# 09 y 11 - Lambda Functions, Tuples, Lists, Aliasing and Cloning

## Tipos estructurados y mutabilidad

Los tipos numéricos `int` y `float` son **tipos escalares**. Es decir, los objetos de estos tipos no tienen una estructura interna accesible. En cambio, `str` puede considerarse un **tipo estructurado**, o **no escalar**. Podemos utilizar la indexación para extraer caracteres individuales de una cadena y la segmentación para extraer subcadenas.

Ahora vamos a introducir cuatro tipos estructurados adicionales. Uno, `tuple`, es una simple generalización de `str`. Los otros tres –`list`, `range` y `dict`– son más interesantes.

---

## Tuplas

Al igual que las cadenas, las tuplas son **secuencias ordenadas inmutables de elementos**. La diferencia es que los elementos de una tupla no tienen por qué ser caracteres. Los elementos individuales pueden ser de cualquier tipo, y no es necesario que sean del mismo tipo entre sí.

Los literales de tipo tupla *se escriben encerrando entre paréntesis una lista de elementos separados por comas*. Por ejemplo, podemos escribir

```
1  t1 = ()
2  t2 = (1, 'two', 3)
3  print(t1)
4  print(t2)
```

Como era de esperar, las sentencias print producen la salida

```
1  ()
2  (1, 'two', 3)
```

Viendo este ejemplo, se podría pensar que la tupla que contiene el único valor `1` se escribiría `(1)`. Pero esto seria incorrecto dado que los paréntesis se utilizan para agrupar expresiones, `(1)` no es más que una forma verbosa de escribir el número entero `1`. Para denotar la

tupla única que contiene este valor, escribimos `(1,)`. Casi todo el mundo que usa Python ha omitido accidentalmente alguna vez esa molesta coma.

La repetición puede utilizarse en tuplas. Por ejemplo, la expresión `3* (“a”, 2)` se evalúa como `(“a”, 2, “a”, 2, “a”, 2)`.

Al igual que las cadenas, las tuplas pueden concatenarse, indexarse y rebanarse.

```
t1 = (1, ‘two', 3)
t2 = (t1, 3.25)
print(t2)
print((t1 + t2))
print((t1 + t2)[3])
print((t1 + t2)[2:5])
```

La segunda sentencia de asignación asocia el nombre `t2` a una tupla que contiene la tupla a la que está asociada `t1` y el número de coma flotante `3.25`. Esto es posible porque una tupla, como todo en Python, es un objeto, por lo que las tuplas pueden contener tuplas. Por lo tanto, la primera sentencia `print` produce la salida,

```
((1, 'two', 3), 3.25)
```

La segunda sentencia `print` imprime el valor generado al concatenar los valores ligados a `t1` y `t2`, que es una tupla con cinco elementos. Produce la salida

```
(1, 'two', 3, (1, 'two', 3), 3.25)
```

La siguiente sentencia selecciona e imprime el cuarto elemento de la tupla concatenada (como siempre en Python, la indexación empieza en 0), y la sentencia siguiente crea e imprime una porción de esa tupla, produciendo la salida

```
(1, 'two', 3)
(3, (1, 'two', 3), 3.25)
```

Una sentencia `for` puede utilizarse para iterar sobre los elementos de una tupla. Y el operador `in` puede utilizarse para comprobar si una tupla contiene un valor específico. Por ejemplo, el código siguiente

```
def intersect(t1, t2):
    """Assumes t1 and t2 are tuples
    Returns a tuple containing elements that are in both t1 and t2"""

```

```
    result = ()

    for e in t1:
        if e in t2:
            result += (e,)

    return result

print(intersect((1, 'a', 2), ('b', 2, 'a')))
```

imprime `("a", 2)`.

## Asignaciones múltiples

Si conoces la longitud de una secuencia (por ejemplo, una tupla o una cadena), puede ser conveniente utilizar la sentencia de asignación múltiple de Python para extraer los elementos individuales. Por ejemplo, la sentencia `x, y = (3, 4)`, asociará `x` a `3` e `y` a `4`. Del mismo modo, la sentencia `a, b, c = “xyz”` asociará `a` a `“x”`, `b` a `“y”`, y `c` a `“z”`.

