

# Estructura de datos en Python – Tupla

## 1. Introducción

Una estructura de datos en Python es una forma particular de organizar y almacenar datos para que puedan ser utilizados de manera eficiente. Cada estructura de datos tiene propiedades específicas que determinan cómo se gestionan los datos, qué tipo de operaciones se pueden realizar en ellos y con qué eficiencia.

Algunas de las estructuras de datos más comunes en Python son:

**Listas (list):** Una secuencia ordenada de elementos que pueden ser de cualquier tipo. Las listas son mutables, lo que significa que se pueden modificar (agregar, eliminar o cambiar elementos).

**Tuplas (tuple):** Similar a las listas, pero son inmutables, lo que significa que no se pueden modificar una vez creadas.

**Conjuntos (set):** Una colección de elementos únicos, sin orden. No permite duplicados y es útil para realizar operaciones de conjuntos como intersección, unión, etc.

**Diccionarios (dict):** Almacenan pares de clave-valor. Las claves deben ser únicas y se utilizan para acceder a los valores correspondientes.

**Pilas y colas:** Se pueden implementar usando listas. Las pilas siguen el principio LIFO (Last In, First Out), mientras que las colas siguen el principio FIFO (First In, First Out).

Cada una de estas estructuras tiene diferentes ventajas dependiendo de cómo necesites almacenar y acceder a los datos.

Veamos a continuación la lista.

## 2. Tupla – tuple()

Las tuplas en Python son un tipo de dato que permite almacenar datos de cualquier tipo. Al contrario que las listas son inmutables, permiten almacenar un conjunto arbitrario de datos. Es decir, podemos guardar en ellas prácticamente lo que sea.

### Crear una tupla

En Python podemos crear una tupla vacía de dos formas distintas tal y como se muestra en la siguiente figura:

```

7  lista01 = list()
8  lista02 = []
9  print(lista01, lista02)

```

```

[] []

```

La función `list()` sin argumentos, permite crear una lista vacía. En la línea 8 se muestra otra forma de crear una lista vacía.

Para crear lista con elementos se puede hacer de varias maneras, veámoslas a continuación:

```

11 lista01 = ['a','b','c']
12 lista02 = [1,2,3]
13 lista03 = [1,'a', ['b', 2], (3,4), {1:'a'}, True]
14 lista04 = list('123') # crear una lista a partir de un iterable
15 lista05 = list([1,2,3])
16 print(lista01)
17 print(lista02)
18 print(lista03)
19 print(lista04)
20 print(lista05)

```

```

['a', 'b', 'c']
[1, 2, 3]
[1, 'a', ['b', 2], (3, 4), {1: 'a'}, True]
['1', '2', '3']
[1, 2, 3]

```

La línea 11 muestra la creación de una lista de tres caracteres, y la 12 de tres números enteros. Estas listas tienen todos los elementos del mismo tipo, en la línea 13 se muestra que los elementos no tienen que ser del mismo tipo. En la línea 14 se muestra que el comando `list()` se puede utilizar para crear una lista por medio de un iterador, en este caso se hace mediante un string y en la línea 15 se hace a través de una lista. La lista creada tiene como elementos cada uno de los elementos del iterador.

### Acceder a los elementos de una lista

Para acceder a cualquier elemento de una lista se debe poner el nombre de la lista seguida de la posición que se quiere acceder encerrados entre corchetes.

Las posiciones de las listas comienzan en la posición 0 y la última es la que corresponde con la longitud de la lista -1.

Al igual que con los string las listas pueden recorrerse de izquierda a derecha con valores desde 0 hasta `len(lista) - 1`. La función `len()` devuelve el número de elementos que tiene la lista.

Pero también pueden recorrerse de derecha a izquierda con valores que van desde -1 hasta `-len(lista)`.

Cuando se intenta acceder a una posición que no existe en la lista se produce un error.

Veamos algunos ejemplos:

```
22 # acceder a los elementos de una lista
23
24 # de izq a der (0..nro_eltos-1)
25 lista = [0,1,2,3,4,5,6,7,8,9]
26 print(lista[0], lista[5], lista[8]+lista[9])
27
28 # de der a izq (-nro_eltos..-1)
29 lista = [0,1,2,3,4,5,6,7,8,9]
30 print(lista[-1], lista[-4], lista[-9]+lista[-10])
```

```
0 5 17
9 6 1
```

```
33 a = [3]
34 print(a[2])
```

```
File "ejercicio.py", line 34, in module
    print(a[2])
    ~~~~
IndexError: list index out of range
```

## Modificación de los elementos de una lista

Para modificar un elemento de la lista, basta con acceder a su posición por el lado izquierdo de la instrucción de asignación. Veamos un ejemplo:

```
38 a = [1,2]
39 print(a)
40 a[1] = 7
41 print(a)
```

```
[1, 2]
[1, 7]
```

## Asignaciones con las listas

Hay que tener cuidado a la hora de asignar listas a una nueva variables, veamos el comportamiento que se produce al hacerlo.

```
38 a = [1,2]
39 b = a
40 print(a,b)
41 b[0] = 9
42 a[1] = 7
43 print(a,b)
```

```
[1, 2] [1, 2]
[9, 7] [9, 7]
```

Se crea la lista **a** con los elementos 1 y 2. Luego asignamos a una nueva variable **b** la lista **a**. Posteriormente se cambia en **b** el primer elemento y en **a** el segundo elemento. Como resultado se aprecia que los cambios hechos se reproducen en las dos listas que comparten la misma zona de memoria.

