Unidad 2.4

Tipos de colección

1. Introducción

Los tipos de colección son estructuras de datos fundamentales que permiten almacenar, organizar y manipular colecciones de elementos. Estas colecciones son esenciales para resolver problemas complejos de manera eficiente, ya que facilitan el almacenamiento de múltiples valores en una sola variable, lo que resulta más práctico que manejar cada valor de forma individual.

Python ofrece una variedad de tipos de colección diseñados para cubrir diferentes necesidades, desde mantener los elementos en un orden específico hasta garantizar la unicidad de los datos. Además, estos tipos de colección pueden admitir la manipulación dinámica, lo que los convierte en herramientas indispensables en la programación.

2. Listas

Las listas son **ordenadas** y **mutables** (que se pueden agregar/eliminar elementos). Al ser ordenada, podemos acceder a los elementos de determinadas posiciones, ordenarla, o incluir un elemento en una posición determinada. Además, las listas nos permiten guardar diferentes tipos de datos en una misma lista, incluso otras listas.

2.1 Creación

Hay varios detalles para tener en cuenta a la hora de crear listas. Las listas se pueden crear utilizando corchetes “[“ “]” vacía o separando los elementos con comas.

```
x = []
y = [1,2,3,4]
print(x,y)
```

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

También es posible utilizar el método list() de varias formas. Como resumen, se puede usar sobre iterables (datos que se pueden recorrer):
-list(cadena): Nos devuelve una lista donde cada caracter de la cadena es un elemento de la lista
-list(colección): Utilizando el método list sobre cualquier tipo de colección (que veremos más adelante) nos devolverá una lista. Si se hace sobre un diccionario (dict) solo guardará las claves en la lista.
-list(rango): podemos invocar la función list() sobre una función range(número) que nos devolverá un rango del 0 al (número indicado-1). Por ejemplo:

```
x = range(5)

list(x)
```

```
[0, 1, 2, 3, 4]
```

-Objetos iterables personalizados: Si un objeto personalizado implementa el protocolo iterable (es decir, tiene un método __iter__()), puede convertirse en una lista.

-Iteradores: Convierte cualquier objeto iterador (como generadores) en una lista, consumiéndolo completamente (esto quiere decir que una vez se use llega a su máximo, por lo que volverlo a usar no devolvería nada).

```
generador = (x**2 for x in range(3))
resultado = list(generador)
print(resultado)

print(list(generador))
```

```
[0, 1, 4]
[]
```

2.2 Operaciones

-Indexación:

Como hemos visto al principio, las listas son ordenadas. Esto quiere decir que podemos acceder a cualquier elemento por su índice (si no existe, arrojará error):

```
x = ["Python", 20, True]

print(x[0])
```

```
'Python'
```

-Modificación:

Las listas son mutables (no como las cadenas de caracteres) por lo que podemos modificar un elemento de la siguiente forma:

```
x = ["Python", 20, True]

x[1] = 21

x
```

```
['Python', 21, True]
```

-Adición:

Podemos añadir elementos a una lista de varias formas

1. .append(elemento): añade al final de la lista el elemento que le indiquemos entre paréntesis. Hay que entender que como la lista es mutable, el uso de esta función no devolverá una lista nueva, sino que modificará la original.
2. .insert(elemento,posicion): añade el nuevo elemento en la posición especificada (teniendo en cuenta que la 0 es la primera).

-Suma:

Podemos crear una lista nueva como la suma de dos listas:

```
x = ["Python", 20, True]
y = [1,2,3]
z = x+y

print(z)
```

```
['Python', 20, True, 1, 2, 3]
```

Esto es similar a utilizar la función .extend(lista) para concatenar dos listas.

