Unidad 4. Colecciones.

Colecciones.

Ya vimos cómo manejar en Python las listas de datos con el tipo de datos LIST.
Pero el intérprete cuenta además con otros tres tipos de datos que nos facilitan el trabajo con grupos o listas de datos, estos son:

- Tuplas.
- Conjuntos.
- Diccionarios.

Lo que no incluye Python es el manejo de colas y pilas, pero su manejo se puede implementar mediante el uso de listas.

Tuplas.

Las Tuplas son similares a las listas, con la diferencia de que son inmutables, es decir, su valor no puede variar durante la ejecución del programa.
Muchas funciones utilizan este tipo de dato para devolver resultados, para asegurarse que este no será modificado, por lo que es conveniente aprender a diferenciarlas.

Las Tuplas, se definen como las listas, con la diferencia que, en lugar de corchetes, usamos paréntesis.

```
tpl1 = (1,4,'Hola',['a','b','c'])
print(tpl1)
```

Podemos ver el contenido por pantalla:

```
(1, 4, 'Hola', ['a', 'b', 'c'])
```

Al igual que como lo hacíamos con las listas, en las Tuplas también podemos mezclar todo tipo de datos, incluidas otras listas.
También aceptan indexación, por ejemplo, si queremos consultar, por el segundo elemento de nuestra tupla, lo podemos hacer de la siguiente manera:

```
tpl1 = (1,4,'Hola',['a','b','c'])
print(tpl1[1])
```

Y por pantalla, veremos el elemento escogido, en nuestro caso será el número 4.
Al igual que como sucedía con las listas, si necesitamos ver el último elemento, utilizamos el -1 (menos uno).

```
tpl1 = (1,4,'Hola',['a','b','c'])
print(tpl1[-1])
```

Y podremos ver, en nuestro caso, por pantalla, el último elemento de nuestra tupla:

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

Otra funcionalidad de las listas que también tenemos en las Tuplas es el poder acceder a porciones de la misma, por ejemplo, si quisiéramos ver desde el segundo elemento hasta el último deberíamos hacer lo siguiente:

```
tpl1 = (1,4,'Hola',['a','b','c'])
print(tpl1[1:])
```

Y por pantalla, deberíamos ver lo siguiente:

```
(4, 'Hola', ['a', 'b', 'c'])
```

Se aplica para las Tuplas todo lo visto para la creación y manejo de sub listas.
También podemos ubicar un determinado elemento dentro de una lista que esté dentro de una tupla.
Por ejemplo:

```
tpl1 = ([1,2,3],[4,5,6])
print(tpl1[1][2])
```

Lo que estamos pidiéndole a Python, es que nos devuelva, del elemento 1 de nuestra tupla (la segunda lista) el elemento en la posición 2 (en este caso, será el número 6).
Habíamos definido a las Tuplas como un tipo de dato inmutable, probemos que sucede si queremos modificar uno de sus elementos:

```
tpl1 = (1,2,3,4)
tpl1[0]=37
```

```
Traceback (most recent call last):
  File "D:\0000_Clases\Python\practica.py", line 2, in <module>
    tpl1[0]=37
TypeError: 'tuple' object does not support item assignment
```

Al querer modificar uno de los elementos de la tupla, el intérprete nos da un mensaje de error.

LEN.

Disponemos también de la función LEN para conocer la longitud de la tupla o de alguna de las listas que la integran:

```
tpl1 = ([1,2,3,4],[5,6,7])
print(len(tpl1))
print(len(tpl1[0]))
```

En el ejemplo anterior, tenemos una tupla cuyos elementos son dos listas, la primera de 4 elementos y la segunda de 3 elementos.
La longitud de la tupla es de 2 elementos, mientras que la longitud del primer elemento de la tupla, es 4:

```
2
4
```

INDEX.