Esto ocurre porque en la línea 41 y 42 se realizan cambios en los elementos de la lista pero no se asigna una nueva lista. Por tanto las dos variables siguen apuntando a la misma zona de memoria en la que se encuentra la lista.

Veamos un nuevo ejemplo:

```
45 a = [1,2]
46 b = a
47 print(a,b)
48 a = ['a']
49 print(a,b)
```

```
[1, 2] [1, 2]
['a'] [1, 2]
```

Veamos ahora el siguiente ejemplo que se difiere del anterior, en la asignación de la lista **a**, que ahora se asigna una nueva lista y no se cambia uno de sus elementos.

```

52  a = [1,2]
53  b = a
54  print(a,b)
55  a = ['a']
56  print(a,b)

```

```

[1, 2] [1, 2]
['a'] [1, 2]

```

Como se aprecia en el código (línea 55) ahora a la variable a se le asigna una nueva lista. Eso hace que ahora a apunte a la zona de memoria de la nueva lista y b mantiene la referencia a la lista anterior, tal y como se muestra en la segunda imagen.

Hasta ahora hemos trabajado con listas cuyos elementos eran inmutable, ¿Qué pasaría si los elementos de la lista fueran mutables, como por ejemplo otra lista?

Estudiemos el siguiente caso:

```

58  a = [[1,2],['a','b'], ['c','d']]
59  b = a
60  print(f'a = {a}')
61  print(f'b = {b}')
62  a[2] = [3,4]
63  b[1][1] = 0
64  print(f'a = {a}')
65  print(f'b = {b}')

```

```

a = [[1, 2], ['a', 'b'], ['c', 'd']]
b = [[1, 2], ['a', 'b'], ['c', 'd']]
a = [[1, 2], ['a', 0], [3, 4]]
b = [[1, 2], ['a', 0], [3, 4]]

```

Al igual que con el ejemplo anterior, cuando la lista tiene cualquier tipo de elemento sea mutable o no, la asignación de una lista a otra permite que los cambios realizados en cualquiera de las listas se reflejen en las dos, tal y como se muestra en las imágenes anteriores.

## Listas de listas

Las listas en Python permiten almacenar cualquier tipo de elementos, en particular a otra lista, un ejemplo de ellos sería la siguiente lista

```

69 a = [1, ['a', 2], ['b', [3, 4], 'd']]
70 print(a[0]) # elemento de nivel 1
71 print(a[1])
72 print(a[1][1]) # elemento de nivel 2
73 print(a[2])
74 print(a[2][1])
75 print(a[2][1][0]) # elemento de nivel 3

```

```

1
['a', 2]
2
['b', [3, 4], 'd']
[3, 4]
3

```

La línea 69 define la lista `a` con 3 elementos. El segundo elemento de `a` es otra lista, los elementos de esa lista serán de nivel 2 con respecto a la lista `a`. El tercer elemento de `a` es otra lista que a su vez tiene como segundo elemento otra lista `[3,4]`. Estos elementos con respecto a la lista `a` son de nivel 3.

La línea 72 muestra la forma de acceso a los elementos de nivel 2 y la línea 75 lo hace con los de 3. Ver los resultados obtenidos en la segunda imagen mostrada.

## Recorrer una lista

Existen diversas formas para recorrer una lista

### Uso de `for..in`

Esta forma de recorrer la lista permite pasar por todos los elementos de la lista uno por cada iteración. Veamos un ejemplo:

```

80 lista = [0,'x',2,3,'a',5,6,7,8,'9']
81 for elto in lista:
82     print(elto, end = ' ')
83 print()

```

```

0 x 2 3 a 5 6 7 8 9

```

Como se aprecia en cada iteración, el valor de la variable `elto` toma el valor correspondiente al elemento de la lista.

### Uso de `for..range`

Esta forma de recorrer la lista permite pasar por todas posiciones de los elementos que hay en la lista una por cada iteración. Veamos un ejemplo:

```
86 lista = [0,'x',2,'a',4,5,6,7,8,'9']
87 for pos in range(len(lista)):
88     print(lista[pos], end = ' ')
89 print()
```

```
0 x 2 a 4 5 6 7 8 9
```

Recordamos que `range(n)` genera un iterador de números enteros entre 0 y `n-1`. Si `n` es la longitud de la lista, entonces el iterador genera los números correspondientes a las posiciones de los elementos en la lista.

