6.1 LISTAS

Las **listas** son secuencias de elementos de cualquier tipo y sin límite de longitud. El constructor y el tipo de dato es `list` en Python, aunque existe una representación más usual: una secuencia de elementos separados por comas y rodeados por corchetes.

```
>>> a = list([1,2,3])
>>> a
[1, 2, 3]
>>> type(a)
<class 'list'>
```

Para conocer todas las operaciones disponibles en las listas, se puede utilizar la función `dir` aplicada sobre una lista:

```
>>> dir(list())  # Igual que usar dir([1, 2, 3])
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__init_
subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_
ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

A continuación, se presentan las diferentes operaciones disponibles para la manipulación de listas en Python (`lst1` es un objeto de tipo lista; `elem`, un objeto arbitrario; `iter1`, un iterable diferente a `lst1`, y `pos`, un número entero que define la posición dentro de la lista):

- lst1.**append**(`elem`): añade el elemento elem al final de la lista `lst1`.
- lst1.**clear**(): elimina todos los elementos de `lst1`. Es equivalente a `del lst1[:]`.
- lst1.**copy**(): devuelve una copia superficial de `lst1`. Es equivalente a `lst1[:]`. Esta copia superficial solo copia las referencias de variables del primer nivel, por lo que, si los objetos o variables de niveles más profundos cambian, afectará a la copia y al original. Si se pretende hacer una copia real, es recomendable el uso de la librería **copy** usando las funciones `copy.copy` o `copy.deepcopy`.

- lst1.**count**(`elem`): cuenta el número de veces que el elemento `elem` aparece en la lista `lst1`.
- lst1.**extend**(`iter1`): extiende la lista `lst1` añadiendo todos los elementos del iterable `iter1`. Esta función es equivalente a `lst1[len(lst1):] = iter1`.
- lst1.**index**(`elem`[, `inicio`[, `final`]]): devuelve el índice de la posición que ocupa el elemento `elem` dentro de la lista `lst1`. El índice comienza por 0. Si se especifican los parámetros opcionales "inicio" o "final", estos determinarán la zona en la que se debe buscar el elemento `elem`, aunque el índice devuelto siempre es relativo al inicio de la lista. Si `elem` no se encuentra en la lista o en la sublista especificada, se elevará una excepción del tipo `ValueError`.
- lst1.**insert**(`pos`, `elem`): inserta `elem` en la posición anterior definida por el índice `pos`. Por ejemplo, `lst1.insert(0, elem)` insertaría `elem` al inicio de la lista, y `lst1.insert(len(lst1), elem)` lo insertaría al final de la misma. Este último sería equivalente a `lst1.append(elem)`.
- lst1.**pop**([`pos`]): elimina y devuelve el elemento de la posición definida por `pos`. El parámetro es opcional y, si se omite, se devuelve y elimina el último elemento de la lista.
- lst1.**remove**(`elem`): elimina la primera ocurrencia de `elem` en la lista. Si no existe ninguna ocurrencia de `elem` en la lista, se elevará una excepción del tipo `ValueError`.
- lst1.**reverse**(): invierte el orden de los elementos de la lista en el sitio (no genera una nueva lista, sino que lo hace en la misma; este método es conocido como *reverse in place*).
- lst1.**sort**(`key`=None, `reverse`=False): ordena la lista `lst1` en el sitio. En el parámetro key se puede añadir cualquier función que se utilice al ordenar, y el parámetro reverse se utiliza para que el orden sea inverso o no. Por lo general, en el parámetro key se utiliza un tipo de función anónima denominada lambda.

A continuación, se muestran ejemplos en los que se utilizan las operaciones disponibles en listas:

```
>>> a = [9, 9, 6]  # Similar a list([9, 9, 6])
>>> a.append(34.21)
>>> a
[9, 9, 6, 34.21]
>>> b = list(range(4))  # Creación de listas usando iterable
range
```

```
>>> b
[0, 1, 2, 3]
>>> a.extend(b)  # Extendiendo a con lista b
>>> a
[9, 9, 6, 34.21, 0, 1, 2, 3]
>>> a.extend(range(3))  # Extendiendo a con iterable range(3)
>>> a
[9, 9, 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.count(2)  # Número de ocurrencias del número 2
2
>>> a.index(2)  # Índice del primer elemento 2
6
>>> a.index(2, 7)  # Índice del elemento 2 contando desde la
pos 7
10
>>> a.index(2, 7, 9) # Índice 2 contando desde la pos 7 hasta
la 9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 2 is not in list
>>> b.clear()
>>> b
[]
>>> a
[9, 9, 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.insert(0, 'Pepe')
>>> a
['Pepe', 9, 9, 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.insert(3, 'Juan')
>>> a
['Pepe', 9, 9, 'Juan', 6, 34.21, 0, 1, 2, 3, 0, 1, 2]
>>> a.insert(34252, 'Ana')
>>> a
['Pepe', 9, 9, 'Juan', 6, 34.21, 0, 1, 2, 3, 0, 1, 2, 'Ana']
>>> a.pop(4)
```