Este método nos permite saber la posición que ocupa un determinado elemento dentro de la tupla, y en caso de existir, nos devolverá un mensaje de error:

```
tpl1 = (4,5,6,7,8)
print(tpl1.index(6))
```

En nuestro ejemplo, veremos por pantalla el número 2, que es el índice del elemento 6 dentro de la tupla.

COUNT.

Nos devuelve la cantidad de veces que un determinado elemento se encuentra dentro de una tupla, en caso de no existir, nos devolverá el número cero.

```
tpl1 = (4,5,6,7,8,6)
print(tpl1.count(6))
```

Al ejecutarlo, veremos por pantalla el número 2, ya que el elemento 6, aparece dos veces dentro de nuestra tupla.
Si en cambio consultamos por un elemento inexistente, nos dirá que dicho elemento, se encuentra cero veces dentro de nuestra tupla.

```
tpl1 = (4,5,6,7,8,6)
print(tpl1.count('hola'))
```

****** ¿Cómo podemos agregar un elemento al final de nuestra tupla?***

Conjuntos.

Los conjuntos son listas desordenadas de elementos únicos (no repetidos). Se los suele utilizar para eliminar duplicados, verificar pertenencia y soportan operaciones matemáticas complejas.
Podemos definir un conjunto mediante la función SET:

```
conjunto = set()
print(conjunto)
```

O mediante el uso de llaves:

```
conjunto = {1,2,3}
print(conjunto)
```

El método más utilizado de los conjuntos, el ADD que me permite añadir un nuevo elemento al conjunto.
Veamos que sucede si a nuestro conjunto, le añadimos el número 4:

```
conjunto = {1,2,3}
conjunto.add(4)
print(conjunto)
```

Al mostrarlo por pantalla, vemos que se agregó el número 4 al final de nuestro conjunto:

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

¿Qué sucederá si añadimos el número cero?

```
conjunto = {1,2,3}
conjunto.add(4)
print(conjunto)
conjunto.add(0)
print(conjunto)
```

```
{1, 2, 3, 4}
{0, 1, 2, 3, 4}
```

Vemos que el número cero, se agregó al principio del conjunto.
Probemos de agregar la letra 'M':

```
conjunto = {1,2,3}
conjunto.add(4)
print(conjunto)
conjunto.add(0)
print(conjunto)
conjunto.add('M')
print(conjunto)
```

```
{1, 2, 3, 4}
{0, 1, 2, 3, 4}
{0, 1, 2, 3, 4, 'M'}
```

Vemos que la letra se agrega al final de la lista, a continuación de los números.
Probemos de agregar más letras:

```
conjunto = {1,2,3}
conjunto.add(4)
print(conjunto)
conjunto.add(0)
print(conjunto)
conjunto.add('M')
print(conjunto)
conjunto.add('Z')
conjunto.add('A')
print(conjunto)
```

```
{1, 2, 3, 4}
{0, 1, 2, 3, 4}
{0, 1, 2, 3, 4, 'M'}
{0, 1, 2, 3, 4, 'M', 'Z', 'A'}
```

Por lo que vemos, a pesar de haber colocado el cero al principio de la lista, este tipo de colección, es del tipo desordenada.

Pertenencia.

Podemos verificar si un determinado elemento pertenece a un grupo:

```
conjunto = {'hola','chau'}
print('buenas tardes' in conjunto)
```

Tenemos un conjunto con saludos y queremos verificar si el elemento 'buenas tardes' pertenece a nuestro conjunto. El resultado será FALSE.
También puedo verificar si un elemento no está dentro del grupo, de la siguiente manera:

```
conjunto = {'hola','chau'}
print('buenas tardes' not in conjunto)
```

En este caso, el resultado será TRUE ya que es cierto que 'buenas tardes' no está en nuestro conjunto.

Duplicados.