Con esta forma de recorrerlo, se accede a los elementos de la lista mediante `lista[pos]`, al igual que con la forma anterior de recorrido se tiene acceso a los elementos uno en cada iteración.

### Uso de `while`

Con un bucle `while` se puede recorrer la lista a través de las posiciones que ocupan sus elementos, es muy similar al uso de `for` con `range()`. Veamos un ejemplo de ello:

```
91 # recorrido con while
92 lista = [0,'c',2,3,4,5,13,7,8,'x']
93 pos = 0
94 while pos < len(lista):
95     print(lista[pos], end = ' ')
96     pos += 1
97 print()
```

```
0 c 2 3 4 5 13 7 8 x
```

En este caso se debe utilizar una variable que tendrá los valores de las posiciones de los elementos en la lista, se debe recordar que la variable se debe incrementar en 1 antes de la siguiente iteración.

### Uso de `enumerate()`

`enumerate()` es una función en Python a la que se le pasa como argumento un iterador y este retorna un iterador de tuplas de dos elementos el primero de los cuales es la posición del elemento dentro del iterador y como segundo el propio elemento. Veamos un ejemplo de su uso:

```
100 lista = ['a', 2, 'c', 'd', 13, 'f', 0, 'h']
101 for pos, elto in enumerate(lista):
102     print(pos, elto)
```

```
0 a
1 2
2 c
3 d
4 13
5 f
6 0
7 h
```

```
104 lista = ['a', 2, 'c', 'd', 13, 'f', 0, 'h']
105 for tupla in enumerate(lista):
106     print(tupla)
```

```
(0, 'a')
(1, 2)
(2, 'c')
(3, 'd')
(4, 13)
(5, 'f')
(6, 0)
(7, 'h')
```

Las últimas dos imágenes demuestran que `enumerate()` genera tuplas de dos elementos.

## Recorrer una lista anidada

Para recorrer todos los elementos de una lista anidada es necesario usar bucles anidados. Recordamos que un bucle es anidado cuando en una iteración de un bucle se debe ejecutar otro bucle.

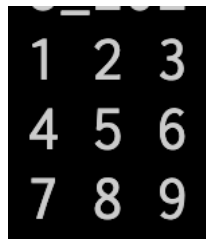
Veamos dos ejemplos de como recorrer un bucle anidado.

### Uso de `for..in`

En este caso el primer `for` se maneja entre las listas internas de la lista principal y el `for` interno se mueve entre los elementos de cada sublista. Veamos como hacerlo



```
108 lista = [[1,2,3],[4,5,6],[7,8,9]]
109 for fila in lista:
110     for elto in fila:
111         print(elto, end=' ')
112     print()
```

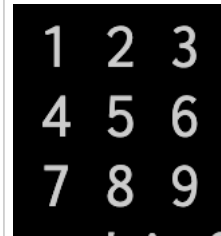


1	2	3
4	5	6
7	8	9

### Uso de for..range

Similar al ejemplo anterior pero ahora se trabaja con las posiciones de los elementos y no con el propio elemento. En este caso ha de hacerse uso de los [] y [][] para acceder a las filas y elementos de la lista anidada.

```
115 lista = [[1,2,3],[4,5,6],[7,8,9]]
116 for fila in range(len(lista)):
117     for columna in range(len(lista[fila])):
118         print(lista[fila][columna], end = ' ')
119     print()
```




1	2	3
4	5	6
7	8	9

En la línea 116 se obtiene la longitud de la lista principal y en la 117 se obtiene la longitud de cada sublista de la lista principal. En la línea 118 se muestra como acceder a los elementos de la lista anidada.

### Uso de for..while

Al igual que con la listas simples recorrer una lista anidada mediante el uso de dos bucles while es similar al uso del for..range(). La única diferencia es que hay que recordar que hay que incrementar en uno (o varias posiciones según el caso) las variables que se mueven a través de las subsistas y elementos de la lista anidada. Veamos como se hace.

```
122 lista = [[1,2,3],[4,5,6],[7,8,9]]
123 fila = 0
124 while fila < len(lista):
125     columna = 0
126     while columna < len(lista[fila]):
127         print(lista[fila][columna], end = ' ')
128         columna += 1
129     print()
130     fila += 1
```



1	2	3
4	5	6
7	8	9

## Recorrer una lista mediante comprensión

Una lista se puede recorrer de forma abreviada haciendo uso de la comprensión implementada en Python. Si queremos imprimir los elementos de una lista se haría de la siguiente forma:

```
133 lista = [0,1,2,'x',4,5,6,7,8,'*']
134 [print(elto, end = ' ') for elto in lista]
135 print()
```

```
0 1 2 x 4 5 6 7 8 *
```

## Slice sobre una lista [::]

Tal y como se vio con los string la operación slice permite obtener una parte de la lista a la que se le aplica.

