Colecciones en Python (Listas y Tuplas)

Las colecciones nos permiten agrupar datos, objetos, y otras estructuras bajo un mismo nombre.
En Python existen cuatro colecciones básicas: Listas, Tuplas, Conjuntos y Diccionarios, estas (colecciones) nos permiten agrupar datos, objetos y otras estructuras bajo un mismo nombre. Muy importantes en el manejo de bases de datos por que la información la debemos almacenar en alguna parte y la mejor parte para hacer el almacenamiento en memoria son las colecciones.

¿Ventajas?

Hasta el momento hemos trabajado solo con variables, si fuéramos a realizar un algoritmo que lea las 40 notas de 40 estudiantes y saque el promedio, la nota más alta, la más baja, etcétera, necesitaríamos 40 variables: n1, n2, n3, ……. n40. y utilizar el condicional if por ejemplo para ir comparando todas las notas y calcular la mas alta o la mas baja. Esto se debe a que estamos utilizando variables que son independientes, en mi mente como programador si les veo relación, pero para Python no existe ninguna relación entre ellas.
Pero cuando ya hacemos utilización de colecciones podemos decir algo como esto:

notas=[5,3,4.2, 2.9] aquí estoy creando una colección de datos, aquí estoy ganando varias cosas:

1. Relación entre los datos, las calificaciones en este caso se encuentran relacionadas y asociadas a un mismo nombre (notas), y voy a poder acceder a cualquiera de ellos a través de subíndices, podría por ejemplo hacer esto:
   print (notas[1]) # estoy imprimiendo la nota del segundo estudiante porque el primero esta en la posición cero.
2. La colección es un objeto y los objetos tienen métodos, o les puedo aplicar ciertas funciones, por ejemplo:
   print (max(notas)) # estoy obteniendo la nota mas alta de los estudiantes.
   print (min(notas)) # estoy obteniendo la nota más baja de los estudiantes.
   print (sum(notas)/len(notas)) # estoy obteniendo el promedio de notas de los estudiantes.

   En Python las listas son dinámicas, eso quiere decir que le puedo agregar, en otros lenguajes se llaman arreglos y debo definir cuantos elementos va a tener el arreglo, es decir son estáticos, en Python lo mas cercano a un arreglo son las listas como las del ejemplo de notas.
   Las listas y todas las demás colecciones que vamos a ver son completamente dinámicas, es decir puedo agregarle nuevos elementos.

   print (notas) # puedo imprimir las notas una a una sin necesidad de utilizar el ciclo for.
   notas.append (1) # con el método append agrego una nota, en este caso la nota 1 para que sea la menor.
   notas.append (input("ingrese la nota")) # puedo decirle al usuario que me agregue los elementos de la lista pasándole input
   En este caso me toma las notas como string, si queremos que las tome como enteros, entonces debemos tener algo así:
   notas.append (int(input("ingrese la nota")))

Esto también nos muestra otra característica de las listas y es que no tiene que almacenar elementos del mismo tipo.

Listas

Las listas son el tipo de colección en Python equivalente a los arreglos. Colección ordenada de elementos accesibles a través de un índice.

Creación y Acceso

Las listas utilizan corchetes para la creación y acceso de sus elementos.

```
lenguajes = ["Python", "Java", "PHP"]
print (lenguajes)  #Imprime la lista completa
print (lenguajes[0]) #Imprime el valor de la posición 0
lenguajes [0] = "Python 3" #Da un nuevo valor al elemento de la posición 0
print (lenguajes)
```

¿Podrías indicar que imprime el siguiente código?

```
numeros = [3,4,6,2,6,8,4,9]
res = numeros [-3] + numeros [5]
print (res)
```

Otra forma alternativa de crear listas es mediante la función ***list*** que recibe como argumento un objeto iterable.

Forma a.

```
mi_lista = list("1234567") # aquí el iterable es un string
print (type(mi_lista)) # tipo list
print (mi_lista)
```

También puedo crear una lista con la función ***range***

Forma b.

```
mi_lista = list(range(1,11))
print (type(mi_lista))
print (mi_lista)
print (mi_lista[3]) # accedo  la posición 4
```

Una lista en Python puede almacenar elementos de cualquier tipo de dato, incluso otras listas

```
mi_lista = [2, 3.5, True, "amigo", [3, 8, "a"]]
print (mi_lista)
```

¿De qué manera accederías al elemento con valor 6 en la siguiente lista? Completa el código de la línea 2

```
mi_lista = [1, 2, 3, ['p', 'q', [5, 6, 7]]]
print(mi_lista[3][2][1])
```

¿Como acceder al mismo valor utilizando índices negativos?

