Tema 2

Listas y Cadenas de caracteres

Listas

Una **lista** es una estructura de datos que permite almacenar múltiples elementos en una sola variable.

- Es una estructura de tipo secuencia, se puede recorrer en orden.
- Las listas son objetos **mutables**, lo que significa que sus elementos pueden ser modificados después de su creación.
- Las listas pueden contener cualquier tipo de dato (números, cadenas, otras listas, etc.) y también pueden contener una mezcla de tipos de datos.
- Incluso podemos tener una lista de listas (como una matriz)

Sintaxis: una lista se define colocando los elementos entre **[ ]**

```
lista_vacia = []
lista_enteros = [1, 2, 3]
lista_heterogenea = ['a', 1, "hola"]
lista_listas = [[1,2], [3, 4]]
```

Creando listas

Además de definir directamente una lista encerrando entre corchetes sus elementos, podemos crear una lista a partir de otra estructura iterable y el constructor de listas, **list()**.

Por ejemplo, una cadena es una estructura iterable (se puede recorrer secuencialmente a través de sus elementos). En la imagen se crea una lista a partir de una cadena.

```
cadena = "hola"
mi_lista = list(cadena)
print(mi_lista)
```

```
['h', 'o', 'l', 'a']
```

Indexación

En Python existen varias formas de indexar los elementos de una lista:

Índice positivo: El índice 0 se refiere al primer elemento, el índice 1 al segundo, y así sucesivamente.

```
mi_lista = [10, 20, 30, 40]
print("Elemento 0:", mi_lista[0])
print("Elemento 1:", mi_lista[2])
```

```
Elemento 0: 10
Elemento 1: 30
```

Índice negativo: Los índices negativos empiezan desde el final de la lista, siendo -1 el último elemento, -2 el penúltimo y así sucesivamente.

```
mi_lista = [10, 20, 30, 40]
print("Elemento -1:", mi_lista[-1])
print("Elemento -2:", mi_lista[-2])
```

```
Elemento -1: 40
Elemento -2: 30
```

Indexación / slicing

Slicing: Permite obtener sublistas (es un nuevo objeto lista) seleccionando un rango de índices. El formato es `[inicio:fin]`. El elemento `inicio` se incluye, el elemento `fin` NO es incluido.

```
mi_lista = [10, 20, 30, 40]
sublista = mi_lista[1:3]
print(sublista)
```

```
[20, 30]
```

Incremento: Por defecto el slicing va de 1 en 1, pero podemos especificar un **incremento** diferente. En este caso el formato es `[inicio:fin:incremento]`

```
mi_lista = [10, 20, 30, 40]
print(mi_lista[0:4:2])
```

```
[10, 30]
```

Inicio y fin se pueden obviar, siendo en este caso desde el principio hasta el final (incluido). El incremento puede ser negativo, haciendo el slicing en orden inverso.

```
mi_lista = [10, 20, 30, 40]
print(mi_lista[2::-1])
```

```
[30, 20, 10]
```

Modificación de un valor

- Podemos modificar un valor indexando la posición y con una asignación

```
lista = [1, 2, 3]
lista[0] = 99
print(lista)
```

```
[99, 2, 3]
```

- También podemos modificar varios valores indexando una slice

```
lista = [1, 2, 3, 4, 5]
lista[1:2] = [88, 99]
print(lista)
```

```
[1, 88, 99, 3, 4, 5]
```

Comprobando lista vacía

- En Python existe una función predefinida que devuelve cuántos elementos tiene una estructura de datos, **len()**.
- Para saber si la lista está vacía o no, es más recomendable comprobar su valor booleano directo:
  - Una lista vacía se evalúa a False
  - Una lista con elementos se evalúa a True.

```
if mi_lista: #Equivale a: if len(mi_lista) > 0

if not mi_lista: #Equivale a if len(mi_lista)==0
```

Recorriendo listas

- Una lista se puede recorrer con un **for** de manera eficiente.
- Recuerda que un **for** en python recorre una estructura **iterable**. El **for** acaba automáticamente cuando acaba de recorrer el iterable.

```
for elemento in lista:
    print(elemento) #elemento es una variable que toma cada valor de lista
```

Operadores

El uso de operadores aritméticos para hacer operaciones aritméticas es conocido por todos. Pero estos operadores también están definidos para otros tipos de datos además de los numéricos, como las listas.