Recordamos que la operación slice [valor\_inicial : valor\_final : salto] tiene tres parámetros que representan la posición inicial a usar, la posición final - 1 y cada cuantos elementos se realiza el salto. Tener en cuenta además que el salto podría tener valores positivo o negativo y que debe estar acorde al valor creciente o decreciente de los dos parámetros anteriores. Veamos algunos ejemplos de su uso.

Para la siguiente lista	144 lista = [0,1,2,3,4,5,6,7,8,9]
145 lista2 = lista[:5] 146 print(lista2)	[0, 1, 2, 3, 4]
147 lista2 = lista[5:] 148 print(lista2)	[5, 6, 7, 8, 9]
149 lista2 = lista[2:6] 150 print(lista2)	[2, 3, 4, 5]
151 lista2 = lista[1:7:2] 152 print(lista2)	[1, 3, 5]
153 lista2 = lista[6:2:-1] 154 print(lista2)	[6, 4]

```
157 lista2 = lista[-3:-7:-1]
158 print(lista2)
```

```
[7, 6, 5, 4]
```

```
159 lista2 = lista[-7:-3:1]
160 print(lista2)
```

```
[3, 4, 5, 6]
```

```
171 lista2 = lista[::-1]
172 print(lista2)
```

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

### Añadir elementos a una lista

Existen distintas formas de añadir elementos a una lista, veamos ejemplos de como hacerlo.

#### Uso de `append()`

`append(valor)` añade valor como último elemento de la lista, valor puede ser cualquier tipo de datos de los usados en Python. Veamos algunos ejemplos.

```
182 lista = list() # = []
183 lista.append(1)
184 lista.append(2)
185 print(lista)
```

```
[1, 2]
```

#### Añadir elementos mediante `[]`

Se pueden añadir elementos al final o al comienzo de una lista poniendo estos elementos en otra lista y concatenarlas. Veamos ejemplos

```
187 lista = [1,2]
188 lista += [3,4] # al final de la lista
189 lista = ['a','b'] + lista # al comienzo de la lista
```

```
['a', 'b', 1, 2, 3, 4]
```

En la línea 188 se añade a la lista los elementos 3 y 4 al final de la misma. Con la línea 189 se ejemplifica como añadirlos al comienzo de la lista.

### Uso de insert()

En las listas existe el método `insert()` que permite añadir un elemento en una posición determinada de la lista. Este método tiene dos parámetros `insert(posición, elemento)`.

Veamos unos cuantos ejemplos para explicar su funcionamiento.

```
192 lista = [0,2]
193 lista.insert(1,'a')
194 print(lista)
195 lista.insert(9,'z')
196 print(lista)
197 lista.insert(0,4)
198 print(lista)
199 lista.insert(-2,'*')
200 print(lista)
```

```
[0, 'a', 2]
[0, 'a', 2, 'z']
[4, 0, 'a', 2, 'z']
[4, 0, 'a', '*', 2, 'z']
```

En la línea 192 se establece la lista para el ejemplo con los valores 0 y 2.

En la línea 193 se agrega un elemento en la posición 1. Recordar que las posiciones de la lista comienzan en 0 y terminan en longitud de la lista menos 1.

Si al insertar un elemento mediante `insert()` ponemos una posición que sea mayor que la última posición de la lista, el elemento es añadido al final de la lista (línea 195)

Para insertar en la primera posición de la lista debe usarse como posición el valor 0 (línea 197).

Si al insertar un elemento mediante `insert()` ponemos una posición con valor negativo el elemento es añadido a la lista siguiendo el recorrido de la lista con posiciones negativas de derecha a izquierda.

### Uso de extend()

Otro método para añadir elementos al final de una lista es `extend()`. Este método debe recibir como parámetro un iterador y el resultado es que añade al final de la lista cada uno de los elementos del iterador suministrado. Veamos ejemplos de su uso.

```
337 lista = ['a','b','c','d','c']
338 lista.extend('123')
339 print(lista)
```

```
['a', 'b', 'c', 'd', 'c', '1', '2', '3']
```

```
341 lista = ['a','b','c','d','c']
342 lista.extend(range(2))
343 print(lista)
```

```
['a', 'b', 'c', 'd', 'c', 0, 1]
```

```
345 lista = ['a','b','c','d','c']
346 lista.extend([6,7,9])
347 print(lista)
```

```
['a', 'b', 'c', 'd', 'c', 6, 7, 9]
```

```
349 lista = ['a','b','c','d','c']
350 lista.extend({0:'a', 1:'b'})
351 print(lista)
```

```
['a', 'b', 'c', 'd', 'c', 0, 1]
```

## Borrar elementos de una lista

### Uso de remove()

El método remove() se utiliza para eliminar de la lista la primera ocurrencia de un elemento dado siempre que exista. Si el elemento no existe entonces se produce un error que se deberá controlar con try o preguntar antes de eliminar. Veamos algunos ejemplos para demostrar como usarlo.

