Listas (continuación).

Al igual que las cadenas de caracteres, las listas pueden ser rebanadas (sub listas).
Por ejemplo, podríamos armar una lista a partir de otra lista más grande, utilizando solamente los primeros tres elementos.

```
lista = [1,2,3,4,5,6]
sub_lista = lista[:3]
print(lista)
print(sub_lista)
```

Definimos una nueva lista a la que llamamos sub_lista y le cargamos los primeros tres elementos de la lista original.
Así se ve el resultado por pantalla:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3]
```

Las listas también soportan operaciones de concatenación:

```
lista_1 = [1,2,3]
lista_2 = [4,5,6]
print(lista_1 + lista_2)
```

```
[1, 2, 3, 4, 5, 6]
```

Como ya habíamos visto, se puede agregar o quitar elementos de una lista, por lo que son del tipo mutable (es posible cambiar su contenido)
Otra operación interesante que podemos hacer con las sub listas, es reemplazar de una sola vez, el contenido de cierta parte de la lista original, sin necesidad de hacerlo de a un elemento.
Veamos un ejemplo, tengo una lista con cuatro letras "a" y quiero reemplazar la segunda y la tercera, por una letra "b".

Hacemos lo siguiente:

```
palabras = ['a','a','a','a']
print(palabras)

palabras[1:3]= ['b','b']
print(palabras)
```

Recordar siempre que en este tipo de pase de argumentos donde pasamos el inicio y el fin, el inicio siempre está incluido, pero el final no. Es por eso que como solo quiero cambiar dos elementos, pongo el número 3.

```
['a', 'a', 'a', 'a']
['a', 'b', 'b', 'a']
```

También puedo eliminar una porción de la lista, por ejemplo, las recién ingresadas letras "b":

```
palabras = ['a','a','a','a']
print(palabras)

palabras[1:3]= ['b','b']
print(palabras)

palabras[1:3]=[]
print(palabras)
```

Al ejecutarlo, esto veremos por pantalla:

```
['a', 'a', 'a', 'a']
['a', 'b', 'b', 'a']
['a', 'a']
```

Y con el mismo método, podemos también vaciar una lista entera:

```
palabras = ['a','a','a','a']
print(palabras)

palabras[:]=[]
print(palabras)
```

Lo anterior es equivalente a hacer:

```
palabras = ['a','a','a','a']
print(palabras)

palabras = []
print(palabras)
```

El resultado será el mismo, va a quedar la lista original, vacía:

```
['a', 'a', 'a', 'a']
[]
```

Listas anidadas.

Podemos anidar las listas, por ejemplo, para armar una matriz de datos.

```
linea1 = [1,2,3]
linea2 = [4,5,6]
linea3 = [7,8,9]

matriz = [linea1,linea2,linea3]

print(matriz)
```

En este caso, cuando la veamos por pantalla, veremos los elementos de una lista y a continuación los elementos de las demás listas:

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

Si queremos ubicar un elemento determinado de nuestro arreglo, podemos indicarle dos índices a nuestra lista matriz.

El primer índice indica la fila y el segundo indica la posición dentro de dicha fila.
Por ejemplo, si queremos mostrar por pantalla el número 6 de nuestra matriz, deberemos utilizar sus "coordenadas".
El primer argumento es el número de línea en donde se encuentra (contando a partir de cero) en nuestro caso será la línea 1, y a continuación, la posición dentro de dicha línea (también empieza a contar desde cero) en nuestro caso, el número 6 se encuentra en la posición 2.

```
linea1 = [1,2,3]
linea2 = [4,5,6]
linea3 = [7,8,9]

matriz = [linea1,linea2,linea3]

print(matriz[1][2])
```

Lo que estamos haciendo con nuestro PRINT es mostrar por pantalla el elemento de la matriz que se encuentra en la fila 1, posición 2, es decir, el número 6.

Recorriendo un arreglo.

Si bien Python proporciona herramientas para el uso de arreglos, podemos recorrer nuestra matriz que acabamos de crear de forma muy sencilla con una estructura FOR anidada:
Veamos un ejemplo en el que creamos tres listas, de tres elementos cada una y las anidamos creando una matriz de 3x3.

```
linea1 = [1,2,3]
linea2 = [4,5,6]
linea3 = [7,8,9]
matriz = [linea1,linea2,linea3]
```

Para recorrerlo, lo primero que debeos averiguar, es cuantas filas tiene nuestra matriz.

También deberemos averiguar el número de columnas, lo que podemos obtener fácilmente contando los elementos de una de sus filas, por ejemplo, la primera:

```
filas = len(matriz)
columnas = len(matriz[0])
```

Por último, anidamos dos estructuras FOR que recorran fila por fila, del número cero hasta la última, y por cada fila, que recorra todos sus elementos (columnas) del primero al último.

