LISTAS

AMPLIACIÓN DE FUNCIONALIDAD

Algoritmos y estructuras de datos I

COPIAR EL CONTENIDO DE UNA LISTA

```
lista1 = [1, 3, 5, 8]
lista2 = lista1.copy()

print(id(lista1), id(lista2))
#los identificadores son distintos
```

Podemos encontrar un escenario en el que queramos hacer una copia de la lista. La forma más directa de realizar esta acción es a través del método *copy()*.

Como hemos visto, el signo de igualdad = se puede utilizar para construir un duplicado de una lista. Sin embargo, la nueva lista estará vinculada a la existente. Esto significa que la nueva lista también se verá alterada si se actualiza la lista original.

Entonces, una copia de los elementos existentes en una lista se puede lograr utilizando el método *copy()*, que no toma ningún parámetro en su llamado.

ORDENAR UNA LISTA

Las listas son secuencias ordenadas. Esto quiere decir que sus elementos siempre se devuelven en el mismo orden en que fueron añadidos. Sin embargo, es posible ordenar los elementos de una lista con el método *sort()*, que ordena los elementos de la lista y modifica la lista actual (no se obtiene una nueva lista).

Si se desea obtener los resultados ordenados de forma descendente, se debe especificar el parámetro *reverse=True* en el llamado.

```
numeros = [3, 11, 6, 1, 7, 8]

numeros.sort() #Orden ascendente
print(numeros)

numeros.sort(reverse=True) #Orden descendente
print(numeros)
```

Lic. Yaps David P.

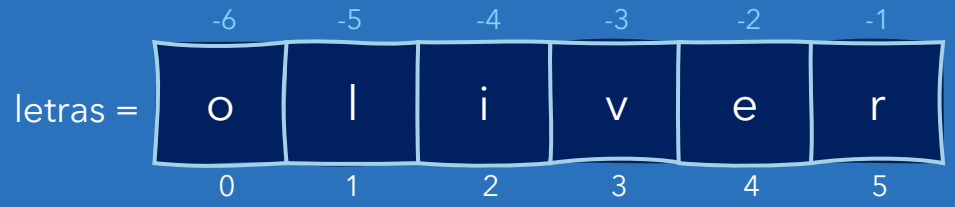
SEGMENTACIÓN DE LISTAS (REBANADO)

Para acceder a un rango de elementos en una lista, es necesario dividir la misma. Una forma para lograr éste objetivo, es utilizar el operador de corte simple, es decir, dos puntos (:). Con este operador, se puede especificar dónde comenzar el corte y dónde terminar:

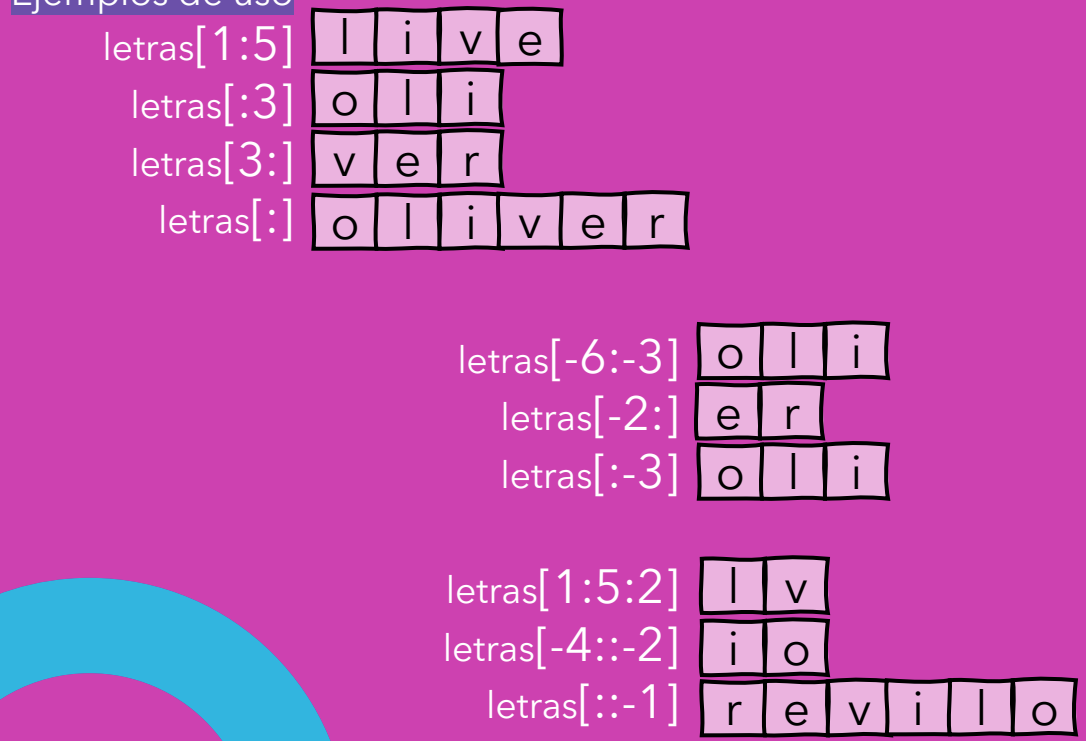
(opcional)

lista [inicio : fin : *paso*]

La segmentación de listas devuelve una lista nueva a partir de la lista existente sin alterar la original:

