Unidad 4

Secuencias

Una **secuencia** es una serie de elementos que se suceden unos a otros, y que tienen relación entre sí. Por ejemplo, la secuencia de Fibonacci.

En Python, existen varios tipos de secuencias, entre ellos: **listas**, **tuplas**, **strings** y **rangos**

Rangos

Un rango representa una secuencia de números (enteros) inmutable. Para crear un rango, usamos la función `range()` de Python, que se puede usar de diferentes formas:

- `range(fin)` → Crea un rango desde 0 hasta el número que le indiquemos (sin incluirlo). Ejemplo: `range(5)` crea un rango con los números 0, 1, 2, 3, y 4.
- `range(inicio, fin)` → Crea un rango entre los números que indiquemos como incio y fin. Ejemplo: `range(1, 6)` crea un rango con 1, 2, 3, 4, y 5.
- `range(inicio, fin, paso)` → Crea un rango que va desde inicio hasta fin, saltando la cantidad de números que indiquemos. Ejemplo: `range(5, 26, 5)` → 5, 10, 15, 20, 25

> 💡 Los valores de inicio, fin, y paso, pueden ser negativos. Por defecto, el valor de paso es 1.

Strings

Un string o cadena de caracteres, es una secuencia que solo admite caracteres como elementos.

```
texto = 'Hola Mundo'
print(texto[0]) # Imprime 'H'
```

Los strings son inmutables, es decir que no podemos modificar directamente los caracteres del string.

Tuplas

Las tuplas son un tipo de secuencia que puede contener elementos de cualquier tipo, y en una misma tupla pueden coexistir elementos de diferente tipo.

```
tupla = (1, 2, 3, 'hola')
print(tupla[0]) # Imprime el número 1
print(tupla[3]) # Imprime 'hola'
tupla[3] = 'hola mundo' # Error, no se pueden editar los elem
```

Las tuplas son estructuras de datos **inmutables**, por lo que no podemos editar sus elementos o agregar nuevos elementos una vez creada. Para cualquier modificación, será necesario copiar la tupla en otra variable.

Listas

Las listas son un tipo de secuencia genérica, que puede contener elementos de cualquier tipo, pero a diferencia de las tuplas, las listas son **mutables**. Podemos cambiar el tamaño, agregar, quitar, o modificar elementos dentro de la lista.

```
lista = [1, 2, 3, 'hola']
print(lista[0]) # Imprime el número 1
print(lista[3]) # Imprime 'hola'
lista[3] = 'Hola mundo' # Modificando un elemento
print(lista[3]) # Imprime 'Hola mundo'
lista.append(10) # Agregando un nuevo elemento al final de la
print(lista[4]) # Imprime el número 10
```

Es importante notar que si queremos operar sobre una lista sin modificar la lista original, podemos copiarla. Para esto es necesario emplear el método `copy()`, o de otro modo estaremos haciendo referencia a la lista original.

```
a = [1, 2, 3]
b = a # Esto es una referencia, si modificamos b también se m

a = [1, 2, 3]
b = a.copy() # Esto es una copia, podemos operar sobre b sin
```

Python tiene varios métodos para agregar y eliminar elementos de las listas:

Método	Descripción
append(valor)	Agregar elemento al final de la lista
clear()	Borrar todos los elementos de la lista
copy()	Devuelve una copia de la lista
count(valor)	Devuelve la cantidad de elementos con el valor indicado
extend(lista)	Agrega los elementos de otra lista al final de la lista
insert(posición, valor)	Inserta el elemento en la posición indicada
pop(posición)	Elimina el elemento en la posición indicada, o si no se especifica posición, elimina el último elemento
remove(valor)	Elimina el primer elemento que encuentre con el valor indicado
reverse()	Invierte el orden de los elementos de la lista
sort()	Ordena la lista en orden ascendente