¿Acertaste? De esa manera podemos acceder a cualquier elemento de una lista.

¿Recuerdas el concepto de Slicing que vimos con los datos tipo cadena? Pues también es aplicable a las colecciones: listas.

```
mi_lista = list (range (1,10)) #[1,2,3,4,5,6,7,8,9] generamos una lista con el iterador range,
print (mi_lista) #[1,2,3,4,5,6,7,8,9]
print (mi_lista [1:4])# [2,3,4], el stop no se tiene en cuenta, llegamos hasta un punto anterior.
print (mi_lista [:2])# [1,2] como omito valor a la izquierda, se toma por defecto el 0
print (mi_lista [7:])# [8,9] posición 7 que es el 8 hasta el final, el 9
print (mi_lista [-5:-1])# [5,6,7,8] no tocaría el menos 1 por eso llega hasta 8.
print (mi_lista [-3:]) #[7,8,9] va hasta el final.
```

Como vamos viendo, las listas en Python son muy flexibles. Aquí algunas otras operaciones nativas sobre las listas.

```
mi_lista = [1, 2, 3, 4, 5]
mi_lista [0:3] = ["a", "b", "c"] # puedo asignar múltiples valores a múltiples
                                   posiciones con este tipo de sintaxis.
print (mi_lista)
```

```
mi_lista = [1, 2, 3, 4, 5]
mi_lista = mi_lista + [6] #Agregar un valor con el operador +
print (mi_lista)
mi_lista = mi_lista + [7, 8, 9] #Agregar varios valores
print (mi_lista)
```

```
mi_lista = [1, 2, 3]
x, y, z = mi_lista #Asignar a varias variables el contenido de una lista por asignación múltiple.
print (f"valor de x {x}")
print (f"valor de y {y}")
print (f"valor de z {z}")
```

Eliminación e Inserción

Para eliminar un elemento de una lista utilizamos la palabra reservada ***del***
Nota: del no es una función, como tal no requiere paréntesis.

```
lenguajes = ["Python", "Java", "PHP"]
del lenguajes[1]
print (lenguajes)
```

Podemos insertar datos en una posición específica con el método ***insert*** o al final de la lista con ***append***.

```
lenguajes = ["Python", "Java", "PHP"]
lenguajes.append ("C++")
lenguajes.append ("Ruby")
lenguajes.insert(2, "Visual Basic") # me inserta un elemento en la posición 2 de la lista
print (lenguajes)

lenguajes.insert(round(len(lenguajes)/2),"Delphi") # dos funciones: round y len para
                                                   insertar un nuevo elemento en la
                                                   mitad de la lista.
print (lenguajes)
```

Para saber si un elemento se encuentra dentro de una lista usamos la palabra reservada ***in***.

```
lenguajes = ["Python", "Java", "PHP"]
print ("Python" in lenguajes)
print ("C#" in lenguajes) # en ambos me responde si es true o false.
```

Nota: En otros lenguajes se requiere utilizar ciclos y recorrer posición por posición.

Recorrido de listas

Dado que cada lista es un objeto iterable, las podemos recorrer con nuestro ciclo ***for***
for variable in lista:

```
numeros = [1, 2, "papa", 3, 4, 0]
print(numeros) # si solo queremos imprimir los elementos de la lista no usamos ciclo for

for x in numeros:
  print (x, end= ",") # para hacer cálculos u operaciones sobre los elementos usamos el ciclo for
                      #  cambiamos el end por coma para que me aparezca la lista en una sola línea.
                         Por defecto es un salto de línea.
numeros = [1, 2, 3, 4, 5, "papa"] # estamos accediendo a los valores de la lista.

for x in numeros:
```

```
    if type(x) is int:
        if x % 2 == 0:
            print (x)
```

Si queremos que el usuario ingrese los valores podemos utilizar la siguiente sintaxis:

```
lista =[ ]
x = int(input("Ingrese la cantidad de datos a almacenar en la lista "))
for i in range(x):
    lista.append(int(input("ingrese valor ")))
print ("La lista es ", lista)
```

