Arrays y listas

Un array es una estructura estática, de tamaño prefijado, que permite almacenar dentro diferentes datos, a los que podemos acceder por su posición numérica en la secuencia (empezando normalmente por la posición 0). En Python no existen arrays propiamente dichos; en su lugar, se utilizan **listas**, con un comportamiento similar pero flexible, donde podemos añadir y quitar elementos fácilmente en/de cualquier posición.

1. Definición y uso básico de listas

Una lista en Python será una secuencia de elementos que *pueden ser de distintos tipos*. Se representa con los elementos separados por comas, entre corchetes. Por ejemplo:

```
datos = ['Nacho', 'Pepe', 2015, 2013]
```

Además, podemos crear una lista sin valores iniciales, usando corchetes vacíos o la instrucción `list`:

```
datos = []
datos = list()
```

1.1. Acceder a los elementos de una lista

Como hemos dicho, los elementos de una lista se referencian por un índice o posición numérica, empezando en cero. Así, si tenemos una lista como esta:

```
datos = ['Nacho', 'Pepe', 2015, 2013]
```

y ejecutamos la instrucción `print(datos[1])`, se imprimirá por pantalla el segundo elemento de la lista, que es *Pepe*. Además, Python ofrece la posibilidad de imprimir una lista entera usando la misma instrucción `print`. En este caso, se mostrarían los datos tal cual están almacenados, separados por comas, y entre corchetes:

```
print(datos[1])     # Pepe
print(datos)        # ['Nacho', 'Pepe', 2015, 2013]
```

Finalmente, Python ofrece la posibilidad de acceder a los elementos desde el final de la lista, usando índices negativos. Así, `datos[-1]` en el ejemplo anterior obtendría el último elemento de la lista (2013 en este caso).

Para recorrer los elementos de una lista, podemos utilizar un `for` que recorra sus elementos...

```
for elemento in datos:
    print(elemento)
```

... o bien un `for` que recorra las posiciones:

```
for i in range(0, len(datos)):
    print(datos[i])
```

1.2. Listas multidimensionales

Una lista puede tener tipos simples (textos, números, etc) o tipos complejos (como por ejemplo, tuplas, u otras listas, u objetos de clases como veremos en otros documentos del curso). Aquí vemos una lista que internamente contiene otras listas:

```
datos = [
    ["Nacho", 40, 233],
    ["Pepe", 70, 231],
    ...
]
```

En este caso (cuando una lista contiene otras listas, o tuplas), la posición numérica dentro de la lista nos llevará a la lista o tupla que ocupa esa posición, con lo que necesitaremos otro índice numérico para indicar qué dato de esa lista o tupla nos interesa. Así, para el ejemplo anterior, `datos[0][0]` haría referencia al primer elemento de la primera lista (el nombre "Nacho"), y `datos[1][2]` al tercer dato de la segunda lista (el valor 231).

1.3. Añadir, modificar y borrar elementos

Si queremos añadir un elemento **al final** de la lista usamos la instrucción `append`. En el caso de querer insertar un elemento **en una posición determinada**, usamos la instrucción `insert`, indicando en qué posición queremos insertar, y el dato que queremos insertar. Automáticamente, todos los elementos a la derecha de esa posición se desplazarán para hacer hueco al nuevo elemento.

También podemos **actualizar el valor** de un dato de la lista, simplemente indicando su posición y el nuevo valor que queremos asignar:

Finalmente, si queremos **eliminar** un dato de la lista, usamos la instrucción `del` , seguida del elemento que queremos borrar. Alternativamente, podemos usar la instrucción `remove` de la propia lista, indicando el dato que queremos borrar. Se borrará la primera ocurrencia de ese dato

Aquí vemos un ejemplo completo de estas instrucciones

```
datos = [1, 2, 3]
datos.append(1000)  # [1, 2, 3, 1000]
datos.insert(1, 20) # [1, 20, 2, 3, 1000]
del datos[2]        # [1, 20, 3, 1000]
datos.remove(20)    # [1, 3, 1000]
```