Este mecanismo resulta especialmente práctico cuando se utiliza con funciones que devuelven varios valores. Considere la definición de la función:

```
def find_extreme_divisors(n1, n2):
    """Assumes that n1 and n2 are positive ints
    Returns a tuple containing the smallest common divisor > 1 and
    the largest common divisor of n1 & n2. If no common divisor,
    other than 1, returns (None, None)"""

    min_val, max_val = None, None

    for i in range(2, min(n1, n2) + 1):
        if n1 % i == 0 and n2 % i == 0:
            if min_val == None:
                min_val = i
            max_val = i

    return min_val, max_val
```

La sentencia de asignación múltiple

```
min_divisor, max_divisor = find_extreme_divisors(100, 200)
```

vinculará `min_divisor` a `2` y `max_divisor` a `200`.

# Rangos e iterables

La función `range()` produce un objeto de tipo `range`. Al igual que las cadenas y las tuplas, los objetos de tipo `range` son inmutables. Todas las operaciones de las tuplas también están disponibles para los rangos, excepto la concatenación y la repetición. Por ejemplo, `range(10)[2:6][2]` se evalúa como `4`. Cuando se utiliza el operador `==` para comparar objetos de tipo rango, devuelve `True` si los dos rangos representan la misma secuencia de enteros. Por ejemplo, `range(0, 7, 2) == range(0, 8, 2)` se evalúa como `True`. Sin embargo, `range(0, 7, 2) == range(6, -1, -2)` es `False` porque, aunque los dos rangos contienen los mismos números enteros, aparecen en un orden diferente.

A diferencia de los objetos de tipo tupla, la cantidad de espacio que ocupa un objeto de tipo rango no es proporcional a su longitud. Dado que un rango está totalmente definido por sus valores de inicio, parada y paso, puede almacenarse en un espacio reducido.

El uso más común de `range` es en los bucles `for`, pero los objetos de tipo `range` se pueden utilizar en cualquier lugar donde se pueda utilizar una secuencia de enteros.

En Python 3, `range` es un caso especial de un objeto iterable. Todos los tipos iterables tienen un método, `__iter__` que devuelve un objeto de tipo iterador. El iterador puede usarse en un bucle `for` para devolver una secuencia de objetos, uno cada vez. Por ejemplo, las tuplas son iterables, y la sentencia `for`

```
for elem in (1, 'a', 2, (3, 4)):
```

crea un iterador que devolverá los elementos de la tupla de uno en uno. Python tiene muchos tipos iterables incorporados, incluyendo cadenas, listas y diccionarios.

Muchas funciones incorporadas útiles operan sobre iterables. Entre las más útiles están `sum`, `min` y `max`. La función suma puede aplicarse a iterables de números. Devuelve la suma de los elementos. Las funciones max y min pueden aplicarse a iterables para los que existe un orden bien definido en los elementos.

### Ejercicio manual

Escribe una expresión que evalúe la media de una tupla de números. Utiliza la función `sum`.

```
media = sum(mi_tupla) / len(mi_tupla)
```

---

# Listas y mutabilidad

Al igual que una tupla, una lista es una *secuencia ordenada de valores*, en la que *cada valor se identifica mediante un índice*. La sintaxis para expresar literales de tipo lista es similar a la utilizada para las tuplas; la diferencia es que utilizamos corchetes en lugar de paréntesis. La lista vacía se escribe como `[]`, y las listas simples se escriben sin esa coma (tan fácil de olvidar) que precede al corchete de cierre.

Dado que las listas son iterables, podemos utilizar una sentencia `for` para iterar sobre los elementos de la lista. Así, por ejemplo, el código

```
L = ['I did it all', 4, 'love']

for e in L:
    print(e)
```

produce la salida,

```
I did it all
4
Love
```

También podemos indexar en listas y trocear listas, igual que en el caso de las tuplas. Por ejemplo, el código

```
L1 = [1, 2, 3]
L2 = L1[-1::-1]

for i in range(len(L1)):
    print(L1[i] * L2[i])
```

imprime

```
3
4
3
```

El uso de corchetes para tres propósitos diferentes (literales de tipo lista, indexación en iterables y corte de iterables) puede llevar a confusión visual. Por ejemplo, la expresión `[1,2,3,4][1:3][1]`, que

evalúa a `3`, utiliza corchetes de tres maneras. Esto no suele ser un problema en la práctica, porque la mayoría de las veces las listas se construyen de forma incremental en lugar de escribirse como literales.