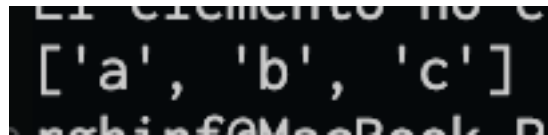
```
204 lista = ['a','b','c']
205 lista.remove('a')
206 print(lista)
```

```
['b', 'c']
```

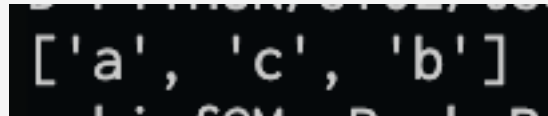
```
208 lista = ['a','b','c']
209 lista.remove('z')
210 print(lista)
```

```
Traceback (most recent call last):
  File "/Users/rghinf/ACADEMICO/CURSO_24_2
<module>
    lista.remove('z')
~~~~~^^^^^^
ValueError: list.remove(x): x not in list
```

```
214 try:
215     lista = ['a','b','c']
216     lista.remove('z')
217     print(lista)
218 except:
219     print('El elemento no está en la lista')
220 print(lista)
```



```
222 lista = ['a','b','c', 'b']
223 if lista.count('b') != 0:
224     lista.remove('b')
225 print(lista)
```



En la primera imagen se muestra como utiliza el método `remove()` para borrar un elemento de la lista (línea 205).

En la segunda imagen se muestra que si el elemento no está en la lista se interrumpe el programa.

En el tercero de los códigos se muestra el uso del `try` para evitar el error al intentar borrar un elemento que no está en la lista.

El cuarto de los códigos muestra como evitar el error usando el método `count()`.

#### *Borrar todas las ocurrencias de un caracter en una lista*

El siguiente ejemplo muestra un ejemplo de como borrar todas las ocurrencias de un caracter en una lista.

```
229 lista = ['a','b','c', 'b']
230 while lista.count('b') != 0:
231     lista.remove('b')
232 print(lista)
```



#### *Cuidado al borrar elementos de una lista dentro del bucle*

Muchas veces cuando estamos iterando con una lista tenemos la necesidad de borrar elementos de la lista que se itera. Veamos con varios ejemplos el comportamiento del siguiente programa en la que se itera sobre la lista y se borra elemento en el interior del bucle.

```

234 lista = ['b','a','b','c']
235 for car in lista:
236     print('+++',lista)
237     if car == 'b':
238         lista.remove(car)
239     print('-->', lista)
240 print('***',lista)

```

```

+++ ['b', 'a', 'b', 'c']
--> ['a', 'b', 'c']
+++ ['a', 'b', 'c']
--> ['a', 'c']
*** ['a', 'c']

```

```

234 lista = ['b','b','a','c']
235 for car in lista:
236     print('+++',lista)
237     if car == 'b':
238         lista.remove(car)
239     print('-->', lista)
240 print('***',lista)

```

```

+++ ['b', 'b', 'a', 'c']
--> ['b', 'a', 'c']
+++ ['b', 'a', 'c']
+++ ['b', 'a', 'c']
+++ ['b', 'a', 'c']
*** ['b', 'a', 'c']

```

```

234 lista = ['b','b']
235 for car in lista:
236     print('+++',lista)
237     if car == 'b':
238         lista.remove(car)
239     print('-->', lista)
240 print('***',lista)

```

```

+++ ['b', 'b']
--> ['b']
+++ ['b']

```

```

269 lista = ['b','b','a','c']
270 for pos in range(len(lista)):
271     print('+++',lista)
272     if lista[pos] == 'b':
273         lista.remove(lista[pos])
274     print('-->', lista)
275 print('***',lista)

```

```

+++ ['b', 'b', 'a', 'c']
--> ['b', 'a', 'c']
+++ ['b', 'a', 'c']
+++ ['b', 'a', 'c']
+++ ['b', 'a', 'c']
Traceback (most recent call last):
  File "/Users/rghinf/ACADEMICO/CURSO
    if lista[pos] == 'b':
        ~~~~~^~~~~~
IndexError: list index out of range

```

Como se aprecian las imágenes, si intentamos borrar todos los caracteres 'b' de la lista con el primer código propuesto la respuesta que se obtiene es correcta.

Pero si utilizamos el mismo código con otras lista de datos como en los ejemplos 2 y 3 el resultado obtenido no es correcto, ya que no se eliminan todos los caracteres solicitados.

En el último de los códigos se quiere utilizar un `for in .. range(len(lista))` para acceder a los elementos de la lista mediante sus posiciones, pero vemos que al eliminar elementos de la lista, se termina obteniendo un error por fuera de índice.

En conclusión no es buena idea iterar con `for .. in lista` o `for .. in range` en una lista de la que se prevé eliminar elementos de ella.

Veamos ejemplos de como hacerlo de forma correcta, esto se hará mediante el uso de la instrucción `while`.