```
for fila in range(0,filas):
    for columna in range (0,columnas):
        print(matriz[fila][columna],end=',')
```

La sentencia END dentro del print es para evitar el salto de línea.
Esto veremos por pantalla:

```
1,2,3,4,5,6,7,8,9,
```

Y este es el código completo del ejercicio:

```
linea1 = [1,2,3]
linea2 = [4,5,6]
linea3 = [7,8,9]
matriz = [linea1,linea2,linea3]

filas = len(matriz)
columnas = len(matriz[0])

for fila in range(0,filas):
    for columna in range (0,columnas):
        print(matriz[fila][columna],end=',')
```

Creación de listas con LIST.

Podemos combinar la función RANGE con LIST para crear listas, cuyos elementos estén acotados por esta última.
Por ejemplo, queremos crear dos listas, una con los números del 1 al 10 y otra con los impares entre 1 y 10.

```
numeros = list(range(11))
impares = list(range(1,11,2))
print('Números: ',numeros)
print('Impares: ',impares)
```

Esta es la salida por pantalla:

```
Números:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Impares:  [1, 3, 5, 7, 9]
```

La sentencia PASS.

La sentencia PASS no hace nada, se utiliza en los casos en que el intérprete Python requiere una instrucción, pero no una acción, por ejemplo:

```
while True:
    pass
```

Este código no hace nada, solo se queda esperando que se presione una tecla.
También se utiliza (lo vamos a ver más adelante) en programación orientada a objetos, cuando declaramos una clase en su mínima expresión para luego desarrollar su contenido.

Unidad 3. Funciones.

Introducción.

Para definir funciones, utilizamos la palabra reservada DEF y a continuación, el nombre que le damos a la función y a lista de parámetros.
Las sentencias que forman parte del cuerpo de la función, comienzan a partir de la siguiente línea, y deben estar identadas (sangría).
Escribamos una función básica a la que le pasamos un entero por argumento, y nos devuelve dicho número, elevado al cuadrado.

```
def potencia(valor):
    resultado = valor**2
    print('El número ',valor,' elevado al cuadrado es = ',resultado)

potencia(3)
```

Al ejecutarlo, lo que hacemos es definir una función llamada potencia que recibe un valor por parámetro, lo guarda en la variable valor, lo eleva al cuadrado y muestra el resultado por pantalla.

```
El número  3  elevado al cuadrado es =  9
```

Una función definida puede ser asignada a otra (sin definir) para luego utilizarla como si fuese la función original.
Veamos un ejemplo simple con nuestra función potencia definida anteriormente.

```
def potencia(valor):
    resultado = valor**2
    print('El número ',valor,' elevado al cuadrado es = ',resultado)

otra_funcion = 0

print(type(potencia))
print (type(otra_funcion))
```

En nuestro código tenemos una función llamada potencia y una variable común llamada otra función.
Si ejecutamos el código anterior, veremos que Python nos informa el tipo de cada uno:

```
<class 'function'>
<class 'int'>
```

Probemos ahora de ver qué sucede si asignamos a la variable, nuestra función potencia:

```
def potencia(valor):
    resultado = valor**2
    print('El número ',valor,' elevado al cuadrado es = ',resultado)

otra_funcion = potencia

print(type(potencia))
print (type(otra_funcion))
```

Si lo ejecutamos, veremos por pantalla que ahora, nuestra variable es una función:

```
<class 'function'>
<class 'function'>
```

Y no solamente cambia el tipo de dato, ahora también podemos usar la nueva función de la misma forma en que utilizábamos la original:

```
def potencia(valor):
    resultado = valor**2
    print('El número ',valor,' elevado al cuadrado es = ',resultado)

otra_funcion = potencia

otra_funcion(4)
```

Este es el resultado visto por pantalla:

```
El número  4  elevado al cuadrado es =  16
```

Retorno de valores.

En nuestro ejemplo anterior, la función era la encargada de mostrar por pantalla, el resultado de la operación matemática. Pero podemos declararla de tal forma que retorne el resultado para poder utilizarlo en otra parte de nuestro código.

```
def potencia(valor):
    resultado=valor**2
    return(resultado)

print('El resultado de elevar 14 al cuadrado es: ',potencia(14))
```

A diferencia del ejemplo anterior, en este caso la función calcula el cuadrado del número pasado por argumento y lo devuelve para ser utilizado en otra parte del programa.

Argumentos y parámetros.

Cuando definimos una función que debe recibir valores, dichos valores son los parámetros de la función.
Cuando invocamos a la función y le pasamos los valores, esos valores se denominan argumento.
Veamos un par de ejemplos de cómo podemos invocar una función a la que debemos pasarle uno o más argumentos.

Pasaje de argumento por posición.

Cuando le enviamos argumentos a una función, estos se reciben y se procesan en base al orden en que son recibidos.
Veamos un ejemplo:

```
def resta (a,b):
    resultado=a-b
    print('Resultado de la resta: ',resultado)

resta(12,4)
```

En nuestro ejemplo, la función espera los valores para a y b, y los asignará en el orden en que se los pasemos.
Por ejemplo, si le pasamos el par de valores (12,4), al número 12 le restará el número 4.

