Secuencias

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2025/07/10 a las 17:31:00

Índice

1. Concepto de secuencia

1.1. Definición

Una **secuencia** *s* es un dato estructurado *iterable* que cumple lo siguiente:

1. Se le puede calcular su longitud (la cantidad de elementos que contiene) mediante la función `len`.
2. Cada elemento que contiene lleva asociado un número entero llamado **índice**, comprendido entre `0` y `len(s) - 1`.
3. Permite el acceso eficiente a cada uno de sus elementos mediante indexación `s[i]`, siendo *i* el índice del elemento.

Las secuencias se dividen en:

- **Inmutables**: cadenas (`str`), tuplas (`tuple`) y rangos (`range`).

- **Mutables**: listas (`list`)

1.2. Operaciones comunes

Todas las secuencias (ya sean cadenas, listas, tuplas o rangos) comparten un conjunto de **operaciones comunes**.

Los rangos son una excepción, ya que sus elementos se crean a partir de una fórmula y, por eso, no admiten ni la concatenación ni la repetición.

La siguiente tabla recoge las operaciones comunes sobre secuencias, ordenadas por prioridad ascendente. *s* y *t* son secuencias del mismo tipo, *n*, *i*, *j* y *k* son enteros y *x* es un dato cualquiera que cumple con las restricciones que impone *s*.

Operación	Resultado
`x in s`	`True` si algún elemento de *s* es igual a *x*
`x not in s`	`False` si algún elemento de *s* es igual a *x*
`s + t`	La concatenación de *s* y *t* (no va con rangos)
`s * n` `n * s`	(*Repetición*) Equivale a concatenar *s* consigo misma *n* veces (no va con rangos)
`s[i]`	El *i*-ésimo elemento de *s*, empezando por 0
`s[i:j]`	Rodaja de *s* desde *i* hasta *j*
`s[i:j:k]`	Rodaja de *s* desde *i* hasta *j* con paso *k*
`len(s)`	Longitud de *s*
`min(s)`	El elemento más pequeño de *s*
`max(s)`	El elemento más grande de *s*
`s.index(x[, i[, j]])`	El índice de la primera aparición de *x* en *s* (desde el índice *i* inclusive y antes del *j*)
`s.count(x)`	Número total de apariciones de *x* en *s*

Además de estas operaciones, las secuencias admiten **comparaciones** con los operadores `==`, `!=`, `<`, `<=`, `>` y `>=`.

Dos secuencias *s* y *t* son iguales (*s* `==` *t*) si:

- Son del mismo tipo (`type(s) == type(t)`).
- Tienen la misma longitud (`len(s) == len(t)`).
- Contienen los mismos elementos en el mismo orden (`s[0] == t[0]`, `s[1] == t[1]`, etcétera).

Por supuesto, las dos secuencias son distintas (*s* `!=` *t*) si no son iguales.

Se pueden comparar dos secuencias con los operadores `<`, `<=`, `>` y `>=` para comprobar si una es menor (o igual) o mayor (o igual) que la otra si:

- Son del mismo tipo (si no son del mismo tipo, lanza una excepción).
- No son rangos.

Las comparaciones `<`, `<=`, `>` y `>=` se hacen lexicográficamente elemento a elemento, como en un diccionario.

Por ejemplo, `'adios' < 'hola'` porque `adios` aparece antes que `hola` en el diccionario.

Con el resto de las secuencias se actúa igual que con las cadenas.

Dadas dos secuencias *s* y *t*, para ver si *s* < *t* se procede así:

- Se empieza comparando el primer elemento de *s* con el primero de *t*.
- Si son iguales, se pasa al siguiente hasta encontrar algún elemento de *s* que sea distinto a su correspondiente de *t*.
- Si llegamos al final de *s* sin haber encontrado ningún elemento distinto a su correspondiente en *t*, es porque *s* == *t*.
- En cuanto se encuentre un elemento de *s* que no es igual a su correspondiente de *s*, se comparan esos elementos y se devuelve el resultado de esa comparación.

Los rangos no se pueden comparar con `<`, `<=`, `>` o `>=`.