También nos las podemos arreglar para recorrer una lista a través de sus índices como convencionalmente se hace en otros lenguajes. Para ello, necesitamos usar la función len que nos devuelve la longitud de la lista.

```
mi_lista = [4, 5, 6, 7]
for i in range(0, len(mi_lista)): # estoy accediendo al valor del índice, no es muy Python
    print(mi_lista[i], end=",")

print("\n")

for i in mi_lista:
    print(i, end=",")
```

Python nos ofrece una forma alternativa de acceder al índice de cada elemento de una lista a través de la función ***enumerate***.

```
numeros = [3, 29, 34, 42, 54]
for x in enumerate(numeros): #usamos una sola variable X y contiene el índice y valor.
    print (x)
```

Debido a que la función ***enumerate*** devuelve para cada elemento de la lista un par (índice - valor), podemos obtener ambos al tiempo en la misma variable como en el ejemplo anterior o acceder a cada uno a través de variables diferentes. Veamos:

```
numeros = ["A", "B", "C", "D", "E"]
for a, b in enumerate(numeros): # for permite usar varias variables controladoras como a - b
    print (f"En el indice {a} encontramos el valor {b}")
```

Con lo visto hasta ahora, muéstranos cómo recorrerías una lista de la cual no conoces su longitud usando el ciclo ***while***.

```
lista = [2,3,4,5,6,7,8]
for x in lista: # este código es mas sencillo y seguro que con while.
    print(x)
```

```
lista = [2,3,4,5,6,7,8] #manera alternativa con while
total_elementos = len(lista)
indice = 0
while indice < total_elementos:
  print(lista[indice])
  indice += 1
```

Se cuenta con dos listas y se desean recorrer de manera simultánea mostrando el valor de las dos en cada posición. ¿Ayudarías a terminar el código de tal manera que se obtenga el siguiente resultado?

```
ma - pa
me - pe
mi - pi
mo - po
mu – pu
```

```
lista_1 = ["ma", "me", "mi", "mo", "mu"]
lista_2 = ["pa", "pe", "pi", "po", "pu"]

#continua aquí
for i in range(len(lista_1)): # la variable i toma los valores: [0,1,2,3,4]
    print (lista_1[i], "-", lista_2[i])
for x in range(5): # me trae los elementos del cero al cuatro.
    print (x)
```

Después de haberlo logrado, miremos otra alternativa para el recorrido de dos listas simultáneamente a través de la función ***zip***.

```
lista_1 = ["ma", "me", "mi", "mo", "mu", "ja"]
lista_2 = ["pa", "pe", "pi", "po", "pu", "jo"]
for x in zip (lista_1, lista_2):
    print (x)
```

A la función zip le podemos pasar cualquier número de listas y tendrá la capacidad de generar un iterador que las recorra simultáneamente.
También funciona con listas de diferente cantidad de elementos, en ese caso concatena hasta la menor cantidad. El siguiente ejercicio lo podemos hacer mejor con diccionarios y programación orientada a objetos.

```
nombres = ["maria", "luis", "carlos", "Juan", "Diego"]
edades = [34, 20, 14, 18, 20]
notas = [4, 3.8, 2, 5, 3.8]
for x in zip (nombres, edades, notas):
    print (x) # queda como registros
```

Métodos aplicables a las listas

Existen diferentes métodos útiles a la hora de trabajar con listas. Aquí algunas de ellas:

El método ***sort()*** ordena los elementos de una lista

```
#mi_lista = [5, 6, 3, 8, 1]
mi_lista = ["s1", "a3", "p1", "b5", "x7","a1"]
mi_lista.sort()  #ordena la lista
print (mi_lista)
```

Podemos pasarle el argumento reverse=True al método ***sort()*** para que ordene inversamente.

```
mi_lista = [5, 6, 3, 8, 1]
mi_lista.sort(reverse=True)  #ordena la lista inversamente
print (mi_lista)
```

El método remove () recibe como argumento un objeto y lo borra de la lista, es muy potente porque no elimina el elemento por el índice como se hace con la palabra reservada ***del***.

```
lenguajes = ["Python", "Javax", 5, "Java"]
if "Javax" in lenguajes:
    lenguajes.remove("Javax")
print (lenguajes)
```

Nota que la diferencia entre ***remove ( )*** y del es que al primero le puedo enviar el valor que deseo remover mientras que el segundo del trabaja con los índices.

```
lenguajes = ["Python", "Java", "PHP"]
del lenguajes [1]
print (lenguajes)
```

El método ***reverse ( )*** invierte el contenido de la lista.

```
lenguajes = ["Python", "Java", "PHP"]
lenguajes.reverse()
print (lenguajes)
```