Operadores de secuencias

Para todos los tipos de secuencias, Python tiene diversos operadores y métodos que podemos aplicarles:

x **in** s	Devuelve **True** si x pertenece a s, **False**, en caso contrario
s+t	Concatena la secuencia s y la t en ese orden
s*n	Concatena n veces la secuencia s
s[i] s[-k]	Referencia el elemento de la posición i de la secuencia s Referencia el elemento que está k posiciones antes del final
s[i:j]	Referencia la porción de la secuencia s que va del elemento i al j-1
s[i:j:k]	Referencia la porción de la secuencia s que va del elemento i al j-1, con paso k
>,<,>=,<=,!=,==	Se pueden comparar dos secuencias comparables
len(s)	Devuelve la longitud de la secuencia s
min(s)	Devuelve el mínimo elemento de s
max(s)	Devuelve el máximo elemento de s
sorted(s)	Devuelve s ordenada en forma de lista
reversed(s)	Devuelve s invertida
s.count(x)	Devuelve el número total de ocurrencias de x en s
s.index(x[, i[, j]])	Devuelve el índice de la primera ocurrencia de x en s (en la posición i o superior, y antes de j)

Ordenamiento

Ordenar una estructura de datos, como una lista en Python, es una actividad esencial en la programación. Esta tarea generalmente requiere el uso de un

algoritmo para reorganizar los elementos. El proceso implica cambiar la ubicación de los elementos para que, al final, la estructura contenga los mismos elementos en diferentes posiciones, según un criterio de ordenación.

Es importante destacar que este proceso de ordenamiento se aplica independientemente del estado inicial de la estructura de datos. No importa si la estructura estaba parcial o totalmente ordenada o si no lo estaba en absoluto; el resultado final siempre será el mismo una vez aplicado el algoritmo de ordenamiento.

Python tiene funciones de ordenamiento incorporadas, lo suficientemente flexibles para usar directamente cuando necesitamos ordenar nuestros datos.

Es crucial tener en cuenta que los algoritmos de ordenamiento requieren un considerable esfuerzo computacional, que puede aumentar, a veces exponencialmente, con listas cada vez más grandes. Por lo tanto, debemos ser conscientes del esfuerzo computacional involucrado y estar preparados para manejarlo.

En Python podemos utilizar la función `sorted()`, o el método `sort()`, que es exclusivo para listas. Ambos tienen un uso muy similar, sin embargo, `sort()` modifica la lista en la que se aplica el ordenamiento y no devuelve ningún valor, mientras que `sorted()` no altera la estructura original y devuelve una versión ordenada. Por lo tanto, `sorted()` puede aplicarse a tuplas o cadenas de texto.

Para poder ordenar una secuencia, los tipos de dato de todos sus elementos deben ser comparables, por ejemplo, todos números, todos strings, etcétera. Si tenemos varios tipos de dato en la misma secuencia, no hay forma de decidir cuál es “mayor” o “menor”.

Funciones map y filter

`map()` aplica una función a cada elemento de una secuencia, y genera otra secuencia con los resultados de aplicar dicha función.

Por ejemplo, una función que transforma números elevándolos al cuadrado:

Ejemplo	Salida

`def cuadrado(numero):`
`  return numero * numero`
 	
`lista = [3, 6, 1, 2]`
`print("Lista: ", lista)`
 	
`nueva_lista = list(map(cuadrado, lista))`
`print("Nueva lista: ", nueva_lista)`
	`Lista:  [3, 6, 1, 2]`
	`Nueva lista:  [9, 36, 1, 4]`

💡 La función `map` devuelve como resultado un iterador, si queremos utilizar métodos propios de las listas necesitaremos convertirlo a lista usando `list()`.

`filter()` aplica una función a cada elemento de una secuencia, pero a diferencia de `map`, esta función debe devolver un valor booleano, y `filter` generará otra secuencia solamente con los elementos que hayan "pasado" el filtro.

Por ejemplo, para filtrar dentro de una lista de strings, y quedarnos con los que empiecen por la letra "m":

```
def empieza_con_m(palabra):
    return palabra[0] == 'm'

lista = ["manzana", "balde", "molde", "marrón", "carro",
"pera"]
print("Lista: ")
for i in lista:
    print(i)

print()

nueva_lista = list(filter(empieza_con_m, lista))
print("Nueva lista: ")
for n in nueva_lista:
    print(n)
```

```
Lista:
manzana
balde
molde
marrón
carro
pera

Nueva lista:
manzana
molde
marrón
```

💡 La función `filter` también devuelve un iterador, necesitaremos convertirlo a lista para operar con los métodos propios de las listas.