Ejemplos:

```
>>> (1, 2, 3) == (1, 2, 3)
True
>>> (1, 2, 3) != (1, 2, 3)
False
>>> (1, 2, 3) == (3, 2, 1)
False
>>> (1, 2, 3) < (3, 2, 1)
True
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3) < (1, 2, 4, 5)
True
>>> 'hola' < 'adios'
False
>>> range(0, 3) < range(3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'range' and 'range'
```

2. Inmutables

2.1. Cadenas (`str`)

Las **cadenas** son secuencias inmutables y *hashables* de caracteres.

No olvidemos que en Python no existe el tipo *carácter*. En Python, un carácter es una cadena de longitud 1.

Las cadenas literales se pueden crear:

- Con comillas simples (`'`) o dobles (`"`):

```
>>> 'hola'
'hola'
>>> "hola"
'hola'
```

- Con triples comillas (`'''` o `"""`):

```
>>> """hola
... qué tal"""
'hola\nqué tal'
```

Las cadenas implementan todas las operaciones de las secuencias, además de los métodos que se pueden consultar en https://docs.python.org/3/library/stdtypes.html#string-methods

2.1.1. Formateado de cadenas

Una **cadena formateada** (también llamada ***f-string***) es una cadena literal que lleva un prefijo `f` o `F`.

Estas cadenas contienen **campos de sustitución**, que son expresiones encerradas entre llaves.

En realidad, las cadenas formateadas son expresiones evaluadas en tiempo de ejecución.

Sintaxis:

```
⟨f_string⟩ ::= (⟨carácter_literal⟩ | {{ | }} | ⟨sustitución⟩)*
⟨sustitución⟩ ::= {⟨expresión⟩ [!⟨conversión⟩] [:⟨especif⟩]}
⟨conversión⟩ ::= s | r | a
⟨especif⟩ ::= (⟨carácter_literal⟩ | NULL | ⟨sustitución⟩)*
  ~~~~~~~~~~~~~~
⟨carácter_literal⟩ ::= ⟨cualquier carácter Unicode excepto {, } o NULL⟩
~~~~~~~~~~~~~~~
```

Las partes de la cadena que van fuera de las llaves se tratan literalmente, excepto las dobles llaves `{{` y `}}`, que son sustituidas por una sola llave.

Una `{` marca el comienzo de un **campo de sustitución** (⟨*sustitución*⟩), que empieza con una expresión.

Tras la expresión puede venir un **conversión** (⟨*conversión*⟩), introducida por una exclamación `!`.

También puede añadirse un **especificador de formato** (⟨*especif*⟩) después de dos puntos `:`.

El campo de sustitución termina con una }.

Las expresiones en un literal de cadena formateada son tratadas como cualquier otra expresión Python encerrada entre paréntesis, con algunas excepciones:

- No se permiten expresiones vacías.
- Las expresiones lambda deben ir entre paréntesis.

Los campos de sustitución pueden contener saltos de línea pero no comentarios.

Si se indica una conversión, el resultado de evaluar la expresión se convierte antes de aplicar el formateado.

La conversión !s llama a la función str sobre el resultado, !r llama a repr y !a llama a ascii.

A continuación, el resultado es formateado usando la función format.

Finalmente, el resultado del formateado es incluido en el valor final de la cadena completa.

La sintaxis general de un especificador de formato es:

⟨*especif*⟩ ::= [[⟨*relleno*⟩]⟨*alig*⟩][⟨*signo*⟩][**#**][**0**][⟨*ancho*⟩][⟨*grupos*⟩][**.**⟨*precision*⟩][⟨*tipo*⟩]
⟨*relleno*⟩ ::= ⟨*cualquier carácter*⟩
⟨*alig*⟩ ::= **<** | **>** | **=** | **^**
⟨*signo*⟩ ::= **+** | **-** | ⟨*espacio*⟩
⟨*ancho*⟩ ::= ⟨*dígito*⟩+
⟨*grupos*⟩ ::= **_** | **,**
⟨*precision*⟩ ::= ⟨*dígito*⟩+
⟨*tipo*⟩ ::= **b** | **c** | **d** | **e** | **E** | **f** | **F** | **g** | **G** | **n** | **o** | **s** | **x** | **X** | **%**

Los especificadores de formato de nivel superior pueden incluir campos de sustitución anidados.

Estos campos anidados pueden incluir, a su vez, sus propios campos de conversión y sus propios especificadores de formato, pero no pueden incluir más campos de sustitución anidados.