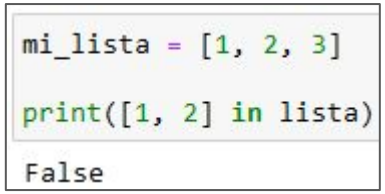
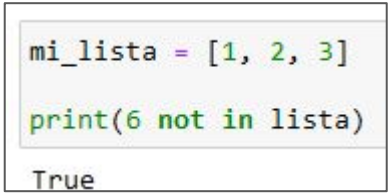
Operador	Acción	Ejemplo

+	“concatenación”: Une 2 listas en una sola.	`lista1 = [1, 2]`
		`lista2 = [3, 4]`
		`lista_resultado = lista1 + lista2`
		`print(lista_resultado)`
		`[1, 2, 3, 4]`

*	“repetición”: Repite una lista n veces	`lista = [1, 2]`
		`lista_resultado = lista * 3`
		`print(lista_resultado)`
		`[1, 2, 1, 2, 1, 2]`

+=	concatenación con asignación	
*=	repetición con asignación	

Operadores de pertenencia

Operador	Acción	Ejemplo
in	Comprueba si un **elemento** está dentro de una lista. La expresión se evalúa a **True** si el elemento **está** en la lista, False en caso contrario.	 En el ejemplo se comprueba si la lista [1, 2] está dentro “mi_lista”, lo que es falso, ya que “mi_lista” no tiene ningún elemento que sea una lista (es una lista de números).
not in	Comprueba si un **elemento** no está dentro de una lista. La expresión se evalúa a **True** si el elemento **NO** está en la lista, False en caso contrario.	 En este ejemplo se comprueba si el número 6 no está en la lista,

Listas: Métodos

Todos los métodos modifican el objeto y deben usarse mediante la sintaxis: ***objeto.metodo(parámetros)***

Inserción:

`append(x)`: Agrega el elemento x al final de la lista.

```
mi_lista = [1]
mi_lista.append(2)
print(mi_lista)
```

```
[1, 2]
```

`extend(iterable)`: Agrega todos los elementos de un iterable al final de la lista.

```
mi_lista.extend([3, 4])
print(mi_lista)
```

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

`insert(i, x)`: Agrega el elemento x en la posición i de la lista, desplazando al resto.

```
mi_lista.insert(0, 66)
print(mi_lista)
```

```
[66, 1, 2, 3, 4]
```

Listas: Métodos

Todos los métodos deben usarse a través del objeto mediante la sintaxis: ***objeto.metodo(parámetros)***

Borrado:

`remove(x)`: Elimina la primera aparición de x en la lista.

```
mi_lista.remove(3)
print(mi_lista)
```

```
[66, 1, 2, 4]
```

`pop([i])`: Elimina y retorna el elemento en la posición i. Si no se especifica índice, elimina y devuelve el último.

```
mi_lista = [66, 1, 2, 4]
print(mi_lista.pop(0))
print(mi_lista)
```

```
66
[1, 2, 4]
```

`clear()`: Elimina todos los elementos de la lista.

```
mi_lista.clear()
print(mi_lista)
```

```
[]
```

Listas: Métodos

- **`index(x)`**: Devuelve el índice de la primera aparición del valor especificado.
    - Se puede indicar el índice desde el que comenzar la búsqueda -> lista.index(x, i)
- **`count(x)`**: Devuelve la cantidad de veces que aparece el valor especificado en la lista.
- **`sort()`**: Ordena los elementos de la lista de menor a mayor.
    - Ordenar en orden inverso con el parámetro `reverse=True` -> lista.sort(reverse=True)
- **`reverse()`**: Invierte el orden de los elementos en la lista.
- **`copy()`**: Devuelve una copia de la lista. Esta copia es un nuevo objeto en memoria.

Funciones primitivas de Python aplicables a listas

Estas funciones no forman parte de ninguna librería, sino que son funciones del propio intérprete.