```
229 lista = ['a','b','c', 'b']
230 while lista.count('b') != 0:
231     lista.remove('b')
232 print(lista)
```

```
['a', 'c']
```

### Uso de `pop()`

El método `pop()` se utiliza para eliminar el último elemento de una lista. A diferencia de `remove()`, este método retorna el elemento que se ha eliminado. Si la lista está vacía intentar borrar un elemento con `pop()` genera un error que debe controlarse.

También se puede utilizar `pop()` para eliminar un elemento de una posición específica dentro de la lista.

Veamos a continuación ejemplos de uso de `pop()`

```
279 lista = ['a','b','c','d']
280 caracter_borrado = lista.pop()
281 print(lista)
282 print(caracter_borrado)
```

```
['a', 'b', 'c']
d
```

```
284 lista = []
285 caracter_borrado = lista.pop()
286 print(lista)
```

```
Traceback (most recent call last)
File "/Users/rghinf/ACADEMICO/C...
caracter_borrado = lista.pop()
IndexError: pop from empty list
```

```
288 lista = ['a','b','c','d']
289 caracter_borrado = lista.pop(2)
290 print(lista)
291 print(caracter_borrado)
```

```
['a', 'b', 'd']
c
```

En el primer código se muestra el uso normal del método de `pop()`, notar que en la variable `caracter_borrado` se retorna el caracter que se elimina.



En el segundo código se muestra que pasa si se intenta borrar cuando la lista está vacía.

En el tercero se muestra el uso de `pop()` para borrar un elemento en una posición determinada. Tener en cuenta que si la posición indicada es mayor que la longitud de la lista se producirá un error que debe ser tratado.

## Buscar elementos de una lista

Otra de las operaciones más comunes en el uso de la lista es la búsqueda de elementos. Veamos los métodos que tiene Python para realizarlo.

### Uso de `index()`

El método `index()` permite buscar elementos en una lista, este método retorna la posición en la que se encuentra la primera ocurrencia del caracter buscado dentro de la lista.

Veamos unos ejemplos para demostrar su uso:

```
297 lista = ['a','b','c','d','c']
298 pos = lista.index('c')
299 print(f'la posición {pos}')
```

la posición 2

```
301 lista = ['a','b','c','d','c']
302 pos = lista.index('x')
303 print(f'la posición {pos}')
```

```
Traceback (most recent call last)
File "/Users/rgghinf/ACADEMICO
pos = lista.index('x')
ValueError: 'x' is not in list
```

Para evitar el error se puede usar `try` o el método `count()`, veamos ejemplos de su uso:

```
307 lista = ['a','b','c','d','c']
308 try:
309     print(lista.index('x'))
310 except:
311     print('Error el elto no está en la lista')
```

Error el elto no está en la lista

```
313 lista = ['a','b','c','d','c']
314 if lista.count('x') != 0:
315     print(lista.index('x'))
316 else:
317     print('El elto no está en la lista')
```

El elto no está en la lista

### Otros parámetros en `index()`

`index()` tiene un segundo parámetro que indica la posición desde la cual se quiere buscar el elemento.

<pre> 322 lista = ['a','b','c','d','c'] 323 pos = lista.index('c', 3) 324 print(f'La posición es: {pos}') </pre>	<pre> La posición es: 4 </pre>
<pre> 326 lista = ['a','b','c','d','c'] 327 pos = lista.index('c', 7) 328 print(f'La posición es: {pos}') </pre>	<pre> Traceback (most recent call last) File "/Users/rghinf/ACADEMICO/ pos = lista.index('c', 7) ValueError: 'c' is not in list </pre>
<pre> 331 lista = ['a','b','c','d','3','a','c'] 332 pos = lista.index('c', 3, 7) 333 print(f'La posición es: {pos}') </pre>	<pre> La posición es: 6 </pre>

En el primero de los códigos se muestra el uso de `index()` con dos parámetros el segundo de los cuales muestra la posición desde la que comienza la búsqueda.

El segundo de los códigos indica una posición mayor o igual que la longitud de la lista y se muestra el error que se produce al ejecutarlo.

El tercero de los códigos muestra que `index()` tiene un tercer parámetro que indica la posición final de la búsqueda (recordar que siempre es uno menos que el valor indicado).

## Ordenar los elementos de una lista

Los elementos de una lista pueden ser ordenados siempre y cuando todos los elementos que contengan sean del mismo tipo. Veamos un ejemplo de como funcionan el método y la función que permiten realizarlo en Python.

### Uso de `sort()`

El método `sort()` ordena los elementos de la lista en la que se ejecuta. Tiene en cuenta que no se crea una lista nueva al usarlo. Veamos ejemplo de su uso. Evidentemente los elementos que se quieran ordenar deben ser todos del mismo tipo.

<pre> 355 lista = ['e','b','c','d','f'] 356 lista.sort() 357 print(lista) </pre>	<pre> ['b', 'c', 'd', 'e', 'f'] </pre>
<pre> 359 lista = [3,-2,5,-4,3, 0] 360 lista.sort() 361 print(lista) </pre>	<pre> [-4, -2, 0, 3, 3, 5] </pre>