Una de las cualidades más interesantes de los conjuntos, es que sus elementos son únicos, es decir, no poseen duplicados.
Veamos qué pasa si tenemos un conjunto con elementos que se repiten más de una vez.

```
conjunto = {2,3,4,2,5,2,2}
print(conjunto)
```

Al mostrar por pantalla nuestro conjunto, nos encontramos con que no nos muestra los repetidos:

```
{2, 3, 4, 5}
```

****** ¿Qué longitud tiene nuestro conjunto?***

Una manera interesante de explotar esta característica de los conjuntos, es eliminando duplicados de nuestras listas.
Por ejemplo, tenemos una lista con números duplicados, y la insertamos dentro de un conjunto:

```
lista = [2,2,3,4,2,2,5]
print(lista)
conjunto = set(lista)
print(conjunto)
```

Veamos que hay en cada uno mostrándolos por pantalla:

```
[2, 2, 3, 4, 2, 2, 5]
{2, 3, 4, 5}
```

Vemos que la lista contiene varias veces al número 2, pero al introducir la lista dentro de un conjunto, se eliminaron todos los elementos repetidos.
De ser necesario, podemos volver a cargar los elementos nuevamente en nuestra lista original:

```
lista = [2,2,3,4,2,2,5]
print(lista)
conjunto = set(lista)
print(conjunto)
lista=list(conjunto)
print(lista)
```

Este es el resultado, ahora nuestra lista original, ya no contiene elementos repetidos:

```
[2, 2, 3, 4, 2, 2, 5]
{2, 3, 4, 5}
[2, 3, 4, 5]
```

Podemos hacer todo este procedimiento de manera más sencilla:

```
lista = [2,2,3,4,2,2,5]
lista = list(set(lista))
print(lista)
```

****** ¿Qué pasa si hacemos esto mismo con una cadena de caracteres o con una lista de letras?***

Diccionarios.

Este tipo de colección, a l igual que los conjuntos, es muy utilizadas en Python.
Son colecciones desordenadas.
Los diccionarios tienen elementos definidos por una clave, que debe ser única, y un valor.
Podemos crear un diccionario directamente, sin cargarle elementos utilizando llaves:

```
diccionario = {}
print(type(diccionario))
```

```
<class 'dict'>
```

La sintaxis del diccionario será nombre = {clave1:valor1, clave2:valor2}
Veamos un ejemplo donde creamos un diccionario que traduce números del castellano al inglés.

```
diccionario = {'uno':'one','dos':'two','tres':'three'}
print(diccionario['dos'])
```

En nuestro ejemplo, la clave será el número en castellano, y el valor, el número en inglés.
Si queremos saber por ejemplo como se dice dos en inglés, utilizamos el diccionario como se ve en ejemplo, pasándole la clave y este nos devolverá el valor:

```
two
```

Podemos utilizar números como claves o como valores:

```
diccionario = {1:'one',2:'two',3:'three'}
print(diccionario[3])
```

```
three
```

Podemos modificar los valores del diccionario referenciándolos por su clave, por ejemplo, del diccionario anterior, queremos cambiar el valor asociado a la clave 3 (THREE) y ponerle, por ejemplo, el valor FOUR.

```
diccionario = {1:'one',2:'two',3:'three'}
diccionario[3]='four'
print(diccionario[3])
```

Veremos ahora por pantalla el valor FOUR en lugar de THREE.

Podemos eliminar elementos del diccionario mediante DEL, función que recibe el nombre del diccionario y la clave que queremos eliminar:

```
diccionario = {1:'one',2:'two',3:'three'}
print(diccionario)
del(diccionario[3])
print(diccionario)
```

```
{1: 'one', 2: 'two', 3: 'three'}
{1: 'one', 2: 'two'}
```

Podemos operar con los valores a través de sus índices, podemos aplicarles, por ejemplo, un incremento o podemos operar entre ellos.

```
diccionario = {'a':1,'b':2,'c':3}
print(diccionario['a']+10)
print(diccionario['a']+diccionario['b'])
```

```
11
3
```

En el ejemplo anterior, definimos un diccionario con las letras a, b y c como claves y los números 1, 2 y 3 como valores.
Probamos de sumar 10 unidades al valor de la clave ‘a’ y también vimos cómo podemos operar los valores mediante sus claves.
Estas modificaciones podemos, no solo hacerlas para obtener un resultado, sino hacerlas definitivas en el propio diccionario.

```
diccionario = {'a':1,'b':2,'c':3}
diccionario['a']+=10
print(diccionario['a'])
```

Si ejecutamos el código anterior, veremos que el valor correspondiente a la clave 'a' aumentó en 10 unidades su valor dentro del diccionario.

