Listas en Python

Las listas en Python son un tipo de dato que permite almacenar datos de cualquier tipo. Son mutables y dinámicas, lo cual es la principal diferencia con los sets y las tuplas.

Crear listas Python

Las listas en Python son uno de los tipos o estructuras de datos más versátiles del lenguaje, ya que permiten almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas prácticamente lo que sea. Si vienes de otros lenguajes de programación, se podría decir que son similares a los arrays.

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

También se puede crear usando `list` y pasando un objeto iterable.

```
lista = list("1234")
```

Una lista sea crea con `[]` separando sus elementos con comas `,`. Una gran ventaja es que pueden almacenar tipos de datos distintos.

```
lista = [1, "Hola", 3.67, [1, 2, 3]]
```

Algunas propiedades de las listas:

- Son **ordenadas**, mantienen el orden en el que han sido definidas
- Pueden ser formadas por tipos **arbitrarios**
- Pueden ser **indexadas** con `[i]`.
- Se pueden **anidar**, es decir, meter una dentro de la otra.
- Son **mutables**, ya que sus elementos pueden ser modificados.
- Son **dinámicas**, ya que se pueden añadir o eliminar elementos.

Acceder y modificar listas

Si tenemos una lista `a` con 3 elementos almacenados en ella, podemos acceder a los mismos usando corchetes y un índice, que va desde `0` a `n-1` siendo `n` el tamaño de la lista.

```
a = [90, "Python", 3.87]
print(a[0]) #90
print(a[1]) #Python
print(a[2]) #3.87
```

Se puede también acceder al último elemento usando el índice `[-1]`.

```
a = [90, "Python", 3.87]
print(a[-1]) #3.87
```

De la misma manera, al igual que `[-1]` es el último elemento, podemos acceder a `[-2]` que será el penúltimo.

```
print(a[-2]) #Python
```

Y si queremos modificar un elemento de la lista, basta con asignar con el operador `=` el nuevo valor.

```
a[2] = 1
print(a) #[90, 'Python', 1]
```

Un elemento puede ser eliminado con diferentes métodos como veremos a continuación, o con `del` y la lista con el índice a eliminar.

```
l = [1, 2, 3, 4, 5]
del l[1]
print(l) #[1, 3, 4, 5]
```

También podemos tener **listas anidadas**, es decir, una lista dentro de otra. Incluso podemos tener una lista dentro de otra lista y a su vez dentro de otra lista. Para acceder a sus elementos sólo tenemos que usar `[]` tantas veces como niveles de anidado tengamos.

```
x = [1, 2, 3, ['p', 'q', [5, 6, 7]]]
print(x[3][0])    #p
print(x[3][2][0]) #5
print(x[3][2][2]) #7
```

También es posible crear sublistas más pequeñas de una más grande. Para ello debemos de usar `:` entre corchetes, indicando a la izquierda el valor de inicio, y a la izquierda el valor final que no está incluido. Por lo tanto `[0:2]` creará una lista con los elementos `[0]` y `[1]` de la original.

```
l = [1, 2, 3, 4, 5, 6]
print(l[0:2]) #[1, 2]
print(l[2:6]) #[3, 4, 5, 6]
```

Y de la misma manera podemos modificar múltiples valores de la lista a la vez usando `:`.

```
l = [1, 2, 3, 4, 5, 6]
l[0:3] = [0, 0, 0]
print(l) #[0, 0, 0, 4, 5, 6]
```

Hay ciertos operadores como el `+` que pueden ser usados sobre las listas.

```
l = [1, 2, 3]
l += [4, 5]
print(l) #[1, 2, 3, 4, 5]
```

Y una funcionalidad muy interesante es que se puede asignar una lista con `n` elementos a `n` variables.

```
l = [1, 2, 3]
x, y, z = l
print(x, y, z) #1 2 3
```

Iterar listas

En Python es muy fácil iterar una lista, mucho más que en otros lenguajes de programación.

```
lista = [5, 9, 10]
for l in lista:
    print(l)
#5
#9
```

```
#10
```

Si necesitamos un índice acompañado con la lista, que tome valores desde `0` hasta `n-1`, se puede hacer de la siguiente manera.

```
lista = [5, 9, 10]
for index, l in enumerate(lista):
    print(index, l)
#0 5
#1 9
#2 10
```

O si tenemos dos listas y las queremos iterar a la vez, también es posible hacerlo.

```
lista1 = [5, 9, 10]
lista2 = ["Jazz", "Rock", "Djent"]
for l1, l2 in zip(lista1, lista2):
    print(l1, l2)
#5 Jazz
#9 Rock
#10 Djent
```

Y por supuesto, también se pueden iterar las listas usando los índices como hemos visto al principio, y haciendo uso de `len()`, que nos devuelve la longitud de la lista.

```
lista1 = [5, 9, 10]
for i in range(0, len(lista)):
    print(lista1[i])
#5
#9
#10
```

Métodos listas

append(<obj>)

El método `append()` añade un elemento al final de la lista.

```
l = [1, 2]
l.append(3)
print(l) #[1, 2, 3]
```

extend(<iterable>)

El método `extend()` permite añadir una lista a la lista inicial.

```
l = [1, 2]
l.extend([3, 4])
print(l) #[1, 2, 3, 4]
```

insert(<index>, <obj>)

El método `insert()` añade un elemento en una posición o índice determinado.

```
l = [1, 3]
l.insert(1, 2)
print(l) #[1, 2, 3]
```

`remove(<obj>)`

El método `remove()` recibe como argumento un objeto y lo borra de la lista.

```
l = [1, 2, 3]
l.remove(3)
print(l) #[1, 2]
```

`pop(index=-1)`

El método `pop()` elimina por defecto el último elemento de la lista, pero si se pasa como parámetro un `índice` permite borrar elementos diferentes al último.

```
l = [1, 2, 3]
l.pop()
print(l) #[1, 2]
```

`reverse()`

El método `reverse()` inverte el órden de la lista.

```
l = [1, 2, 3]
l.reverse()
print(l) #[3, 2, 1]
```

`sort()`

El método `sort()` ordena los elementos de menos a mayor por defecto.

```
l = [3, 1, 2]
l.sort()
print(l) #[1, 2, 3]
```

Y también permite ordenar de mayor a menor si se pasa como parámetro `reverse=True`.

```
l = [3, 1, 2]
l.sort(reverse=True)
print(l) #[3, 2, 1]
```

`index(<obj>[,index])`

El método `index()` recibe como parámetro un objeto y devuelve el índice de su primera aparición. Como hemos visto en otras ocasiones, el índice del primer elemento es el `0`.

```
l = ["Periphery", "Intervals", "Monuments"]
print(l.index("Intervals"))
```

También permite introducir un parámetro opcional que representa el índice desde el que comenzar la búsqueda del objeto. Es como si ignorara todo lo que hay antes de ese índice para la búsqueda, en este caso el `4`.

```
l = [1, 1, 1, 1, 2, 1, 4, 5]
print(l.index(1, 4)) #5
```