Para más información, consultar https://docs.python.org/3.7/library/string.html#format-specification-mini-language

Ejemplos de cadenas formateadas:

```
>>> nombre = 'Pepe'
>>> f'El nombre es: {nombre}'              # Se sustituye la variable por su valor
'El nombre es: Pepe'
>>> apellidos = 'Pérez'
>>> f'El nombre es: {nombre} {apellidos}'  # Igual
'El nombre es: Pepe Pérez'
>>> f'El nombre es: {nombre + apellidos}'  # Se puede usar cualquier expresión
'El nombre es: Pepe Pérez'
>>> f'Formato con anchura: {nombre:10}'    # Las cadenas se alinean a la izquierda
'Formato con anchura: Pepe      '
>>> f'Formato con anchura: {nombre:<10}'   # Igual que lo anterior
'Formato con anchura: Pepe      '
>>> f'Formato con anchura: {nombre:>10}'   # Alinea a la derecha
'Formato con anchura:       Pepe'
>>> f'Formato con anchura: {nombre:^10}'   # Alinea al centro
'Formato con anchura:    Pepe   '
```

Ejemplos de cadenas formateadas con números positivos:

```
>>> x, y = 400, 300
>>> f'La suma de {x} y {y} es {x + y}' # Se puede usar cualquier expresión
'La suma de 400 y 300 es 700'
>>> f'Formato con anchura: {x:10}'     # Los números se alinean a la derecha
'Formato con anchura:        400'
>>> f'Formato con anchura: {x:>10}'    # Igual que lo anterior
'Formato con anchura:        400'
>>> f'Formato con anchura: {x:2}'      # Ancho demasiado pequeño, se ignora
'Formato con anchura: 400'
>>> f'Formato con anchura: {x:<10}'    # Alinea a la izquierda
'Formato con anchura: 400        '
>>> f'Formato con anchura: {x:>10}'    # Alinea a la derecha
'Formato con anchura:        400'
>>> f'Formato con anchura: {x:^10}'    # Alinea al centro
'Formato con anchura:    400     '
>>> f'Formato con anchura: {x:=10}'    # En positivos no hay diferencia con >
'Formato con anchura:        400'
>>> f'Formato con anchura: {x:@<10}'   # Alinea a la izquierda, rellena con @
'Formato con anchura: 400@@@@@@@'
>>> f'Formato con anchura: {x:@>10}'   # Alinea a la derecha, rellena con @
'Formato con anchura: @@@@@@@400'
>>> f'Formato con anchura: {x:@^10}'   # Alinea al centro, rellena con @
'Formato con anchura: @@@400@@@@'
>>> f'Formato con anchura: {x:@=10}'   # En positivos no hay diferencia con >
'Formato con anchura: @@@@@@@400'
```

Ejemplos de cadenas formateadas con números negativos:

```
>>> z = -400
>>> f'Formato con anchura: {z:10}'   # A la derecha, signo junto al nº
'Formato con anchura:       -400'
>>> f'Formato con anchura: {z:<10}'  # A la izquierda, signo junto al nº
'Formato con anchura: -400       '
>>> f'Formato con anchura: {z:>10}'  # A la derecha, signo junto al nº
'Formato con anchura:       -400'
>>> f'Formato con anchura: {z:^10}'  # Al centro, signo junto al nº
'Formato con anchura:    -400    '
>>> f'Formato con anchura: {z:=10}'  # A la derecha, signo junto al relleno
'Formato con anchura: -      400'
>>> f'Formato con anchura: {z:010}'  # A la derecha, signo junto al relleno
'Formato con anchura: -000000400'
>>> f'Formato con anchura: {z:0<10}' # A la izquierda, signo junto al nº
'Formato con anchura: -400000000'
>>> f'Formato con anchura: {z:0>10}' # A la derecha, signo junto al nº
'Formato con anchura: 000000-400'
>>> f'Formato con anchura: {z:0^10}' # Al centro, signo junto al nº
'Formato con anchura: 000-400000'
>>> f'Formato con anchura: {z:0=10}' # A la derecha, signo junto al relleno
'Formato con anchura: -000000400'
>>> f'Formato con anchura: {z:@<10}' # A la izquierda, signo junto al nº
'Formato con anchura: -400@@@@@@'
>>> f'Formato con anchura: {z:@>10}' # A la derecha, signo junto al nº
'Formato con anchura: @@@@@@-400'
>>> f'Formato con anchura: {z:@^10}' # Al centro, signo junto al nº
'Formato con anchura: @@@-400@@@'
>>> f'Formato con anchura: {z:@=10}' # A la derecha, signo junto al relleno
```

```
'Formato con anchura: -@@@@@@400'
```