-Repetición:
Podemos crear una lista nueva multiplicando otra:

```
x = [1,2,3,4]

y = x*2

print(y)
```

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

-Eliminar elementos:
De nuevo, podemos eliminar de varias formas:

1. remove(elemento): Elimina el elemento que le pasemos como parámetro. En las listas se pueden repetir elementos, pero esta función solo eliminará la primera aparición del mismo.
2. .pop([posición]: Elimina el elemento que ocupa la posición indicada. Si no le pasamos ninguna posición, se elimina el último elemento por defecto. Esta función devuelve el elemento eliminado.

-Desempaquetar:
Se pueden asignar varios valores de la lista a la vez a nuevas variables

```
lista = ["Python", 20, True]
a,b,c = lista

print(a,b,c)
print(type(a),type(b),type(c))
```

```
Python 20 True
<class 'str'> <class 'int'> <class 'bool'>
```

-Ordenar:
Podemos ordenar las listas utilizando la función sort(key, reverse). Ordena la lista (no devuelve otra nueva) utilizando <. Hay que tener cuidado ya que si los tipos de datos de la lista no son los mismos o compatibles, Python arrojará un error. Son compatibles los numéricos y los booleanos (ya que True y False cuentan como 1 y 0 respectivamente). El parámetro key se utiliza para pasar una función para obtener un valor concreto de cada elemento de la lista, por ejemplo, convertirlo a todo mayúscula antes de comparar. Por

otro lado, el valor reverse indica si la ordenación se hace de forma inversa (el valor por defecto es false).

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

y = ["b","d","a","e","c"]

x.sort()

y.sort(key=str.upper, reverse=True)

print(x)

print(y)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
['e', 'd', 'c', 'b', 'a']
```

La función sorted(lista, key, reverse) también se puede utilizar para ordenar listas, pero en este caso sí que devuelve una lista nueva, no cambia la original.

-Buscar:
Podemos buscar de varias formas en una lista:

1. Contando: Si usamos el método .count(elemento), que nos devolverá el número de veces que el elemento aparece en la lista.
2. Buscando el índice: con el método .index(elemento, [inicio], [final]) podemos buscar la primera aparición del elemento indicado. También podemos buscar en un espacio determinado de la lista indicando las posiciones de inicio y final de la búsqueda.
3. Pertenencia: utilizando la expresión de la forma *elemento in lista*, nos devolverá un booleano indicando si el elemento se encuentra o no en la lista.

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

print(1 in x)

print(20 in x)
```

```
True
False
```

-Slice:
Se pueden obtener sublistas de la misma forma que vimos en el tema de cadenas de caracteres:
-lista[inicio:fin] # Desde la posición *inicio* hasta la posición *fin*.
-lista[inicio:] # Toda la cadena empezando desde la posición *inicio*.
-lista[:fin] # Toda la cadena hasta la posición *fin*.
-lista[:] # Toda la cadena
-lista[inicio:fin:salto] # Desde la posición *inicio* hasta la *fin* pero haciendo saltos de *salto* elementos.

-Recorrer listas:
Al igual que en las cadenas, podemos recorrer las listas con *for elemento in lista*.

-Otras funciones:
Hay otras funciones que trabajan con listas como len(lista), que devuelve la longitud; max(lista), que devuelve el elemento máximo; min(lista), que devuelve el mínimo; sum(lista), que hace un sumatorio…

3. Tuplas

Las tuplas son otro de los tipos de colección en Python y comparten ciertas similitudes con las listas, pero con una característica principal que las distingue: son inmutables. Esto significa que, una vez creadas, no pueden ser modificadas, lo que las hace ideales para almacenar datos que no deben cambiar durante la ejecución del programa.