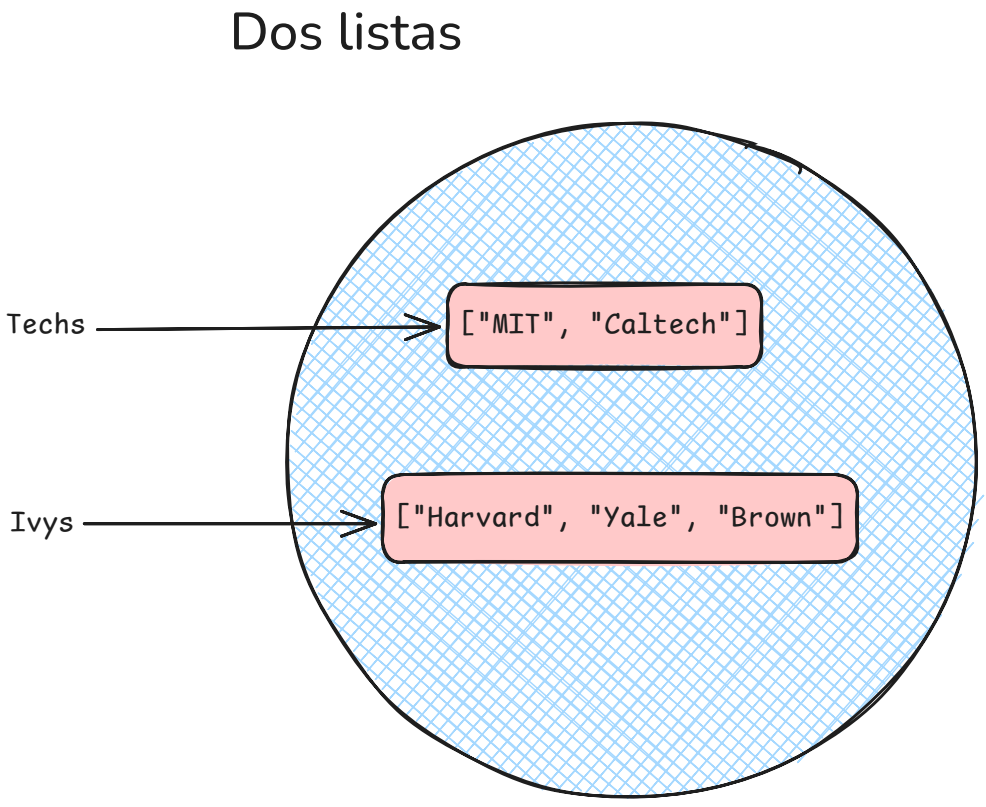
Las listas difieren de las tuplas en un aspecto muy importante: **las listas son mutables**. En cambio, las tuplas y las cadenas son inmutables. Muchos operadores pueden utilizarse para crear objetos de tipos inmutables, y las variables pueden vincularse a objetos de estos tipos. Pero los objetos de tipos inmutables no pueden modificarse una vez creados. En cambio, los objetos de tipo mutable pueden modificarse después de su creación.

La distinción entre mutar un objeto y asignar un objeto a una variable puede, al principio, parecer sutil. Sin embargo, si sigues repitiendo el mantra: «**En Python una variable es simplemente un nombre, es decir, una etiqueta que se puede adjuntar a un objeto**», te aportará claridad. Y quizá la siguiente serie de ejemplos también te ayude.

Cuando las declaraciones

```
1   Techs = ['MIT', 'Caltech']
2   Ivys = ['Harvard', 'Yale', 'Brown']
```

el intérprete crea dos nuevas listas y les asigna las variables apropiadas, como se muestra en la siguiente imagen.




Las declaraciones de asignación

```
Univs = [Techs, Ivys]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
```

también crean nuevas listas y vinculan variables a ellas. Los elementos de estas listas son a su vez listas. Las tres sentencias `print`