```
Resultado de la resta:  8
```

Probemos que sucede si los pasamos invirtiendo el orden:

Pasaje de argumento por nombre.

En Python es posible desentendernos del orden en que pasemos los argumentos, si al invocar la función, especificamos a que parámetro de la misma corresponde cada valor que pasemos.
Ejemplo:

```
def resta (a,b):
    resultado=a-b
    print('Resultado de la resta: ',resultado)

resta(b=4,a=12)
```

Si bien estamos pasando el par de valores en el mismo orden que nuestro ejemplo anterior (cuando la función hacía 4 – 12) en esta ocasión, le especificamos a que parámetro corresponde cada uno.
Este es el resultado ahora:

```
Resultado de la resta:  8
```

Función invocada sin argumentos.

¿Qué pasaría si a nuestra función dedicada a restar dos valores que le pasamos por argumento, no le enviáramos uno de los argumentos o directamente ninguno?

```
def resta (a,b):
    resultado=a-b
    print('Resultado de la resta: ',resultado)

resta()
```

Al ejecutar el código del ejemplo, veríamos por pantalla el siguiente mensaje de error:

```
Traceback (most recent call last):
  File "D:\0000_Clases\Python\practica.py", line 5, in <module>
    resta()
TypeError: resta() missing 2 required positional arguments: 'a' and 'b'
```

Podemos evitar este tipo de error, asignando a nuestros parámetros, valores por defecto.

```
def resta (a=None,b=None):
    if a==None or b==None:
        print('Error. Debe pasar dos valores.')
        return
    resultado=a-b
    print('Resultado de la resta: ',resultado)

resta()
```

Al ejecutar el código, estamos invocando una función sin pasarle ningún argumento, pero en la definición de dicha función, dejamos especificado un valor por defecto para cada parámetro (None).
Al no recibir los argumentos, la función devuelve un mensaje de error:

```
Error. Debe pasar dos valores.
```

Argumentos indeterminados.

Podemos definir una función que reciba un número indeterminado de argumentos de la siguiente manera:

```
def lista (argumentos):
    print(argumentos)
    print(len(argumentos))

lista([1,'Hola',['a','b','c']])
```

Definimos una función que espera un argumento, pero en realidad le pasamos una lista.
La función muestra por pantalla los argumentos recibidos y la longitud en caso de que sea una lista.

```
[1, 'Hola', ['a', 'b', 'c']]
3
```

Si quisiéramos utilizar esta misma función para pasarle un solo argumento, lo aceptaría sin problemas, pero nos daría un mensaje de error por aplicarle la función LEN a un valor entero.

Números aleatorios.

En nuestros primeros códigos de Python vimos cómo importar el módulo keyword.py que se encuentra en la instalación del intérprete en la ruta \ lib
En esa misma carpeta tenemos el módulo random.py que vamos a importar de la misma manera:

```
import random
```

Para utilizarlo, nos valemos del método randrange(x,y,z) donde:

- x: inicio del rango de números.
- y: fin del rango de números (no está incluido)
- z: incremento

Solo es necesario el primer argumento, veamos algunos ejemplos de uso:

```
# Números aleatorios
import random

print('Número entre 1 y 10: ',random.randrange(11))

print()
print('Número entre 20 y 30: ',random.randrange(20,31))

print()
print('Número impar entre 1 y 20: ',random.randrange(1,21,2))
```

Este código nos dará la siguiente salida por pantalla:

```
Número entre 1 y 10:  2

Número entre 20 y 30:  28

Número impar entre 1 y 20:  9
```

Método RANDINT.

Este método a diferencia del anterior, si incluye en valor de fin de rango:

```
# Números aleatorios
import random

print()
print('Número entre 1 y 10: ',random.randint(1,10))
```

Método CHOICE (SECUENCIA).

Devuelve un elemento al azar de una lista de elementos predefinida.
Por ejemplo, tenemos una lista con marcas de autos y queremos seleccionar una al azar:

```
# Números aleatorios
import random
marcas=['Ford','Fiat','Chevrolet']
print('Marca: ',random.choice(marcas))
```

Método SHUFFLE ( ).

Mezcla al azar los elementos de una lista.

Supongamos que tenemos una lista con los números del 1 al 5 y reordenamos sus elementos antes de mostrarlos por pantalla:

```
# Números aleatorios
import random
numeros=[1,2,3,4,5]
print(numeros)

random.shuffle(numeros)
print(numeros)

random.shuffle(numeros)
print(numeros)
```

```
[1, 2, 3, 4, 5]
[4, 3, 5, 2, 1]
[1, 4, 2, 5, 3]
```