Nota: las cadenas no tienen el método reverse. Que puedo hacer:
Ejemplo:

```
cadena= “Hola Mundo”
lista=list(cadena) # convierto la cadena en una lista.
print(lista) # a la lista si aplica la función reverse.
lista.reverse()
print(lista)
```

#pasamos luego la lista de nuevo a cadena con el método ***join***

```
cadena= “ ”.join(lista) # cadena vacía y el método de cadena join paso una lista y sus
                          elementos se convierten en cadena.
print(cadena)
```

Otros métodos o funciones aplicables sobre listas

clear()
Vacía todos los ítems de una lista:

```
lista.clear()
[ ]
```

extend()
Une una lista a otra:

```
l1 = [1,2,3]
l2 = [4,5,6]
l1.extend(l2)
[1, 2, 3, 4, 5, 6]
```

count()
Cuenta el número de veces que aparece un ítem:

```
veces=["Hola", "mundo", "mundo","Hola"].count("Hola")
print(veces)
```

index()
Devuelve el índice en el que aparece un ítem (error si no aparece):

```
["Hola", "mundo", "mundo"].index("mundo")
1
```

pop()
Extrae un ítem de la lista y lo borra:

```
l = [10,20,30,40,50]
print(l.pop()) # me extrae el ultimo elemento y me lo muestra.
50
Print(l)
[10, 20, 30, 40]
l = [10,20,30,40,50]
print(l.pop(2)) # le paso 2 como parámetro y me elimina el 30 de la lista.
print(l)
```

Reto: Eliminar todas las ocurrencias de un número en una lista

```
l = [30,20,30,40,30,50,30,20,30,30]
print(l.count(30))
l.remove(30)
l.remove(30)
l.remove(30)
l.remove(30)

for x in range(l.count(30)):
    l.remove(30)
print(l)
```

otra forma:

```
l = [30,20,30,40,50,30,20,30]
while 30 in l:
    l.remove(30)

print(l)
```

Tuplas

Generalidades

Las Tuplas son similares a las listas, pero con la característica de ser inmutables, es decir, sus valores no pueden cambiar. Las Tuplas se crean con paréntesis () en lugar de corchetes [] pero al igual que las listas, se accede a sus posiciones con corchetes [] Veamos:

```
mi_tupla = (2,3,4,5,6,9)
print (type(mi_tupla))

mi_tupla = [2,3,4,5,6,9]
print (type(mi_tupla))
```

Debe quedar claro que la forma como yo cree la colección es lo que va a determinar si se trata de una lista o una tupla.

```
print (mi_tupla)
print (mi_tupla[1])
print (mi_tupla[1:3])
print (mi_tupla[:])
# Todos los resultados los muestra entre paréntesis con lo cual puedo identificar que se trata de una tupla, pero el acceso a las posiciones lo hago con corchetes.
```

Intenta cambiar el contenido de la posición 1 en la siguiente tupla por el valor de "f":

```
mi_tupla = ("a","b","c","d")
print (type(mi_tupla))
mi_tupla[1] = "f"
print (mi_tupla[1])
print(mi_tupla)
```

Como puedes ver, el contenido de una tupla es inmutable
¿Qué ventaja tiene entonces una Tupla sobre una Lista?

- Son útiles cuando necesito crear un arreglo con valores de referencia y como programador debo garantizar que no cambien, por ejemplo, una tupla con los días de la semana.
- La seguridad, necesito guardar datos de forma segura (una versión de solo lectura).
- La eficiencia, para Python y para cualquier lenguaje aquello que es constante es mucho más rápido que aquello que puede ser variable, en bases de datos por ejemplo cuando tenemos un campo de tipo char la consulta es más rápida que si consulto un campo de tipo varchar. Entonces si voy a tener datos que no van a cambiar, es mejor practica de programación meterlos en una tupla que en una lista.