```

363 lista = [3,-2,5,'z',3, 0]
364 lista.sort()
365 print(lista)

```

```

Traceback (most recent call last):
  File "/Users/rghinf/ACADEMICO/CURSO_24_25/PRO_2024/COD-PYTHON/U
    lista.sort()
    ~~~~~^
TypeError: '<' not supported between instances of 'str' and 'int'

```

Si queremos invertir el orden creciente que por defecto aplica el método `sort()` se debe utilizar el parámetro `reverse` con el valor `True` tal y como se muestra en la imagen siguiente:

```

367 lista = ['e','b','c','d','f']
368 lista.sort(reverse = True)
369 print(lista)

```

```
['f', 'e', 'd', 'c', 'b']
```

### Uso de `sorted()`

`sorted()` es una función que permite crear una lista ordenada a partir de la lista origen que se le pasa como parámetro. Al igual que el método `sort()` el orden predeterminado es creciente, para cambiarlo se debe incluir el parámetro `reverse = True`. Veamos ejemplos de su uso.

```

377 lista1 = [10,2,30,4,25]
378 lista2 = sorted(lista1)
379 print(f'lista1 -> {lista1}')
380 print(f'lista2 -> {lista2}')

```

```

lista1 -> [10, 2, 30, 4, 25]
lista2 -> [2, 4, 10, 25, 30]

```

```

382 lista1 = [10,2,30,4,25]
383 lista2 = sorted(lista1, reverse=True)
384 print(f'lista1 -> {lista1}')
385 print(f'lista2 -> {lista2}')

```

```

lista1 -> [10, 2, 30, 4, 25]
lista2 -> [30, 25, 10, 4, 2]

```

### Invertir los elementos de una lista

Los elementos de una lista pueden ser invertidos siempre y cuando todos los elementos que contengan sean del mismo tipo. Veamos un ejemplo de como funcionan el método y la función que permiten realizarlo en Python.

### Uso de `reverse()`

El método `reverse()` invierte los elementos de la lista en la que se ejecuta. Tiene en cuenta que no se crea una lista nueva al usarlo. Veamos ejemplo de su uso.

```

389 lista = ['e','b','c','d','p','z']
390 lista.reverse()
391 print(lista)

```

```
['z', 'p', 'd', 'c', 'b', 'e']
```

```
393 lista = [10,2,30,4,25]
394 lista.reverse()
395 print(lista)
```

```
[25, 4, 30, 2, 10]
```

### Uso de reversed()

reversed() es una función que permite crear una lista que invierte los elementos de la lista origen que se le pasa como parámetro. En este caso reversed retorna un iterador al que debemos pasarlo a la función list() para generar la lista invertida. Veamos ejemplos de su uso.

```
401 lista1 = [10,2,'0',4,25]
402 lista3 = list(reversed(lista1))
403 print(f'lista1 -> {lista1}')
404 print(f'lista3 -> {lista3}')
```

```
lista1 -> [10, 2, '0', 4, 25]
lista3 -> [25, 4, '0', 2, 10]
```

### Eliminar todos los elementos de una lista

En Python hay un método que permite eliminar todos los elementos de una lista, este método es clear().

### Uso de clear()

clear() es un método que elimina todos los elementos de la lista. No genera una nueva lista. Veamos su uso.

```
408 lista = ['e','b','c','d',2,'z']
409 lista.clear()
410 print(f'lista -> {lista}')
```

```
lista -> []
```

### Copia avanzada de una lista

Al comienzo del tema vimos un ejemplo de como copiar una lista mediante el método de asignación y vimos los problemas que se produce al modificar sus elementos.

Recordamos el ejemplo:

```

414 a = [2,3]
415 b = a
416 print(f'lista a -> {a}')
417 print(f'lista b -> {b}')
418 a[0] = 9
419 print(f'lista a después -> {a}')
420 print(f'lista b después -> {b}')

```

```

lista a -> [2, 3]
lista b -> [2, 3]
lista a después -> [9, 3]
lista b después -> [9, 3]

```

Como se puede apreciar al cambiar un elemento de la lista en a o en b el cambio se reproduce en ambas listas.

Veamos como solucionar este problema

### Uso de copy()

Con este método se realiza una copia de la lista que evita que los elementos inmutables se puedan cambiar sin que afecte los cambios a ambas listas. Veamos un ejemplo.

```

423 a = [2,3]
424 b = a.copy()
425 print(f'lista a -> {a}')
426 print(f'lista b -> {b}')
427 a[0] = 9
428 b[1] = '*'
429 print(f'lista a después -> {a}')
430 print(f'lista b después -> {b}')

```

```

lista a -> [2, 3]
lista b -> [2, 3]
lista a después -> [9, 3]
lista b después -> [2, '*']

```

En el caso de que los elementos sean mutables, la copia realizada con el método copy() no funciona correctamente y reproduce el mismo error que la copia realizada con la asignación. Veamos un ejemplo del fallo.