2. Otras operaciones con listas

Hay otras operaciones que nos pueden resultar útiles sobre una lista. Por ejemplo, la instrucción `len(lista)` devuelve el número de elementos actualmente almacenado en la lista (tamaño de la lista). Esto nos puede servir para recorrer tanto listas simples como multidimensionales

```
datos = [1, 2, 3, 4]
print(len(datos))       # 4
datos2 = [
    [1, 2, 3],
    [4, 5, 6, 7]
]
print(len(datos2))      # 2 (filas)
print(len(datos2[0]))   # 3 (datos de la primera fila)
```

La instrucción `list(valor)` convierte un elemento en una lista de valores.

```
datos = list('123')
print(datos)            # ['1', '2', '3']
```

La operación `lista1 + lista2` concatena los datos de dos listas

```
datos1 = [1, 2, 3, 4]
datos2 = [4, 5, 6]
datosTotales = datos1 + datos2  # [1, 2, 3, 4, 4, 5, 6]
```

La operación `lista * N` genera una nueva lista donde los elementos de la lista original aparecen repetidos N veces

```
datos = [4, 5, 6]
datosRepetidos = datos * 3      # [4, 5, 6, 4, 5, 6, 4, 5, 6]
```

La expresión `n in lista` comprueba si el dato *n* está en la lista

```
datos = [4, 5, 6]
if 4 in datos:
    print("Existe el dato 4")
```

Las instrucciones `max(lista)` y `min(lista)` obtienen el mayor / menor valor de la lista, respectivamente. La función `sum` calcula la suma total de los elementos indicados.

```
datos = [4, 5, 6]
print(max(datos))       # 6
print(sum(datos))       # 15
```

La instrucción `lista.count(objeto)` obtiene el número de apariciones del objeto en la lista

```
datos = [4, 5, 6, 4]
print(datos.count(4))   # 2
```

La instrucción `lista.index(objeto)` obtiene la posición donde aparece por primera vez el objeto en la lista. Si el elemento no se encuentra, se produce una excepción en el programa. Podemos pasarle como segundo dato desde qué posición queremos seguir buscando.

```
datos = [4, 5, 6, 4]
print(datos.index(4))       # 0
print(datos.index(4, 1))    # 3
```

La instrucción `lista.sort(funcion)` ordena una lista según el criterio especificado en la función indicada. Para listas de datos simples (listas de enteros, de strings...) no hace falta indicar ninguna función: las listas se ordenan automáticamente de menor a mayor. Si queremos una ordenación inversa, usamos un parámetro adicional llamado `reverse`:

```
datos = [4, 2, 7, 5]
datos.sort()                # [2, 4, 5, 7]
datos.sort(reverse=True)    # [7, 5, 4, 2]
```

Finalmente, la instrucción `lista.reverse()` invierte el orden de la lista. Este método NO devuelve una nueva lista, sino que afecta a la original. Si queremos mantener el orden original y que la lista invertida sea otra nueva, podemos usar la instrucción `lista[::-1]` y asignar el valor a otra variable

```
datos = [1, 2, 3, 4]
datos.reverse()                # [4, 3, 2, 1]
datosInvertidos = datos[::-1]  # [1, 2, 3, 4]
```

Ejercicio 1:

Crea un programa llamado `ListaInvertida.py` que le pida al usuario que introduzca un conjunto de nombres separados por comas, y le muestre por pantalla la misma lista en orden inverso.

Ejercicio 2:

Crea un programa llamado `BuscaNumeros.py` que le pida al usuario que escriba números. El programa los irá añadiendo uno tras otro a una lista hasta que el usuario escriba 0. Entonces, le pedirá que diga un número y le indicará en qué posiciones de la lista aparece ese número.

AYUDA: vídeo con la solución del ejercicio

Ejercicio 3:

Crea un programa llamado `Identidad.py` que le pida al usuario un tamaño de tabla N y luego le deje rellenar los datos de N filas y N columnas de enteros. Al finalizar, le deberá decir si la tabla que ha rellenado se corresponde o no con una matriz identidad. Una matriz identidad es aquella que tiene unos en su diagonal principal y ceros en el resto. Por ejemplo (para un tamaño 3 x 3):

```
1 0 0
0 1 0
0 0 1
```