Ejemplos de cadenas formateadas con números en coma flotante:

```
>>> from math import pi
>>> f'El valor de pi es {pi:6.3}'     # Ancho 6, precisión 3
'El valor de pi es   3.14'
>>> f'El valor de pi es {pi:10.3}'    # Ancho 10, precisión 3
'El valor de pi es       3.14'
>>> f'El valor de pi es {pi:<10.3}'   # A la izquierda
'El valor de pi es 3.14      '
>>> f'El valor de pi es {pi:>10.3}'   # A la derecha
'El valor de pi es       3.14'
>>> f'El valor de pi es {pi:^10.3}'   # Al centro
'El valor de pi es    3.14   '
>>> f'El valor de pi es {pi:=10.3}'   # A la derecha
'El valor de pi es       3.14'
>>> f'El valor de pi es {pi:10.3f}'   # 3 dígitos en la parte fraccionaria
'El valor de pi es      3.142'
>>> f'El valor de pi es {pi:<10.3f}'  # A la izquierda
'El valor de pi es 3.142     '
>>> f'El valor de pi es {pi:>10.3f}'  # A la derecha
'El valor de pi es      3.142'
>>> f'El valor de pi es {pi:^10.3f}'  # Al centro
'El valor de pi es   3.142   '
>>> f'El valor de pi es {pi:=10.3f}'  # A la derecha
'El valor de pi es      3.142'
>>> f'El valor de pi es {-pi:=10.3f}' # Los negativos, igual que los enteros
'El valor de pi es -    3.142'
```

Más ejemplos:

```
>>> nombre = "Fred"
>>> f"Dice que su nombre es {nombre!r}."
"Dice que su nombre es 'Fred'."
>>> f"Dice que su nombre es {repr(nombre)}."  # repr es equivalente a !r
"Dice que su nombre es 'Fred'."
>>> ancho = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{ancho}.{precision}}"  # campos anidados
'result:      12.35'
>>> import datetime
>>> hoy = datetime.datetime(year=2017, month=1, day=27)
>>> f"{hoy:%B %d, %Y}"  # usando especificador de formato de fecha
'January 27, 2017'
>>> numero = 1024
>>> f"{numero:#0x}"  # usando especificador de formato de enteros
'0x400'
```

2.1.2. Expresiones regulares

Las **expresiones regulares** (también llamados *regex*) constituyen un pequeño lenguaje muy especializado incrustado dentro de Python y disponible a través del módulo `re`.

Usando este pequeño lenguaje es posible especificar **reglas sintácticas** de una forma distinta pero parecida a las *gramáticas EBNF* (aunque con menos poder expresivo).

Esas reglas sintácticas se pueden usar luego para **comprobar si una cadena *se ajusta* a un patrón**.

Este patrón puede ser frases en español, o direcciones de correo electrónico o cualquier otra cosa.

A continuación, se pueden hacer preguntas del tipo: «*¿Esta cadena se ajusta al patrón?*» o «*¿Hay algo que se ajuste al patrón en alguna parte de esta cadena?*».

También se pueden usar las *regexes* para **modificar** una cadena o **dividirla en partes** según el patrón indicado.

El lenguaje de las expresiones regulares es relativamente pequeño y restringido, por lo que no es posible usarlo para realizar cualquier tipo de procesamiento de cadenas.

Además, hay procesamientos que se pueden realizar con *regexes* pero las expresiones que resultan se vuelven muy complicadas.

En estos casos, es mejor escribir directamente código Python ya que, aunque el código resultante pueda resultar más lento, probablemente resulte más fácil de leer.

Para más información sobre cómo crear y usar expresiones regulares, consultar:

- Tutorial de introducción en https://docs.python.org/3/howto/regex.html
- Documentación del módulo `re` en https://docs.python.org/3/library/re.html

2.2. Tuplas

Las **tuplas** (`tuple`) son secuencias inmutables, usadas frecuentemente para representar colecciones de datos heterogéneos (es decir, de tipos distintos).