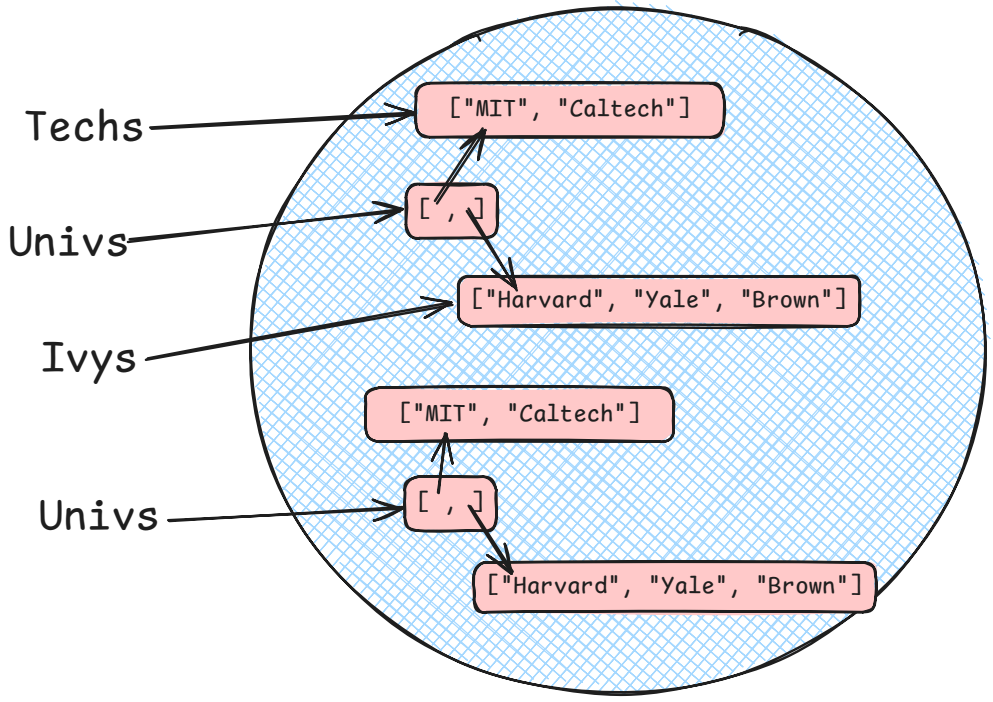
```
print('Univs =', Univs)
print('Univs1 =', Univs1)
print(Univs == Univs1)
```

produce la salida

```
Univs = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
True
```

Parece como si `Univs` y `Univs1` estuvieran vinculadas al mismo valor. Pero las apariencias engañan. Como muestra la siguiente imagen, `Univs` y `Univs1` están vinculados a valores muy diferentes.

Dos listas que parecen tener el mismo valor, pero no lo tienen




Que `Univs` y `Univs1` están vinculados a objetos diferentes puede verificarse utilizando la función incorporada de Python `id`, que devuelve un identificador entero único para un objeto. Esta función nos permite comprobar la igualdad de los objetos comparando sus id.

Una forma más sencilla de comprobar la igualdad de objetos es utilizar el operador `is`. Cuando ejecutamos el código

```
print(Univs == Univs1) #test value equality
print(id(Univs) == id(Univs1)) #test object equality
print(Univs is Univs1) #test object equality
print('Id of Univs =', id(Univs))
print('Id of Univs1 =', id(Univs1))
```

Imprimirá

```
True
False
False
Id of Univs = 4946827936
Id of Univs1 = 4946612464
```

(No esperes ver los mismos identificadores únicos si ejecutas este código. La semántica de Python no dice nada sobre qué identificador se asocia a cada objeto; simplemente requiere que no haya dos objetos con el mismo identificador).

Observa que los elementos de `Univs` no son copias de las listas a las que están vinculadas `Techs` e `Ivys`, sino que son las propias listas. Los elementos de `Univs1` son listas que contienen los mismos elementos que las listas de `Univs`, pero no son las mismas listas. Podemos comprobarlo ejecutando el código

```
print('Ids of Univs[0] and Univs[1]', id(Univs[0]), id(Univs[1]))
print('Ids of Univs1[0] and Univs1[1]', id(Univs1[0]), id(Univs1[1]))
```