3.1 Creación

Se crean utilizando paréntesis ( ) o sin paréntesis en ciertas ocasiones, separando los elementos por comas.

```
mi_tupla = (1, 2, 3)
otra_tupla = "a", "b", "c"  # Paréntesis opcionales
tupla_vacia = ()  # Una tupla vacía
tupla_un_elemento = (5,)  # Importante incluir la coma para diferenciar de un número
un_elemento = (5)  # Importante incluir la coma para diferenciar de un número

print(type(mi_tupla),type(otra_tupla),type(tupla_vacia),type(tupla_un_elemento),type(un_elemento))
```

```
<class 'tuple'> <class 'tuple'> <class 'tuple'> <class 'tuple'> <class 'int'>
```

También se puede utilizar la función tuple() tal y como se usa la función list() (explicada en el apartado anterior)

3.2 Operaciones

A parte de aquellas funciones que afectan a la colección en sí (.append(), .sort()), se pueden utilizar las mismas funciones que para list, pero teniendo en cuenta que las tuplas son inmutables, por lo que repeticiones, ordenar *sorted()* o concatenar solo nos servirán porque crean una tupla nueva.

4. Sets

Un set es un tipo de colección en Python que se caracteriza por ser mutable, desordenado y sin elementos duplicados. Los sets son ideales para realizar operaciones de conjuntos como uniones, intersecciones o diferencias, y para manejar colecciones de datos en las que no necesitamos un orden específico y queremos evitar elementos repetidos. Los elementos de un set deben ser inmutables, no puede contener listas u otros sets.

Por otro lado, también existen los *frozenset* que a diferencia de los *set* son inmutables, es decir, una vez creados no se puede ni añadir ni eliminar elementos.

4.1 Creación

Para crear sets utilizaremos { }. Si queremos crear set o frozenset vacíos tendremos que llamar al constructor, ya que si no Python creará un diccionario vacío.

```
conjunto = set()
conjunto_congelado = frozenset()
conjunto2 = {1,2,3,"Cadena"}

print(type(conjunto),type(conjunto_congelado),type(conjunto2))
```

```
<class 'set'> <class 'frozenset'> <class 'set'>
```

Al igual que las listas y tuplas, podemos llamar a estos constructores pasándoles cualquier objeto iterable.

4.2 Operaciones

-Añadir:

Para añadir elementos a un set (a los frozenset no se puede) podemos hacerlo de varias formas:

1. .add(elemento): permite añadir un elemento al set. Si ya existe ese elemento no hace nada
2. .update(iterable): permite actualizar el conjunto con un iterable. Solo añade aquellos elementos que no se encuentren en el conjunto

-Operadores lógicos:

1. "|" Unión: Une los elementos de dos conjuntos (sin repetir)
2. "&" Intersección: Coge solo aquellos elementos comunes a ambos conjuntos
3. "-" Diferencia: Encuentra los elementos presentes en un conjunto pero no en otro.
4. "^" Diferencia simétrica: Elementos que están en un conjunto u otro, pero no en ambos.

Todas estas operaciones las podemos realizar para crear nuevos conjuntos a partir de otros dos. Si a cualquiera de estos operadores lo seguimos con un =, en lugar de crear un nuevo set, se actualizará el primero. Si se usa para "actualizar" un frozenset, realmente se está creando un frozenset nuevo con la unión de los elementos, los frozenset son inmutables. También se pueden concatenar varias de estas operaciones seguidas, haciéndose en orden.

```
x = {1,2,3}
y = {3,4,5}

union = x | y
interseccion = x & y
diferencia = x - y

print(union)
print(interseccion)
print(diferencia)
```

```
{1, 2, 3, 4, 5}
{3}
{1, 2}
```

```
x = {1,2,3}
y = {3,4,5}

x ^= y

print(x)
```

```
{1, 2, 4, 5}
```

-Pertenencia:

1. in: como en las colecciones anteriores, podemos comprobar si un elemento está en un set utilizando in de la siguiente forma *elemento* in *set*, que nos devolverá True o False
2. Operadores “<>=”: con estos operadores podremos construir sentencias de pertenenencia que devolverán True o False dependiendo de si se cumplen.
    1. Set1 < Set2: Comprueba que Set1 sea un subconjunto de Set2 (subconjunto, no igual).
    2. Set1 <= Set2: Comprueba que los todos elementos de Set1 estén en Set2 (es decir, que sea un subconjunto o igual).
    3. Si en lugar de < utilizamos >, se comprobaría que los elementos de Set2 estén en Set1
    4. Por otro lado, también se pueden utilizar == y != para comprobar igualdad.

```
x = {1,2,3}
y = {1,2,3,4,5}
z = {5,6,7}

print(1 in x)
print(x < y)
print(z < x)
```

```
True
True
False
```

