**# \*\*Listas\*\***

**Son arreglos con comportamiento dinámico y pueden contener elementos de varios tipos así como otras listas. Para generar una lista vacía, podemos usar las siguientes instrucciones:**

```

mi\_lista = []

mi\_lista = list()

```

**También podemos inicializarlas de manera literal:**

numeros = [1, 2, 3, 4, 5] # Todos enteros.

decimales = [1., 2., 3., 4., 5.] # Todos decimales (note la importancia de usar el punto).

stack = [10.3, 1, "agua", -3] # Varios tipos

**Obtener la longitud o tamaño:**

print(len(numeros))

print(len(decimales))

print(len(stack))

**Acceder a elementos individuales. Es importante siempre tener en mente que si una lista tiene N elementos, los índices van de 0 a N-1.**

print(numeros[0], numeros[3]) # Primera posicion (0) y cuarta posicion (3)

print(decimales[-1], decimales[-2]) # Vamos del final hacia atras: -1 es el ultimo, -2 el penultimo

**Slicing: acceder a una secuencia de elementos**

**```**

**arreglo[inicio:final]**

**```**

**Nos retorna una secuencia desde el índice inicio al \*índice final-1\*. Importante, no va retornar lo que esté en la posición final, se queda en una posición anterior.**

datos = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(datos[1:5])

print(datos[7:9])

print(datos[7:10])

**Más ejemplos de slicing**

datos = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(datos[:]) # Accede a toda la lista.

print(datos[5:]) # del indice 5 hasta el final

print(datos[:5]) # del inicio hasta el indice 4 !!!!

print(datos[-5:]) # desde el indice -5 hasta el final

print(datos[2:8:2]) # cada dos elementos desde el 2 hasta 8-2.

print(datos[::2]) # cada dos elementos desde el inicio

**Para verificar si un elemento está en una lista, podemos usar el operador de membresía \*\*in\*\*:**

datos = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(0 in datos) # True

print(1 in datos) # True

print(100 in datos) # False

**Las listas se concatenan con el operador +**

lst = [1, 2, 3] + [0,] + [1.1, 2.2, -1.]

print(lst)

**El operador \* genera repeticiones y NO multiplicación. Esto puede ser útil para inicializar estructuras repetitivas.**

print(10\*[0,] )

print( 2\*[1,2] )

**Agregamos elementos \*individuales\* con la función append() y agregamos una lista de elementos con la función extend()**

lst = [1, 2, 3] + [0,] + [1.1, 2.2, -1.]

**# Uso de la función append.**

print(lst)

lst.append(-5.1)

print(lst)

**# Uso de la función extend.**

lst.extend([10, 20, 30])

print(lst)

**NOTA: si usaramos**

**```**

**lst.append([10, 20, 30])**

**```**

**en lugar de extend, el resultado sería**

**```**

**[1, 2, 3, 0, 1.1, 2.2, -1.0, -5.1, [10, 20, 30]]**

**```**

**El último elemento almacena una lista como objeto.**

**El uso de cada función dependerá del diseño de su programa.**

**También, podemos vaciar una lista con la función:**

**```**

**lst.clear()**

**```**

**El efecto es que lst queda como una lista vacía.**

**¿Cómo recorrer una lista con un ciclo?**

nums = [1,2,3,4,5,6] # Lista de numeros

tot = 0 # Acumulador de la suma

for i in range(len(nums)): # range debe tomar el tamaño de la lista como argumento

tot = tot + nums[i] # sumamos cada elemento

print("tot = ", tot) # imprimimos el resultado

\*\*Tip de Pythonista\*\*

**El ciclo anterior también se puede escribir de la forma:**

nums = [1,2,3,4,5,6] # Lista de numeros

tot = 0 # Acumulador de la suma

for n in nums: # La lista es un iterador

tot = tot + n # sumamos cada elemento

print("tot = ", tot) # imprimimos el resultado

**## \*\*Copias de Estructuras de Datos\*\***

# Creamos una lista:

lst = [1,2,3,4]

print("lst =", lst)

# Hacemos una copia.

lst\_copy = lst.copy()

# Cambiamos datos en nuestra copia

lst\_copy[1] = 4

lst\_copy[3] = 8

lst\_copy.extend([5,6,7,8])

print("lst\_copy = ", lst\_copy)

# La original queda intacta.

print("lst = ", lst)

**## \*\*Copias de Estructuras de Datos\*\***

En muchas ocasiones es necesario crear copias de estructuras de datos. En Python, hay que usar ciertos métodos según la estructura para realizar copias. Veamos el siguiente caso para comprender como Python maneja las estructuras de datos:

# Creamos una lista:

lst = [1,2,3,4]

print("lst =", lst)

# Intentamos (ingenuamente) hacer una copia

lst\_copy = lst

# Cambiamos datos en nuestra copia

lst\_copy[1] = 4

lst\_copy[3] = 8

lst\_copy.extend([5,6,7,8])

print("lst\_copy = ", lst\_copy)

# Veamos que paso con la original.

print("lst = ", lst)

# Podemos verificar que es la misma estructura con el operador is:

print(lst\_copy is lst)

Vemos que al modificar la copia se modifica la original. Esto es porque al ejecutar:

```

lst\_copy = lst

```

lst\_copy guarda una referencia a lst de modo que trabajamos con la misma estructura. Esto hace muchas cosas eficientes en cuanto a uso de memoria.

Este comportamiento lo encontraremos en muchos contextos de Python, y es causa de muchos "bugs" cuando se comienza con este lenguaje. Por lo pronto, para estructuras de datos, lo resolvemos con el método copy. Las listas, tuplas, diccionarios, y sets tienen su método copy. Veamos lo aplicado al ejemplo de lista de arriba:

# Creamos una lista:

lst = [1,2,3,4]

print("lst =", lst)

# Hacemos una copia.

lst\_copy = lst.copy()

# Cambiamos datos en nuestra copia

lst\_copy[1] = 4

lst\_copy[3] = 8

lst\_copy.extend([5,6,7,8])

print("lst\_copy = ", lst\_copy)

# La original queda intacta.

print("lst = ", lst)

# Podemos verificar que no es la misma estructura con el operador is:

print(lst\_copy is lst)