Que imprimirá

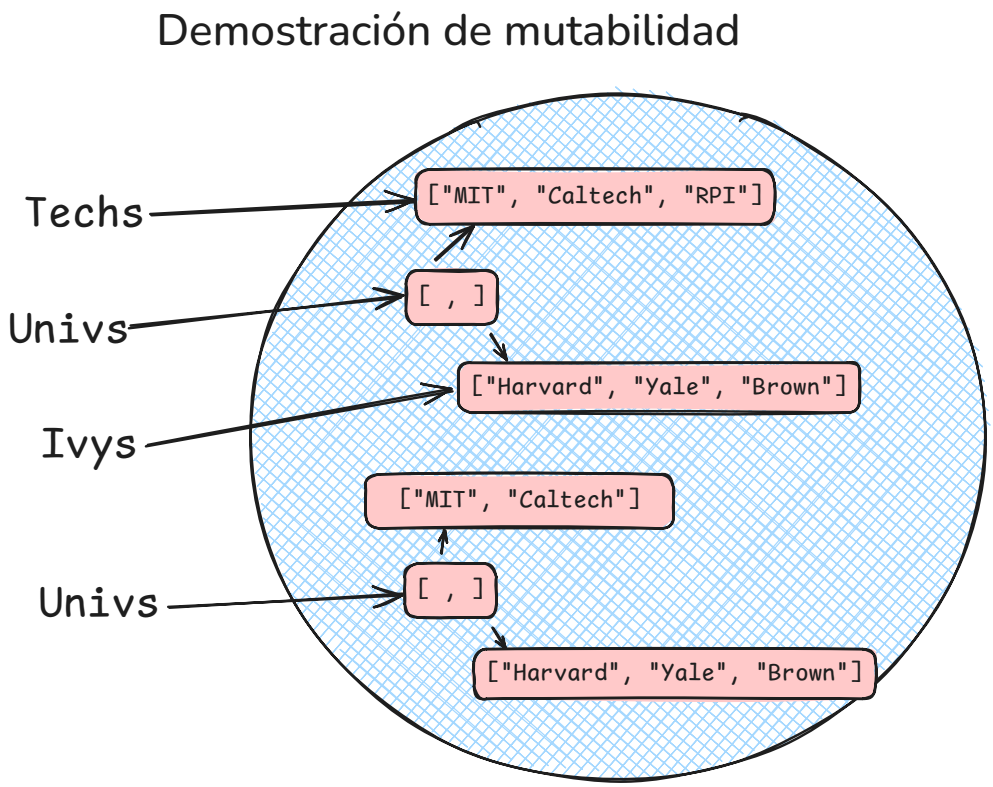
```
Ids of Univs[0] and Univs[1] 4447807688 4456134664
Ids of Univs1[0] and Univs1[1] 4447805768 4447806728
```

¿Por qué tanto alboroto por la diferencia entre igualdad de valores y de objetos? Es importante porque las listas son mutables. Considere el código

```
Techs.append('RPI')
```

El método `append` para listas tiene un efecto secundario. En lugar de crear una nueva lista, muta la lista existente, `Techs`, añadiendo un nuevo elemento, la cadena “RPI” en este ejemplo, al final de la misma.

La siguiente imagen muestra el estado del cálculo después de ejecutar `append`.




El objeto al que está vinculado `Univs` sigue conteniendo las mismas dos listas, pero el contenido de una de ellas ha cambiado. En consecuencia, las sentencias `print`

```
print('Univs =', Univs)
print('Univs1 =', Univs1)
```

Ahora produce la siguiente salida

```
Univs = [['MIT', 'Caltech', 'RPI'], ['Harvard', 'Yale', 'Brown']]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
```

Lo que tenemos aquí se llama **aliasing**.

> **Aliasing**
>
> **Aliasing** es una situación en la que **dos o más variables o caminos hacen referencia al mismo objeto en memoria**. Cualquier cambio (mutación) hecho a través de uno de esos caminos **afecta al objeto compartido**, y por tanto se ve reflejado desde todos los alias.

Hay dos caminos distintos hacia el mismo objeto de lista. Un camino es a través de la variable `Techs` y el otro a través del primer elemento

del objeto de lista al que está vinculado `Univs`. Podemos mutar el objeto a través de cualquiera de los dos caminos, y el efecto de la mutación será visible a través de ambos caminos. Esto puede ser conveniente, pero también traicionero. El aliasing involuntario conduce a errores de programación que a menudo son enormemente difíciles de localizar. Por ejemplo, ¿Qué crees que se imprime con

```
L1 = [[]] * 2
L2 = [[], []]

for i in range(len(L1)):
    L1[i].append(i)
    L2[i].append(i)

print('L1 =', L1, 'but', 'L2 =', L2)
```

Imprime `L1 = [[0, 1], [0, 1]]` pero `L2 = [[0], [1]]`. ¿Por qué? Porque la primera sentencia de asignación crea una lista con dos elementos, cada uno de los cuales es el mismo objeto, mientras que la segunda sentencia de asignación crea una lista con dos objetos diferentes, cada uno de los cuales es inicialmente igual a una lista vacía.

