemos visto que la función mediante la palabra clave return puede retornar un tipo de dato simple desde donde se la invocó.
Lo nuevo en este concepto es que una función también puede retornar una estructura de datos tipo lista. Con esto estamos logrando que una función retorne varios datos ya que una lista es una colección de datos.

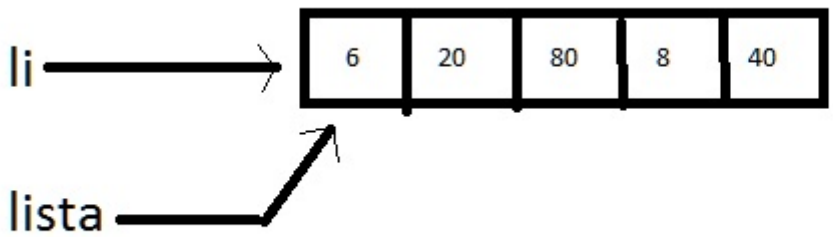
Problema 1:

Confeccionar una función que cargue por teclado una lista de 5 enteros y la retorne. Una segunda función debe recibir una lista y mostrar todos los valores mayores a 10. Desde el bloque principal del programa llamar a ambas funciones.

Recordemos que el programa empieza a ejecutarse desde el bloque principal y no se ejecutan cada función en forma lineal. Lo primero que hacemos en el bloque principal del programa es llamar a la función carga_lista y se la asignamos a una variable que recibirá la referencia de la función que se creará en carga_lista.

Dentro de la función definimos una variable local de tipo lista llamada li (el nombre de la variable local puede ser cualquiera, inclusive se podría llamar lista ya que no hay conflictos en definir variables con el mismo nombre en una función y en el bloque principal).

Cuando finaliza el for ya tenemos almacenado en la lista los 5 enteros y mediante la palabra clave de Python return procedemos a devolver a quien invocó la función la variable "li".
Esto quiere decir que en el bloque principal del programa lista recibe la referencia de "li".



Si no hacemos esa asignación la lista que devuelve la función carga_lista se perdería.

Problema 2:

Confeccionar una función que cargue por teclado una lista de 5 enteros y la retorne. Una segunda función debe recibir una lista y retornar el mayor y el menor valor de la lista. Desde el bloque principal del programa llamar a ambas funciones e imprimir el mayor y el menor de la lista.

Lo interesante que tenemos que sacar de este ejemplo es que cuando tenemos que retornar varios valores podemos utilizar una lista. La función retornar_mayormenor debe devolver dos enteros, y la forma de hacerlo en Python es mediante una lista.
En la función retornar_mayormenor definimos dos variables locales donde almacenamos el primer elemento de la lista que recibimos como parámetro.

Luego mediante un for procedemos a analizar el resto de los elementos de la lista para verificar si hay alguno que sea mayor al que hemos considerado mayor hasta ese momento, lo mismo con el menor.

Cuando sale del for ya tenemos almacenadas en las variables **ma** y **me** el valor mayor y el valor menor de la lista.
La sintaxis para devolver ambos datos es crear una lista e indicando en cada elemento la variable respectiva.

Ahora cuando llamo a esta función desde el bloque principal del programa sabemos que nos retorna una lista y que el primer elemento representa el mayor de la lista y el segundo elemento representa el menor de la lista.

Problema 3:

Desarrollar un programa que permita cargar 5 nombres de personas y sus edades respectivas. Luego de realizar la carga por teclado de todos los datos imprimir los nombres de las personas mayores de edad (mayores o iguales a 18 años)
Imprimir la edad promedio de las personas.

Para resolver este problema debemos crear y cargar dos listas paralelas. En una guardamos los nombres de las personas y en la otra almacenamos las edades.
Definimos una función cargar_datos que crea y carga las dos listas paralelos. Las variables locales que almacenan dichas listas son nom y ed.

Funciones: con parámetros con valor por defecto

En Python se pueden definir parámetros y asignarles un dato en la misma cabecera de la función. Luego cuando llamamos a la función podemos o no enviarle un valor al parámetro. Los parámetros por defecto nos permiten crear funciones más flexibles y que se pueden emplear en distintas circunstancias.

Problema 1:

Confeccionar una función que reciba un string como parámetro y en forma opcional un segundo string con un caracter. La función debe mostrar el string subrayado con el caracter que indica el segundo parámetro

Importante

Los parámetros por defecto deben ser los últimos que se declaren en la función. Se genera un error sintáctico si tratamos de definir una función indicando primero el o los parámetros con valores por defecto:

Funciones: llamada a la función con argumentos nombrados

Esta característica de Python nos permite llamar a la función indicando en cualquier orden los parámetros de la misma, pero debemos especificar en la llamada el nombre del parámetro y el valor a enviarle.

Problema 1:

Confeccionar una función que reciba el nombre de un operario, el pago por hora y la cantidad de horas trabajadas. Debe mostrar su sueldo y el nombre. Hacer la llamada de la función mediante argumentos nombrados.