Las Tuplas al igual que las Listas pueden almacenar diferentes tipos de elementos y pueden conformarse por otras listas o tuplas internas. El acceso a sus elementos funciona igual que con las listas.

```
mi_tupla = (100, 'Hola', [1, 2, 3], -50)
print (mi_tupla[2][-1]) #accediendo al número 3
mi_tupla = (100, 'Hola', (1, (200, 'D', ["x","y","z"])), (1, 2, 3), -50)
print (mi_tupla [2][1][2][1]) #accediendo a la letra "y"
```

Las Tuplas comparten varios métodos de las listas, sin embargo, por ser inmutables, las Tuplas no usan ningún método en cuyo resultado sea un cambio de contenido. Ejemplo append(), insert(), remove(), sort(), reverse(), etc.

```
mi_tupla = (1,2,5,2,6,9,3,2,78,45,3,2,0)
print (mi_tupla.count (2)) #funciona.  No cambia el contenido de la tupla
```

```
mi_tupla = (1,2,5,2,6,9,3,2,78,45,3,2,0)
mi_tupla.sort() #No funciona.  Cambia el contenido de la tupla
print(mi_tupla)
```

Las tuplas también las podemos crear a partir de la función tuple que recibe un objeto iterable. Así como creábamos listas con la función list.

```
mi_tupla = tuple (amoamimama)
print (mi_tupla)

cadena = [2,3,54,65,76,4]
t = tuple (input("valor1:")+""+input("valor2:")+""+input("valor3:"))
print (t)

x =(2,3,4,5,6,2,3,7,9)
print (x.count(2))
```

Por lo demás, podemos resumir diciendo que las listas y las tuplas son semejantes, con la diferencia que las segundas no pueden cambiar su contenido.

Paso de Tupla a Lista y Lista a Tupla

Las funciones list y tuple son las que nos permiten realizar este tipo de conversiones. Miremos

Paso de Lista a Tupla

```
mi_lista = [1,2,3,4,5]
print (f"{mi_lista} es de tipo {type(mi_lista)}")
mi_tupla = tuple (mi_lista)
print (f"{mi_tupla} es de tipo {type(mi_tupla)}")

# Escenario: tengo una lista y por alguna razón quiero conservar esos datos y evitar que se cambien.
```

Paso de Tupla a Lista

```
mi_tupla = (1,2,3,4,5)
print (f"{mi_tupla} es de tipo {type(mi_tupla)}")
mi_lista = list (mi_tupla)
print (f"{mi_lista} es de tipo {type(mi_lista)}")

mi_tupla = (3,8,5,1,9)
print (mi_tupla)
mi_lista = list (mi_tupla)
mi_lista.sort()
mi_tupla = tuple (mi_lista)
print (mi_tupla)

# escenario, tengo una tupla y requiero hacer un cambio, después si deseo llevo de nuevo la lista

modificada y la convierto en tupla para proteger la información.
t1 = (input("ingrese numero:"),input("ingrese numero:"),input("ingrese numero:"))
print (t1)
#estoy creando una tupla (uso de paréntesis) con los valores ingresados.
```

Apropiación

1. Escribir un programa que almacene las asignaturas de un curso (por ejemplo, Matemáticas, Física, Química, Historia y Lengua) en una lista y la muestre por pantalla.

2. Escribir un programa que almacene en una lista los números del 1 al 10 y los muestre por pantalla en orden inverso separados por comas.

3. Escribir un programa que pregunte por las asignaturas de un curso y las almacene en una lista (por ejemplo, Matemáticas, Física, Química, Historia y bilingüismo) y las muestre por pantalla con el mensaje Yo estudio <asignatura>, donde <asignatura> es cada una de las asignaturas de la lista.

4. Pregunte al usuario cuantos elementos desea ingresar en una lista, luego solicite cada uno de ellos y presente el contenido de la lista y su contenido invertido.

5. Solicite al usuario dos frases y devuelva una lista con todas las letras que se repiten en la misma posición de ambas frases.
   Ejemplo:
   Frase 1: h**o**la mundo
   Frase 2: c**o**mo estas
   Respuesta: Lrmp=[“o”]

6. Escribir un programa que pregunte al usuario los números ganadores de la lotería de Risaralda, los almacene en una lista y los muestre por pantalla ordenados de menor a mayor.

7. Leer una frase y almacenar en una tupla la frase leída, pero sin espacios. Mostrar el contenido de la tupla.
   Nota: las posiciones de la tupla son palabras.

8. Crear una tupla con números e indica el numero con mayor valor y el que menor tenga.

9. Escribir un programa que almacene en una lista los siguientes precios, 50, 75, 46, 22, 80, 65, 8, y muestre por pantalla el menor y el mayor de los precios.

10. Escriba un programa que permita crear una lista de palabras. Para ello, el programa tiene que pedir un número y luego solicitar ese número de palabras para crear la lista. Por último, el programa tiene que escribir la lista.