**Ejercicio manual**

¿Qué hace el siguiente código?

```
L = [1, 2, 3]
L.append(L)
print(L is L[-1])
```

**Mi aproximación al ejercicio**:

La primera línea crea una lista `L` con los elementos `[1, 2, 3]`.
La segunda línea, `L.append(L)`, agrega **la propia lista `L` como último elemento** de sí misma.
La tercera línea evalúa `L is L[-1]`. Esto pregunta si el último elemento de la lista (`L[-1]`) es el mismo objeto en memoria que la propia lista `L`. Como lo que se agregó fue exactamente la lista `L`, **ambos apuntan al mismo objeto**, por lo tanto la comparación `is` devuelve `True`.

La interacción de aliasing y mutabilidad con valores de parámetros por defecto es algo a tener en cuenta. Considere el código

```
def append_val(val, list_1 = []):
    List_1.append(val)
```

```
    print(list_1)

append_val(3)
append_val(4)
```

Podrías pensar que la segunda llamada a `append_val` imprimiría la lista `[4]` porque habría añadido 4 a la lista vacía. En realidad, imprimirá `[3, 4]`. Esto ocurre porque, en el momento de definición de la función, se crea un nuevo objeto de tipo lista, con un valor inicial de la lista vacía. Cada vez que se invoca `append_val` sin proporcionar un valor para el parámetro formal `list_1`, el objeto creado en la definición de la función se vincula a `list_1`, muta y luego se imprime. Así, la segunda llamada a `append_val` muta y luego imprime una lista que ya fue mutada por la primera llamada a esa función.

Cuando añadimos una lista a otra, por ejemplo, `Techs.append(Ivys)`, se mantiene la estructura original. El resultado es una lista que contiene una lista. Supongamos que no queremos mantener esta estructura, sino añadir los elementos de una lista a otra. Podemos hacerlo utilizando la concatenación de listas (utilizando el operador `+`) o el método `extend`, por ejemplo

```
L1 = [1,2,3]
L2 = [4,5,6]
L3 = L1 + L2
print('L3 =', L3)

L1.extend(L2)
print('L1 =', L1)

L1.append(L2)
print('L1 =', L1)
```

Imprimira

```
L3 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]]
```

Observe que el operador `+` no tiene ningún efecto secundario. *Crea una nueva lista y la devuelve*. Por el contrario, `extend` y `append` *mutan* `L1`.

Se describe brevemente algunos de los métodos asociados a las listas. Observe que todos ellos, excepto `count` e `index`, mutan la lista.

- `L.append(e)` añade el objeto `e` al final de `L`.
- `L.count(e)` devuelve el numero de veces que `e` ocurre en `L`.

- `L.insert(i, e)` inserta el objeto `e` en `L` en el índice `i`.
- `L.extend(L1)` añade los ítems en la lista `L1` al final de `L`.
- `L.remove(e)` elimina la primer ocurrencia de `e` desde `L`.
- `L.index(e)` devuelve el índice de la primera ocurrencia de `e` en `L`, y muestra una excepción si `e` no esta en `L`.
- `L.pop(i)` remueve y devuelve el ítem en el índice `i` en `L`, muestra una excepción si `L` esta vacío. Si `i` esta omitido, por defecto es `-1`, para remover y devolver el ultimo elemento de `L`.
- `L.sort()` ordena los elementos de `L` en orden ascendente.
- `L.reverse()` revierte el orden de los elementos en `L`.

## Clonado

Normalmente *es prudente evitar mutar una lista sobre la que se está iterando*. Considere el código

```
1   def remove_dups(L1, L2):
2       """Assumes that L1 and L2 are lists.
3       Removes any element from L1 that also occurs in L2"""
4
5       for e1 in L1:
6           if e1 in L2:
7               L1.remove(e1)
8
9   L1 = [1,2,3,4]
10  L2 = [1,2,5,6]
11  Remove_dups(L1, L2)
12  print('L1 =', L1)
```

Le sorprenderá descubrir que imprimirá

```
1   L1 = [2, 3, 4]
```

Durante un bucle `for`, Python lleva la cuenta de dónde se encuentra en la lista utilizando un contador interno que se incrementa al final de cada iteración. Cuando el valor del contador alcanza la longitud actual de la lista, el bucle termina. Esto funciona como cabría esperar si la lista no se muta dentro del bucle, pero puede tener consecuencias sorprendentes si la lista se muta. En este caso, el contador oculto comienza en 0, descubre que `L1[0]` está en `L2`, y lo elimina, reduciendo la longitud de `L1` a 3. El contador se incrementa entonces a 1, y el código procede a comprobar si el valor de `L1[1]` está en `L2`. Observa que este no es el valor original de `L1[1]` (es decir, 2), sino el valor actual de `L1[1]` (es decir, 3). Como puedes ver, es posible averiguar qué ocurre cuando se modifica la lista

dentro del bucle. Sin embargo, no es fácil. Y es probable que lo que ocurra no sea intencionado, como en este ejemplo.