```
6
>>> a.pop()
'Ana'
>>> a.remove(2)
>>> a
['Pepe', 9, 9, 'Juan', 34.21, 0, 1, 3, 0, 1, 2]
>>> a.reverse()
>>> a
[2, 1, 0, 3, 1, 0, 34.21, 'Juan', 9, 9, 'Pepe']
>>> a.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
>>> a.sort(key=lambda x: str(x))
>>> a
[0, 0, 1, 1, 2, 3, 34.21, 9, 9, 'Juan', 'Pepe']
>>> a.sort(key=lambda x: str(x), reverse=True)
>>> a
['Pepe', 'Juan', 9, 9, 34.21, 3, 2, 1, 1, 0, 0]
>>> a.sort(key=lambda x: hash(x) if isinstance(x, str) else
(x ** 2) / (x + 2))  # Las funciones de ordenación pueden ser
complejas
>>> a
['Pepe', 0, 0, 1, 1, 2, 3, 9, 9, 34.21, 'Juan']
```

Usando la función `copy` sobre listas solo se hace una copia superficial, por lo que los elementos más allá del primer nivel, como puede ser una lista dentro de otra lista, no se copian como copias, sino como referencias. Esto hace que realmente sigan estando enlazadas entre sí, aunque estén en dos variables distintas.

```
>>> xs = [[1, 2, 3], ['Juan', 'Ana'], True]
>>> ys = xs.copy()
>>> xs
[[1, 2, 3], ['Juan', 'Ana'], True]
>>> ys
[[1, 2, 3], ['Juan', 'Ana'], True]
```

```
>>> xs[0]
[1, 2, 3]
>>> xs[0][1] = 4536
>>> xs
[[1, 4536, 3], ['Juan', 'Ana'], True]
>>> ys
[[1, 4536, 3], ['Juan', 'Ana'], True]  # ¡ys también se ha
cambiado!
```

Sin embargo, si se utiliza una función que copie en profundidad, como `copy.deepcopy`, este problema desaparece:

```
>>> import copy
>>> yys = copy.deepcopy(xs)
>>> yys
[[1, 4536, 3], ['Juan', 'Ana'], True]
>>> xs
[[1, 4536, 3], ['Juan', 'Ana'], True]
>>> xs[0][2] = 'Coche'
>>> xs
[[1, 4536, 'Coche'], ['Juan', 'Ana'], True]
>>> ys
[[1, 4536, 'Coche'], ['Juan', 'Ana'], True]
>>> yys
[[1, 4536, 3], ['Juan', 'Ana'], True]
```

Gracias a su gran facilidad de uso, las listas pueden crearse fácilmente y crecer muchísimo, pero hay que tener en cuenta que son objetos que se guardan en la memoria del sistema. Si no se hace un buen uso de los recursos disponibles o se usan listas realmente grandes con miles o millones de elementos, se puede provocar un bloqueo por falta de memoria.

Un claro ejemplo es que las listas se pueden concatenar con otras listas y crear nuevos objetos de tipo lista, por lo que hay que tener cuidado de no generar demasiados objetos y utilizar demasiada memoria. Para conocer el espacio que ocupa un objeto en memoria, se puede hacer uso de la función `sys.getsizeof`.

```
>>> a = [1, 2, 3, 4, 5]
>>> sys.getsizeof(a)
96
```

```
>>> id(a)
4326533760
>>> a = [True, 2.5] + a + [2, 'texto']  # Se pueden
concatenar listas aunque el resultado genera una nueva lista
>>> a
[True, 2.5, 1, 2, 3, 4, 5, 2, 'texto']
>>> sys.getsizeof(a)
128
>>> id(a)
4326494016
```

Para eliminar cualquier objeto y, en particular, cualquier elemento de una lista, se puede usar el comando `del` en la posición (o posiciones) que el objeto ocupa en la lista.

```
>>> x = ['María', 'Pepe', 'Juan', 'Ana', 'Paco']
>>> del x[0]
>>> x
['Pepe', 'Juan', 'Ana', 'Paco']
>>> del x[1:3]
>>> x
['Pepe', 'Paco']
```

6.2 TUPLAS

Una tupla es un tipo de secuencia similar a una lista, pero es **inmutable,** por lo que, una vez inicializada, no se puede cambiar ninguno de sus elementos sin generar un nuevo objeto.

En Python, el tipo de las tuplas coindice con el constructor, en este caso, `tuple`. Asimismo, por defecto, cuando se añaden dos valores separados por una coma en cualquier parte de un código Python, estos son interpretados como una tupla de N elementos, y también ocurre si se deja una coma al final de la sentencia como se puede ver a continuación:

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> 3, 4, 5
(3, 4, 5)
>>> 45,
```