Recorriendo un diccionario.

Podemos recorrer todo el diccionario para ver sus elementos con un simple FOR:

```
diccionario = {'a':1,'b':2,'c':3}
for letras in diccionario:
    print(letras)
```

La desventaja de hacerlo así es que solo obtenemos los índices:

```
a
b
c
```

Si lo que necesitamos acceder es el valor asociado a la clave, o ambos, hacemos lo siguiente:

```
diccionario = {'a':1,'b':2,'c':3}
for letras in diccionario:
    print(letras,diccionario[letras])
```

```
a 1
b 2
c 3
```

ITEMS.

Otra forma de recorrer el diccionario y obtener ambos valores de manera similar a la anterior, es utilizando un método propio de los diccionarios, el método ítems.

```
diccionario = {'a':1,'b':2,'c':3}
for clave,valor in diccionario.items():
    print(clave,valor)
```

****** ¿Qué sucede si colocamos dos pares de elementos con la misma clave?***

Si bien el mismo resultado obtendríamos utilizando listas o Tuplas, los diccionarios son más rápidos cuando el volumen de datos es grande porque implementan una técnica llamada Hashing que nos permite acceder a una determinada clave (y su valor) de manera mucho más rápida.

****** ¿Qué nos devuelve la función LEN aplicada a un diccionario?***

Métodos de los diccionarios.

METODO KEYS.
Nos devuelve una lista de todas las claves que posee un diccionario.

```
diccionario = {1:'uno',2:'dos',3:'tres'}
print(diccionario.keys())
```

```
dict_keys([1, 2, 3])
```

METODO VALUES.
Nos devuelve la lista de todos los valores del diccionario.

```
diccionario = {1:'uno',2:'dos',3:'tres'}
print(diccionario.values())
```

```
dict_values(['uno', 'dos', 'tres'])
```

METODO CLEAR.
Elimina todos los elementos de un diccionario.

```
diccionario = {1:'uno',2:'dos',3:'tres'}
print(diccionario)
diccionario.clear()
print(diccionario)
```

```
{1: 'uno', 2: 'dos', 3: 'tres'}
{}
```

METODOS IN y NOT IN.
Nos permite evaluar la pertenencia o no de una clave en un diccionario.

```
diccionario = {1:'uno',2:'dos',3:'tres'}
print('Hola' in diccionario)
print('Hola' not in diccionario)
print(2 in diccionario)
print(2 not in diccionario)
```

```
False
True
True
False
```

Este tipo de comprobaciones son muy útiles ya que las operaciones con claves que no existen dan error en tiempo de ejecución.

METODO UPDATE.
Nos permite agregar pares de elementos clave, valor contenido en otro diccionario.

```
diccionario = {1:'uno',2:'dos',3:'tres'}
print(diccionario)
diccionario2={4:'cuatro'}
diccionario.update(diccionario2)
print(diccionario)
```

```
{1: 'uno', 2: 'dos', 3: 'tres'}
{1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro'}
```

Pilas y Colas.

La mayoría de los lenguajes implementan el uso de pilas y colas, pero Python no, aunque podemos simular el uso de estas mediante las listas.

Pilas.