Una forma de evitar este tipo de problema es utilizar el troceado para clonar (es decir, hacer una copia de) la lista y escribir `for e1 in L1[:]`. Tenga en cuenta que escribir

```
new_L1 = L1
for e1 in new_L1:
```

no resolvería el problema. **No crearía una copia de L1**, sino que simplemente introduciría un nuevo nombre para la lista existente.

El rebanado no es la única forma de clonar listas en Python. La expresión `L.copy()` tiene el mismo valor que `L[:]`. Tanto `slicing` como `copy` realizan lo que se conoce como **copia superficial**. Una copia superficial *crea una nueva lista y luego inserta los objetos* (no copias de los objetos) de la lista a copiar en la nueva lista. El código

```
L = [2]
L1 = [L]
L2 = L1[:]
L2 = copy.deepcopy(L1)
L.append(3)
print(f'L1 = {L1}, L2 = {L2}')
```

imprime `L1 = [[2, 3]]` `L2 = [[2, 3]]` porque tanto `L1` como `L2` contienen el objeto que estaba ligado a `L` en la primera sentencia de asignación.

Si la lista a copiar contiene objetos mutables que también desea copiar, importe el módulo `copy` de la biblioteca estándar y utilice la función `copy.deepcopy` para realizar una copia profunda. El método `deepcopy` crea una nueva lista y luego inserta copias de los objetos de la lista a copiar en la nueva lista. Si sustituimos la tercera línea del código anterior por `L2 = copy.deepcopy(L1)`, se imprimirá `L1 = [[2, 3]]`, `L2 = [[2]]`, porque `L1` no contendría el objeto al que está ligado `L`.

Comprender `copy.deepcopy` es complicado si los elementos de una lista son listas que contienen listas (o cualquier tipo mutable). Consideremos

```
L1 = [2]
L2 = [[L1]]
L3 = copy.deepcopy(L2)
```

```
4   L1.append(3)
```

El valor de `L3` será `[[[2]]]` porque `copy.deepcopy` crea un nuevo objeto no sólo para la lista `[L1]`, sino también para la lista `L1`. Es decir, hace copias hasta el final la mayoría de las veces. ¿Por qué «la mayoría de las veces»? El código

```
1   L1 = [2]
2   L1.append(L1)
```

crea una lista que se contiene a sí misma. Un intento de hacer copias hasta el final nunca terminaría. Para evitar este problema, `copy.deepcopy` hace exactamente una copia de cada objeto, y luego utiliza esa copia para cada instancia del objeto. Esto es importante incluso cuando las listas no se contienen a sí mismas. Por ejemplo,

```
1   L1 = [2]
2   L2 = [L1, L1]
3   L3 = copy.deepcopy(L2)
4   L3[0].append(3)
5   print(L3)
```

imprime `[[2, 3], [2, 3]]` porque `copy.deepcopy` hace una copia de `L1` y la utiliza las dos veces que `L1` aparece en `L2`.

## Comprensión de listas

La comprensión de listas proporciona una forma concisa de aplicar una operación a los valores de secuencia proporcionados al iterar sobre un valor iterable. Crea una nueva lista en la que cada elemento es el resultado de aplicar una operación determinada a un valor de un iterable (por ejemplo, los elementos de otra lista). Es una expresión de la forma

```
1   [expr for elem in iterable if test]
```

Evaluar la expresión equivale a invocar la función

```
1   def f(expr, old_list, test = lambda x: True):
2       new_list = []
3
4       for e in iterable:
5           if test(e):
6               new_list.append(expr(e))
7
8       return new_list
```

Por ejemplo, `[e ** 2 for e in range(6)]` se evalúa en `[0, 1, 4, 9, 16, 25]`, `[e ** 2 for e in range(8) if e % 2 == 0]` se evalúa en `[0, 4, 16, 36]`, y `[x ** 2 for x in [2, "a", 3, 4.0] if type(x) == int]` se evalúa en `[4, 9]`.