También se usan en aquellos casos en los que se necesita una secuencia inmutable de datos homogéneos (por ejemplo, para almacenar datos en un conjunto o un diccionario).

Las tuplas se pueden crear así:

- Si es una tupla vacía, con paréntesis vacíos: `()`
- Si sólo tiene un elemento, se pone una coma detrás:

  ```
  a,
  (a,)
  ```

- Si tiene más de un elemento, se separan con comas:

  ```
  a, b, c
  (a, b, c)
  ```

- Usando la función `tuple(`⟨*iterable*⟩`)`.

Observar que lo que construye la tupla es realmente la coma, no los paréntesis.

Los paréntesis son opcionales, excepto en dos casos:

- La tupla vacía: `()`
- Cuando son necesarios para evitar ambigüedad.

  Por ejemplo, `f(a, b, c)` es una llamada a una función con tres argumentos, mientras que `f((a, b, c))` es una llamada a una función con un único argumento que es una tupla de tres elementos.

Las tuplas implementan todas las operaciones comunes de las secuencias.

En general, las tuplas se pueden considerar como la versión inmutable de las listas.

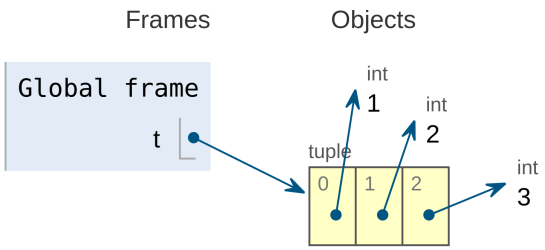
Además, las tuplas son *hashables* si sus elementos también lo son.

En memoria, las tuplas se almacenan mediante una estructura de datos donde sus elementos se identifican mediante un índice, que es un número entero que indica la posición que ocupa el elemento dentro de la tupla.

Por ejemplo, la siguiente tupla:

```
t = (1, 2, 3)
```

se almacenaría de la siguiente forma según lo representa la herramienta Pythontutor:



Tupla almacenada en memoria

2.3. Rangos

Los **rangos** (`range`) representan secuencias inmutables y *hashables* de números enteros y se usan frecuentemente para hacer bucles que se repitan un determinado número de veces.

Los rangos se crean con la función `range`:

```
range([start: int,] stop: int [, step: int]) -> range
```

Cuando se omite *start*, se entiende que es `0`.

Cuando se omite *step*, se entiende que es `1`.

El valor de *stop* no se alcanza nunca.

Cuando *start* y *stop* son iguales, representa el *rango vacío*.

step debe ser siempre distinto de cero.

Cuando *start* es mayor que *stop*, el valor de *step* debería ser negativo. En caso contrario, también representaría el rango vacío.

El **contenido** de un rango *r* vendrá determinado por la fórmula:

$$r[i] = start + step \cdot i$$

donde $i \geq 0$. Además:

- Si $step > 0$, se impone también la restricción $r[i] < stop$.
- Si $step < 0$, se impone también la restricción $r[i] > stop$.

Un rango es **vacío** cuando $r[0]$ no satisface las restricciones anteriores.

Los rangos admiten **índices negativos**, pero se interpretan como si se indexara desde el final de la secuencia usando índices positivos.

Los rangos implementan **todas las operaciones de las secuencias, *excepto* la concatenación y la repetición**.

Esto es debido a que los rangos sólo pueden representar secuencias que siguen un patrón muy estricto, y las repeticiones y las concatenaciones a menudo violan ese patrón.

Los rangos son perezosos y además ocupan mucha menos memoria que las listas o las tuplas (sólo hay que almacenar *start*, *stop* y *step*).

La **forma normal** de un rango es una expresión en la que se llama a la función `range` con los argumentos necesarios para construir el rango:

```
>>> range(10)
range(0, 10)
>>> range(0, 10)
range(0, 10)
>>> range(0, 10, 1)
range(0, 10)
```

```
>>> range(0, 30, 5)
range(0, 30, 5)
>>> range(0, -10, -1)
range(0, -10, -1)
>>> range(4, -5, -2)
range(4, -5, -2)
```

Para ver con claridad todos los elementos de un rango, podemos convertirlo en una tupla o una lista. Por ejemplo:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
```

```
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Dos rangos son considerados **iguales** si representan la misma secuencia de valores, sin importar si tienen distintos valores de *start*, *stop* o *step*.