Importante

Ahora vamos a profundizar la función print que hemos utilizado desde los primeros conceptos.
Como hemos trabajado hasta ahora cada vez que se llama a la función print se muestran todos los datos que le enviamos separados por coma y provoca un salto de línea al final.
Por ejemplo si ejecutamos

```
print("uno")
print("dos")
```

En pantalla aparece:

```
uno
dos
```

La función print tiene un parámetro llamado end, también hay que tener en cuenta que este parámetro por defecto tiene asignado "\n" que es el salto de línea y es por eso que cada vez que se ejecuta un print se produce un salto de línea.
Podemos indicar al parámetro end otro valor, por ejemplo un guión:

```
print("uno",end="-")
print("dos")
```

Ahora el resultado de ejecutar este programa es:

```
uno-dos
```

Ahora vemos que al llamar a print pidiendo que imprima "uno" e indicando que en lugar de producir un salto de línea muestre un guión:

```
print("uno",end="-")
```

Funciones: con cantidad variable de parámetros

Otra variante en la declaración de una función en Python es la definición de una cantidad variable de parámetros.
Para definir una cantidad variable de parámetros debemos anteceder el caracter asterísco (*) al último parámetro de la función.

Problema 1:

Confeccionar una función que reciba entre 2 y n (siendo n = 2,3,4,5,6 etc.) valores enteros, retornar la suma de dichos parámetros.

Para este problema definimos tres parámetros en la función,el primeros y el segundo reciben enteros y el tercero recibe una tupla (por ahora pensemos que una tupla es lo mismo que una lista, más adelante veremos sus diferencias):

```
def sumar(v1,v2,*lista):
    suma=v1+v2
    for x in range(len(lista)):
        suma=suma+lista[x]
    return suma
```

Sumamos los dos primeros valores y luego recorremos la lista y también sumamos sus elementos.
Cuando llamamos a la función sumar podemos hacerlo enviando solo dos parámetros (la lista está vacía en este caso):

```
print(sumar(1,2))
```

Podemos llamar la función enviando 4 parámetros, en este caso la lista tiene dos elementos:

```
print(sumar(1,2,3,4))
```

Y en general podemos llamar la función enviando cualquier cantidad de enteros:

```
print(sumar(1,2,3,4,5,6,7,8,9,10))
```

Funciones: Más tipos de parámetros

Veremos con un ejemplo cómo podemos pasar como parámetro una lista a una función y posteriormente cambiar su contenido y esto se vea reflejado en la variable que le enviamos al llamarla.

Problema 1:

Confeccionar un programa que contenga las siguientes funciones:
1) Carga de una lista y retorno al bloque principal.
2) Fijar en cero todos los elementos de la lista que tengan un valor menor a 10.
3) Imprimir la lista

La primer función permite la carga de una lista de enteros hasta que el operador no desee cargar más valores:

```
def cargar():
    lista=[]
    continua="s"
    while continua=="s":
        valor=int(input("Ingrese un valor:"))
        lista.append(valor)
        continua=input("Agrega otro elemento a la lista[s/n]:")
```

```
    return lista
```

Lo nuevo aparece en la función fijar_cero que recibe como parámetro una lista llamada "li" y dentro de la función modificamos los elementos de la lista, estos cambios luego se ven reflejados en la variable definida en el bloque principal de nuestro programa:

```
def fijar_cero(li):
    for x in range(len(li)):
        if li[x]<10:
            li[x]=0
```

Si ejecutamos este programa e ingresamos algunos elementos de la lista con valores inferiores a 10 veremos luego que la variable global "lista" es modificada:

Expresiones Lambda

Fuente: freecodecamp.org

Las expresiones lambda se usan idealmente cuando necesitamos hacer algo simple y estamos más interesados en hacer el trabajo rápidamente en lugar de nombrar formalmente la función. Las expresiones lambda también se conocen como **funciones anónimas**.
Las expresiones lambda en Python son una forma corta de declarar funciones pequeñas y anónimas (no es necesario proporcionar un nombre para las funciones lambda).

Las funciones Lambda se comportan como funciones normales declaradas con la palabra clave **def**. Resultan útiles cuando se desea definir una función pequeña de forma concisa. Pueden contener solo una expresión, por lo que no son las más adecuadas para funciones con instrucciones de flujo de control.

Sintaxis de una función Lambda

```
lambda argumentos: expresión
```

Las funciones Lambda pueden tener cualquier número de argumentos, pero solo una expresión.

Código de ejemplo:

```
# Función Lambda para calcular el cuadrado de un número
square = lambda x: x ** 2
print(square(3)) # Resultado: 9

# Funcion tradicional para calcular el cuadrado de un numero
def square(num):
  return num ** 2
print(square(5)) # Resultado: 25
```

En el ejemplo de lambda anterior, lambda x: x ** 2 produce un objeto de función anónimo que se puede asociar con cualquier nombre. Entonces, asociamos el objeto de función con square. De ahora en adelante, podemos llamar al objeto square como cualquier función tradicional, por ejemplo, square(10)

Ejemplos de funciones lambda

```
lambda_func = lambda x: x**2 # Funcion que recoge un número entero y devuelve su cuadrado
print(lambda_func(3)) # Retorna 9

lambda_func = lambda x: True if x**2 >= 10 else False
print(lambda_func(3)) # Retorna False
print(lambda_func(4)) # Retorna True
```