La comprensión de listas proporciona una forma cómoda de inicializar listas. Por ejemplo, `[[] for _ in range(10)]` genera una lista que contiene 10 listas vacías distintas (es decir, no suavizadas). El nombre de variable `_` indica que los valores de esa variable no se utilizan para generar los elementos de la lista, es decir, es simplemente un marcador de posición. Esta convención es común en los programas Python.

Python permite múltiples sentencias `for` dentro de una comprensión de lista. Considere el código

```
L = [(x, y)
    for x in range(6) if x % 2 == 0
    for y in range(6) if y % 3 == 0]
```

El intérprete de Python comienza evaluando el primer `for`, asignando a x la secuencia de valores 0,2,4. Para cada uno de estos tres valores de x, evalúa el segundo `for` (que genera cada vez la secuencia de valores 0,3). A continuación, añade a la lista que se está generando la tupla `(x, y)`, produciendo la lista

```
[(0, 0), (0, 3), (2, 0), (2, 3), (4, 0), (4, 3)]
```

Por supuesto, podemos producir la misma lista sin comprensión de lista, pero el código es considerablemente menos compacto:

```
L = []

for x in range(6):
    if x % 2 == 0:
        for y in range(6):
            if y % 3 == 0:
                L.append((x, y))
```

El siguiente código es un ejemplo de anidamiento de una comprensión de lista dentro de otra comprensión de lista.

```
print([[(x,y) for x in range(6) if x % 2 == 0]
    for y in range(6) if y % 3 == 0])
```

Imprimirá `[[(0, 0), (2, 0), (4, 0)], [(0, 3), (2, 3), (4, 3)]]`.

Se necesita práctica para sentirse cómodo con las comprensiones de listas anidadas, pero pueden ser muy útiles. Utilicemos las comprensiones de listas anidadas para generar una lista de todos los números primos menores que 100. La idea básica es usar una comprensión para generar una lista de todos los números candidatos (es decir, del 2 al 99), una segunda comprensión para generar una lista de los restos de dividir un primo candidato entre cada divisor potencial, y la función incorporada `all` para comprobar si alguno de esos restos es 0.

```
[x for x in range(2, 100) if all(x % y != 0 for y in range(3, x))]
```

Evaluar la expresión equivale a invocar la función

```
def gen_primes():
    primes = []

    for x in range(2, 100):
        is_prime = True
        for y in range(3, x):
            if x % y == 0:
                is_prime = False
        if is_prime:
            primes.append(x)
    return primes
```

**Ejercicio manual**

Escribe una comprensión de lista que genere todos los no-primos entre 2 y 100.

```
[n for n in range(2, 101) if any(n % d == 0
    for d in range(2, int(n ** 0.5) + 1))]
```

Evaluó la condición si un numero `n` es divisible por un `d` tal que de resto 0, el segundo `for` me genera todos los posibles divisores de `n` desde 2 hasta la raíz cuadrada de `n`, lo cual si no llegase a tener divisores es primo.

Algunos programadores de Python utilizan las comprensiones de listas de formas maravillosas y sutiles. No siempre es una buena idea. Recuerde que alguien más puede necesitar leer su código, y rara vez es una propiedad deseable para un programa.

---

# Ejercicio manual de la lectura

Implemente la función que cumpla las siguientes especificaciones:

```python
def dot_product(tA, tB):
    """
    tA: a tuple of numbers
    tB: a tuple of numbers of the same length as tA
    Assumes tA and tB are the same length.
    Returns a tuple where the:
    * first element is the length of one of the tuples
    * second element is the sum of the pairwise products of tA and tB
    """
    product_sum = 0
    for i in range(len(tA)):
        product_sum += tA[i] * tB[i]
    return (len(tA), product_sum)

# Examples:
tA = (1, 2, 3)
tB = (4, 5, 6)
print(dot_product(tA, tB)) # prints (3,32)
```

Implemente la función que cumpla las siguientes especificaciones:

```python
def remove_and_sort(Lin, k):
    """ Lin is a list of ints
        k is an int >= 0
    Mutates Lin to remove the first k elements in Lin and
    then sorts the remaining elements in ascending order.
    If you run out of items to remove, Lin is mutated to an empty list.
    Does not return anything.
    """
    if len(Lin) < k:
        Lin.clear()  # vacía la lista si no hay suficientes elementos
    else:
        del Lin[:k]  # elimina los primeros k elementos
    Lin.sort()

# Examples:
L = [1,6,3]
k = 1
remove_and_sort(L, k)
print(L)   # prints the list [3, 6]
```