Por ejemplo:

```
>>> range(20) == range(0, 20)
True
>>> range(0, 20) == range(0, 20, 2)
False
>>> range(0, 3, 2) == range(0, 4, 2)
True
>>> range(0) == range(2, 1, 3)
True
```

El **rango vacío** es un valor que no tiene expresión canónica, ya que cualquiera de las siguientes expresiones representan al rango vacío tan bien como cualquier otra:

- `range(0)`.
- `range(a, a)`, donde a es cualquier entero.
- `range(a, b, c)`, donde $a \geq b$ y $c > 0$.
- `range(a, b, c)`, donde $a \leq b$ y $c < 0$.

```
>>> range(3, 3) == range(4, 4)
True
>>> range(4, 3) == range(3, 4, -1)
True
```

3. Mutables

3.1. Listas

Las **listas** son secuencias *mutables*, usadas frecuentemente para representar colecciones de elementos heterogéneos.

Al ser mutables, las listas **no** son *hashables*.

Se pueden construir de varias maneras:

- Usando corchetes vacíos para representar la lista vacía: `[]`.
- Usando corchetes y separando los elementos con comas:

  `[a]`

  `[a, b, c]`

- Usando la función `list` con las sintaxis `list()` o `list(`⟨*iterable*⟩`)`.

La función `list` construye una lista cuyos elementos son los mismos (y están en el mismo orden) que los elementos de ⟨*iterable*⟩.

⟨*iterable*⟩ puede ser:

- una secuencia,
- un contenedor sobre el que se pueda iterar, o
- un iterador.

Si se llama sin argumentos, devuelve una lista vacía.

Por ejemplo:

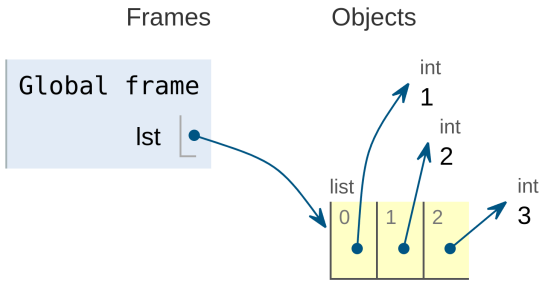
```
>>> list('hola')
['h', 'o', 'l', 'a']
>>> list((1, 2, 3))
[1, 2, 3]
```

En memoria, las listas se almacenan mediante una estructura de datos donde sus elementos se identifican mediante un índice, que es un número entero que indica la posición que ocupa el elemento dentro de la lista.

Por ejemplo, la siguiente lista:

```
lst = [1, 2, 3]
```

se almacenaría de la siguiente forma según lo representa la herramienta Pythontutor:



Lista almacenada en memoria

3.1.1. Listas por comprensión

También se pueden crear **listas por comprensión** usando la misma sintaxis de las **expresiones generadoras** pero encerrando la expresión entre corchetes en lugar de entre paréntesis.

Su sintaxis es:

⟨*lista_comp*⟩ ::= **[**⟨*expresión*⟩ (**for** ⟨*identificador*⟩ **in** ⟨*secuencia*⟩ [**if** ⟨*condición*⟩])$^+$**]**

Por ejemplo:

```
>>> [x ** 2 for x in [1, 2, 3]]
[1, 4, 9]
```

Como se ve, el resultado es directamente una lista, no un iterador.

Por tanto, a diferencia de lo que pasa con las expresiones generadoras, **el resultado de una lista por comprensión no es perezoso**, cosa que habrá que tener en cuenta para evitar consumir más memoria de la necesaria o generar elementos que al final no sean necesarios.

Por ejemplo, la siguiente expresión generadora:

```
res_gen = (x ** 2 for x in range(0, 10000000000000))
```

es mucho más eficiente en tiempo y espacio que la lista por comprensión:

```
res_list = [x ** 2 for x in range(0, 10000000000000)]
```

ya que la expresión generadora devuelve un **iterador** que irá generando los valores de uno en uno a medida que los vayamos recorriendo con `next(res_gen)`.

En cambio, la lista por comprensión genera todos los valores de la lista a la vez y los almacena todos juntos en la memoria.

A cambio, la ventaja de tener una lista frente a tener un iterador es que podemos acceder directamente a cualquier elemento de la lista mediante la indexación.

Las listas por comprensión, al igual que las expresiones generadoras, **determinan su propio *ámbito***.