Algunas de estas que podemos aplicar a listas son:

- **`len(lista)`**: Devuelve el número de elementos en la lista.
- **`min(lista)`**: Devuelve el elemento con el valor mínimo de la lista (si todos los elementos son comparables).
- **`max(lista)`**: Devuelve el elemento con el valor máximo de la lista.
- **`sum(lista)`**: Devuelve la suma de los elementos de la lista (si todos son numéricos).
- **`sorted(lista)`**: Devuelve una nueva lista ordenada (sin modificar la lista original, a diferencia del método lista.sort(), que ordena el propio objeto lista).

2 funciones esenciales para trabajar con listas de valores Booleanos (aunque se pueden aplicar a listas de cualquier tipo, el valor Booleano equivalente a otros tipos lo hemos visto en el Tema 1)

- **`all(lista)`**: Devuelve `True` si todos los elementos de la lista son verdaderos (o si la lista está vacía).
- **`any(lista)`**: Devuelve `True` si al menos un elemento de la lista es verdadero.

Strings (str)

Un **string** es una **secuencia inmutable** de caracteres, no admite modificaciones.

```
cadena = "Python"
cadena[0] = 'L'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[55], line 2
      1 cadena = "Python"
----> 2 cadena[0] = 'L'

TypeError: 'str' object does not support item assignment
```

En general, se permiten casi todas las funciones primitivas y operadores definidas para listas, ya que un string es similar a una lista de caracteres.

String: Uso de comillas

Se permite tanto el uso de comillas simples como dobles.

```
cadena = "Python"

cadena = 'Python'
```

Si queremos incluir el símbolo de comillas dentro de la cadena tenemos 2 opciones:

- Encerrar el string entre un símbolo de comillas diferente

```
cadena = '"Sí", dijeron ellos'
print(cadena)
```

```
"Sí", dijeron ellos
```

- Escapar el carácter con \ -> \’

```
cadena = 'Para salir pulsa \'s\''
print(cadena)
```

```
Para salir pulsa 's'
```

String: operadores

En Python están definidos los operadores **+** y ***** para la clase str.

+ Concatena 2 cadenas

```
cadena = "Hola"
print(cadena + " mundo")
```

```
Hola mundo
```

***** Repite n veces una cadena

```
cadena = "255."
print(cadena * 3 + "255")
```

```
255.255.255.255
```

String: Indexación

La indexación funciona igual que en una lista.

Podemos indexar 1 elemento o hacer una slice.

Sintaxis `[inicio:fin:incremento]`

- Incremento es opcional
- Incremento puede ser negativo
- El fin no es incluido
- Si no ponemos fin, llega hasta el final (incluido)

Un “truco” para obtener el revés de un string o lista mediante slice [::-1]

```
cadena = "Python"
print(cadena[0:2])
```

```
Py
```

```
cadena = "Python"
print(cadena[2:])
```

```
thon
```

```
cadena = "Python"
print(cadena[::2])
```

```
Pto
```

```
cadena = "Python"
print(cadena[::-1])
```

```
nohtyP
```

String: Métodos

La **clase str** tiene multitud de métodos que pueden consultarse en la documentación oficial. Aquí veremos los más comunes.

A diferencia de las listas, los **str** son objetos **inmutables** (no se pueden modificar) por lo que estos **métodos** no modifican el objeto, sino que **RETORNAN** el resultado en un nuevo objeto.

- `upper()`: Retorna todos los caracteres a mayúsculas.
- `lower()`: Retorna todos los caracteres a minúsculas.
- `strip()`: Elimina los espacios en blanco al inicio y al final.
- `replace(old, new)`: Reemplaza una subcadena por otra.
- `split(separator)`: Retorna el string en una lista, utilizando un separador.

Para saber el tamaño de la cadena usamos la función nativa `len()`

f-Strings

Las **f-strings** son una forma eficiente y legible de formatear cadenas en Python.

Permiten **incrustar variables y expresiones** dentro de cadenas utilizando llaves `{}` y anteponiendo `f` a la cadena.

```
nombre = "Ana"
edad = 25
print(f"Mi nombre es {nombre} y tengo {edad} años.")
```

```
Mi nombre es Ana y tengo 25 años.
```