Una PILA es una lista de elementos ordenados que solo permite dos tipos de acciones, agregar un elemento o quitarlo, teniendo en cuenta siempre que el último elemento ingresado, es el primero en salir.
Para simular una pila en Python, comenzamos creando una lista y agregando un elemento haciendo uso de APPEND que lo agrega justo al final, que es lo que necesitamos para simular el ingreso de un elemento a la pila.

```
pila = [1,2,3,4,5,6]
pila.append(7)
```

Y para quitar un elemento, debemos hacer uno del método POP que vimos cuando tocamos el tema de las listas, este método elimina el último elemento de la lista.
Si lo que necesitamos es utilizarlo, deberemos guardarlo en alguna variable:

```
pila = [1,2,3,4,5,6]
print('Pila original: ',pila)
pila.append(7)
print('Pila con un nuevo elemento: ',pila)
variable = pila.pop()
print('Pila con un elemento menos: ',pila)
print('Elemento quitado de la pila: ',variable)
```

```
Pila original:  [1, 2, 3, 4, 5, 6]
Pila con un nuevo elemento:  [1, 2, 3, 4, 5, 6, 7]
Pila con un elemento menos:  [1, 2, 3, 4, 5, 6]
Elemento quitado de la pila:  7
```

Hay que tener en cuenta que si intentamos sacar el último elemento de una pila que está vacía, obtendremos un error, por lo que siempre que trabajemos con pilas, deberemos conocer si la misma tiene o no elementos.

Cola.

Una cola es una lista de elementos en la que el primero en entrar es el primero en salir.
Para trabajar con colas, deberemos importar el módulo DEQUE:

```
from collections import deque
```

Para crear una cola, lo hacemos de la siguiente manera:

```
from collections import deque
cola = deque()
```

Y ahora le agregamos una lista de elementos, por ejemplo, nombres de clientes de un banco:

```
from collections import deque
cola = deque()
cola = deque(['Pedro','Luis','Pepe'])
print(cola)
```

```
deque(['Pedro', 'Luis', 'Pepe'])
```

Si queremos agregar elemento a la cola lo hacemos por medio de APPEND:

```
from collections import deque
cola = deque()
cola = deque(['Pedro','Luis','Pepe'])
print(cola)
cola.append('Laura')
print(cola)
```

```
deque(['Pedro', 'Luis', 'Pepe'])
deque(['Pedro', 'Luis', 'Pepe', 'Laura'])
```

Resumiendo, tenemos una cola, en la que el elemento más a la izquierda es el primero en haber entrado y cada vez que agreguemos otro, se hará al final de la misma, será el elemento más a la derecha.
Con APPEND resolvimos lo de agregar elementos al final, y ahora necesitamos comenzar a quitar elementos, que, por definición, el primero en salir, será el primero en haber entrado, es decir, el elemento más a la izquierda.
Para ello tenemos un método de colas llamado POPLEFT que no sirve para listas, no lo tienen definido, solo se pueden usar en colas.
Este método elimina el primer elemento de nuestra lista, y tal como hicimos con el método POP, si necesitamos guardar el valor para utilizarlo, deberemos hacerlo mediante una variable.

```
from collections import deque
cola = deque()
cola = deque(['Pedro','Luis','Pepe'])
print('Cola original: ',cola)
cola.append('Laura')
print('Cola con un nuevo elemento: ',cola)
variable = cola.popleft()
print('Cola con un elemento menos: ',cola)
print('Elemento quitado de la cola: ',variable)
```

El resultado de ejecutar el ejemplo anterior es el siguiente:

```
Cola original:  deque(['Pedro', 'Luis', 'Pepe'])
Cola con un nuevo elemento:  deque(['Pedro', 'Luis', 'Pepe', 'Laura'])
Cola con un elemento menos:  deque(['Luis', 'Pepe', 'Laura'])
Elemento quitado de la cola:  Pedro
```

Un ejemplo del uso de colas es en los servidores para el manejo de peticiones, en lugar de hacerlo en tiempo real lo hacen mediante colas, respondiendo a las mismas en el orden en que le van llegando.
Los sistemas operativos también hacen uso de las colas para la gestión de instrucciones o conjuntos de datos relacionados con su funcionamiento interno.