-Eliminar:

1. .remove(elemento): podemos eliminar un elemento concreto del conjunto. Si el elemento no existe en el conjunto se genera una excepción (KeyError).
2. .discard(elemento): igual que el anterior pero no da error si no existe el elemento.
3. .pop(): elimina un elemento cualquiera del conjunto. Devuelve KeyError si el conjunto está vacío.
4. .clear(): elimina todos los elementos del conjunto.

5. Diccionarios

Los diccionarios son uno de los tipos de colección más potentes y versátiles en Python. A diferencia de las listas, tuplas y sets, los diccionarios almacenan elementos en forma de pares clave-valor, lo que permite una organización y acceso eficiente a los datos. Este tipo de estructura es ideal cuando necesitas asociar datos (clave) con su correspondiente valor.

Son ordenados (a partir de Python 3.7), mutables y mantiene la unicidad de las claves. Las claves solo pueden ser tipos de datos no mutables.

5.1 Creación

Hay diferentes formas de crear diccionarios:

-Usando llaves {}:

```
diccionario = {"Clave": "Valor"}
diccionario_vacio = {}

print(type(diccionario))
print(type(diccionario_vacio))
```

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

-Usando el constructor dict():

```
diccionario = dict(nombre="Juan", edad=25, ciudad="Madrid")

print(diccionario)
```

```
{'nombre': 'Juan', 'edad': 25, 'ciudad': 'Madrid'}
```

-Con pares clave-valor iterables:

```
lista_pares = [("nombre", "Juan"), ("edad", 25), ("ciudad", "Madrid")]
tupla_pares = (("Clave", "Valor"),("Clave2","Valor2"))

print(type(lista_pares))
print(type(conjunto_pares))

diccionario = dict(lista_pares)
diccionario2 = dict(tupla_pares)

print(diccionario)
print(diccionario2)
```

```
<class 'list'>
<class 'frozenset'>
{'nombre': 'Juan', 'edad': 25, 'ciudad': 'Madrid'}
{'Clave': 'Valor', 'Clave2': 'Valor2'}
```

5.2 Operaciones

-Acceso a valores:
Podemos acceder a valores mediante diccionario[“Clave”]

```
diccionario = {"Manzanas": 19}
numero_manzanas = diccionario["Manzanas"]

print(numero_manzanas)
```

```
19
```

También se puede usar el método .get(Clave, Valor por defecto), que devuelve el valor por defecto (en lugar de error) cuando no encuentra la clave.

-Eliminar elemento:

1. del(dic[clave]): Elimina el elemento *clave* del diccionario *dic*. Si no existe lanza un KeyError.
2. .pop(clave, valorpordefecto): Elimina el elemento *clave* y devuelve su valor asociado. Si no existe devuelve error, a no ser que también le pasemos un valor por defecto.
3. .popitem(): Elimina y devuelve un par clave-valor. Antes de la versión 3.7 es una pareja aleatoria, a partir de esa versión lo hace en orden LIFO (Last in First Out, la última en entrar es la primera en salir).
4. .clear(): Elimina todos los elementos del diccionario.

-Modificar diccionario:

1. .update(other): Podemos actualizar el diccionario usando el método update y pasando como parámetro otro diccionario u otro iterable que devuelva parejas clave-valor.
2. Modificando cada valor: usando diccionario[clave] podemos actualizar el valor de las claves del diccionario (como en las listas).

-.keys(), .values(), .items(): Retornan vistas de las claves, valores, y pares clave-valor respectivamente.

Al igual que en otras colecciones, se puede utilizar *in* como palabra clave para comprobar pertenencia, y los bucles *for clave in diccionario* o *for clave,valor in diccionario* para recorrerlos.