```

433 a = [ [2], [3] ]
434 b = a.copy()
435 print(f'lista a -> {a}')
436 print(f'lista b -> {b}')
437 a[0][0] = 9
438 b[1][0] = '*'
439 print(f'lista a después -> {a}')
440 print(f'lista b después -> {b}')

```

```

lista a -> [[2], [3]]
lista b -> [[2], [3]]
lista a después -> [[9], ['*']]
lista b después -> [[9], ['*']]

```

Para solucionar este problema Python proporciona el método `deepcopy()`.

### Uso de `deepcopy()`

El método `deepcopy()` permite realizar una copia profunda de la lista al que se aplica. Esto provoca que los elementos que son mutables queden protegidos de los cambios que se realicen. Veamos un ejemplo de su uso. Para usar este método se debe importar la librería `copy`.

```
445 import copy
446 from copy import deepcopy
447 a = [ [2], [3] ]
448 b = copy.deepcopy(a)
449 print(f'lista a -> {a}')
450 print(f'lista b -> {b}')
451 a[0][0] = 9
452 b[1][0] = '*'
453 print(f'lista a después -> {a}')
454 print(f'lista b después -> {b}')
```

```
lista a -> [[2], [3]]
lista b -> [[2], [3]]
lista a después -> [[9], [3]]
lista b después -> [[2], ['*']]
```

### Otros métodos a usar en listas `min()`, `max()`, `sum()`

En Python hay métodos que permiten calcular el valor mínimo, máximo y la suma de todos los elementos de una lista. Veamos ejemplos de su uso.

```
457 lista = [1,2,3,4,5]
458 print(f'el menor valor en la lista {min(lista)}')
459 print(f'el menor valor en la lista {max(lista)}')
460 print(f'el menor valor en la lista {sum(lista)}')
```

```
el menor valor en la lista 1
el menor valor en la lista 5
el menor valor en la lista 15
```

### Métodos avanzados para usar en listas `zip()`, `filter()`, `map()`

#### Uso de `zip()`

`zip()` es un método que retorna un iterador que genera tuplas con los elementos de varias listas que se le pasan como parámetros. La primera tupla generada tiene los primeros elementos de cada lista, la segunda los segundos y así sucesivamente. Si las listas no tienen la misma longitud el iterador genera tantas tuplas como la lista de menor longitud.

Veamos unos ejemplos.

```

464 lista1 = [1,2,3,4,5]
465 lista2 = ['a','b','c','d','e']
466 lista3 = ['+', '*', '@', '%', '#']
467 lista_tuplas = zip(lista1, lista2, lista3)
468 for elto1, elto2, elto3 in lista_tuplas:
469     print(f'Los elementos son ->: {elto1}, {elto2}, {elto3}')

```

```

Los elementos de la tupla son ->: 1, a, +
Los elementos de la tupla son ->: 2, b, *
Los elementos de la tupla son ->: 3, c, @
Los elementos de la tupla son ->: 4, d, %
Los elementos de la tupla son ->: 5, e, #

```

```

472 lista1 = [1,2,3,4,5]
473 lista2 = ['a','b','c','d','e']
474 lista3 = ['+', '*', '@', '%', '#']
475 lista_tuplas = zip(lista1, lista2, lista3)
476 for tupla in lista_tuplas:
477     print(tupla)

```

```

(1, 'a', '+')
(2, 'b', '*')
(3, 'c', '@')
(4, 'd', '%')
(5, 'e', '#')

```

```

480 lista1 = [1,2,3,4]
481 lista2 = ['a','b','c','d','e']
482 lista3 = ['+', '*', '@', '%', '#']
483 lista_tuplas = zip(lista1, lista2, lista3)
484 for tupla in lista_tuplas:
485     print(tupla)

```

```

(1, 'a', '+')
(2, 'b', '*')
(3, 'c', '@')
(4, 'd', '%')

```

### Uso de map()

La función `map()` ejecuta una función dada a cada elemento de un iterable (como listas, tuplas, etc.).

Veamos un ejemplo de su uso

```

491 lista = [1,2,3,4]
492
493 def potencia_2(number):
494     return number * number
495
496 iterador_cuadrados = map(potencia_2, lista)
497 # convertimos en lista
498 result = list(iterador_cuadrados)
499 print(result)

```

```
[1, 4, 9, 16]
```

En el ejemplo anterior se ha definido la función `potencia_2` para calcular el cuadrado del número que se le pasa como parámetro. Esta función es la que se va a usar con la función `map` para

aplicársela a cada uno de los elementos de la lista obteniendo así un iterador que luego pasaremos a una lista.

### Uso de filter

La función `filter()` selecciona elementos de un iterable basándose en el resultado de una función. Veamos un ejemplo de su uso

```
504 def es_par(numero):
505     if numero % 2 == 0:
506         return True
507     return False
508
509 numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
510
511 # si un elemento pasado a es_par()
512 # retorna True se selecciona
513 iterator_pares = filter(es_par, numeros)
514
515 lista_pares = list(iterator_pares)
516 print(lista_pares)
```

```
[2, 4, 6, 8, 10]
```