Ese ámbito abarca toda la lista por comprensión, de principio a fin.

Los identificadores que aparecen en la cláusula **for** se consideran *variables ligadas* en la lista por comprensión.

Esos identificadores se van ligando, uno a uno, a cada elemento de la secuencia indicada en la cláusula **in**.

Como son variables ligadas, cumplen estas dos propiedades:

1. Se pueden renombrar (siempre de forma consistente) sin que la lista por comprensión cambie su significado.

   Por ejemplo, las dos listas por comprensión siguientes son equivalentes, puesto que producen el mismo resultado:

```
[x for x in (1, 2, 3)]
```

```
[y for y in (1, 2, 3)]
```

2. No se pueden usar fuera de la lista por comprensión, ya que estarían fuera de su ámbito y no serían visibles.

   Por ejemplo, lo siguiente daría un error de nombre:

```
>>> e = [x for x in (1, 2, 3)]
>>> x       # Intento acceder a la 'x' de la lista por comprensión
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

3.2. Operaciones mutadoras

En la siguiente tabla, *s* es una instancia de un tipo de secuencia mutable (por ejemplo, una lista), *t* es cualquier dato iterable y *x* es un dato cualquiera que cumple con las restricciones que impone *s*:

Operación	Resultado
`s[i] = x`	El elemento *i*-ésimo de *s* se sustituye por *x*
`s[i:j] = t`	La rodaja de *s* desde *i* hasta *j* se sustituye por *t*
`s[i:j:k] = t`	Los elementos de `s[i:j:k]` se sustituyen por *t*
`del s[i]`	Elimina el elemento *i*-ésimo de *s*
`del s[i:j]`	Elimina los elementos de `s[i:j]` Equivale a hacer `s[i:j] = []`
`del s[i:j:k]`	Elimina los elementos de `s[i:j:k]`

Operación	Resultado
`s.append(x)`	Añade *x* al final de *s*; es igual que `s[len(s):len(s)] = [x]`
`s.clear()`	Elimina todos los elementos de *s*; es igual que `del s[:]`
`s.copy()`	Crea una copia *superficial* de *s*; es igual que `s[:]`
`s.extend(t)` `s += t`	Extiende *s* con el contenido de *t*; es igual que `s[len(s):len(s)] = t`
`s *= n`	Modifica *s* repitiendo su contenido *n* veces
`s.insert(i, x)`	Inserta *x* en *s* en el índice *i*; es igual que `s[i:i] = [x]`

Operación	Resultado
s`.pop(`[*i*]`)`	Extrae el elemento *i* de *s* y lo devuelve (por defecto, *i* vale `-1`)
s`.remove(`*x*`)`	Quita el primer elemento de *s* que sea igual a *x*
s`.reverse()`	Invierte los elementos de *s*
s`.sort()`	Ordena los elementos de *s*

La **copia superficial** (a diferencia de la ***copia profunda***) significa que sólo se copia el objeto sobre el que se aplica la copia, **no sus elementos**.

Por tanto, al crear la copia superficial, se crea sólo un nuevo objeto, donde se copiarán las referencias de los elementos del objeto original.

Esto influye, sobre todo, cuando los elementos de una colección mutable también son objetos mutables.

Por ejemplo, si tenemos listas dentro de otra lista, y copiamos ésta última con `.copy()`, la nueva lista compartirá elementos con la lista original:

```
>>> x = [[1, 2], [3, 4]]   # los elementos de «x» también son listas
>>> y = x.copy()           # «y» es una copia de «x»
>>> x is y
False                      # no son la misma lista
>>> x[0] is y[0]
True                       # sus elementos no se han copiado,
>>> x[0].append(9)         # sino que están compartidos por «x» e «y»
>>> x
[[1, 2, 99], [3, 4]]
>>> y
[[1, 2, 99], [3, 4]]
```

El método `sort` permite ordenar los elementos de la secuencia de forma ascendente o descendente:

```
>>> x = [3, 6, 2, 9, 1, 4]
>>> x.sort()
>>> x
[1, 2, 3, 4, 6, 9]
>>> x.sort(reverse=True)
>>> x
[9, 6, 4, 3, 2, 1]
```

Bibliografía

Python Software Foundation. n.d. “Sitio Web de Documentación de Python.” https://docs.python.org/3.