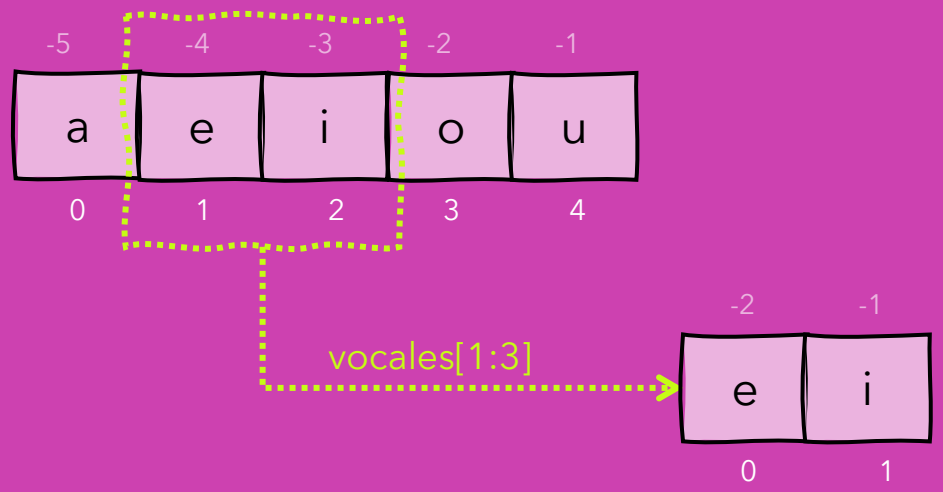


Ejemplos de uso



SEGMENTACIÓN DE LISTAS (REBANADO)

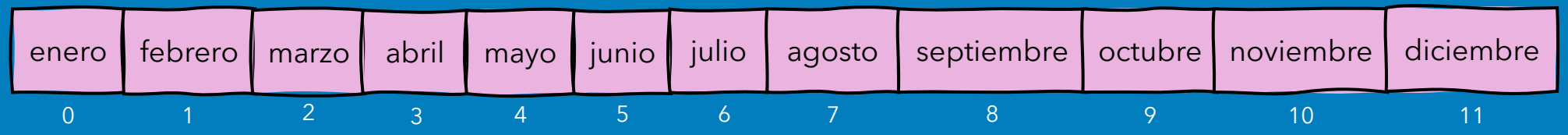
A continuación, se grafican algunos ejemplos del uso del operador, siguiendo las alternativas básicas de su aplicación.




```
vocales = ['a', 'e', 'i', 'o', 'u']

#Utilizando índices positivos:
print(vocales[3:4]) #Desde posiciones específicas
print(vocales[:2]) #Si no se especifica inicio se toma como 0
print(vocales[3:]) #A partir de una posición hasta el final
print(vocales[:]) #Toda la lista completa

#Utilizando índices negativos:
print(vocales[-5:-1]) #Desde la posición -5 hasta -1 (no incluido)
print(vocales[-2:]) #Desde la posición -2 hasta el final
print(vocales[:-3]) #Desde el principio hasta -3 (no incluido)
```



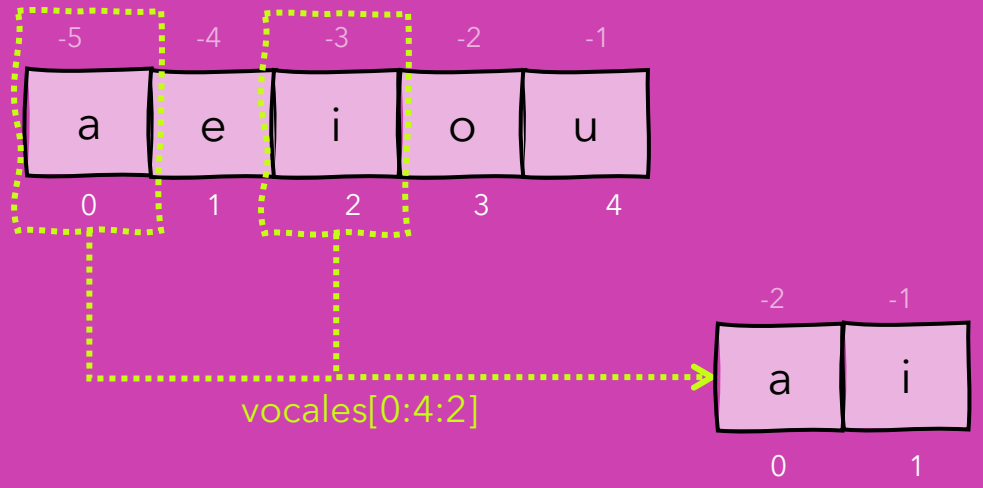
SEGMENTACIÓN DE LISTAS (REBANADO)

En el código presentado, se aplican estas técnicas al conjunto de datos de los meses del año. Dividiendo los meses en trimestres y semestres, se ilustra cómo el operador de rebanado puede ser utilizado para acceder a subconjuntos de datos temporales de manera eficiente y precisa.

```
mesesDelAño = ["enero", "febrero", "marzo", "abril",
"mayo", "junio", "julio", "agosto", "septiembre",
"octubre", "noviembre", "diciembre"]

#Trimestres
primerTrimestre = mesesDelAño[:3] #Enero a marzo
segundoTrimestre = mesesDelAño[3:6] #Abril a junio
tercerTrimestre = mesesDelAño[6:9] #Julio a septiembre
cuartoTrimestre = mesesDelAño[9:] #Octubre a diciembre

# Semestres
primerSemestre = mesesDelAño[:6] #Enero a junio
segundoSemestre = mesesDelAño[6:] #Julio a diciembre
```



SEGMENTACIÓN DE LISTAS CON PASOS

El operador de corte soporta un tercer «argumento» y es el número de pasos que se avanza al cortar:

```
print(vocales[0:4:1]) #Es equivalente a vocales[0:4]
```

Por defecto, es 1; pero se pueden cambiar a otros valores numéricos, por ejemplo, si queremos que sea en pasos de 2:

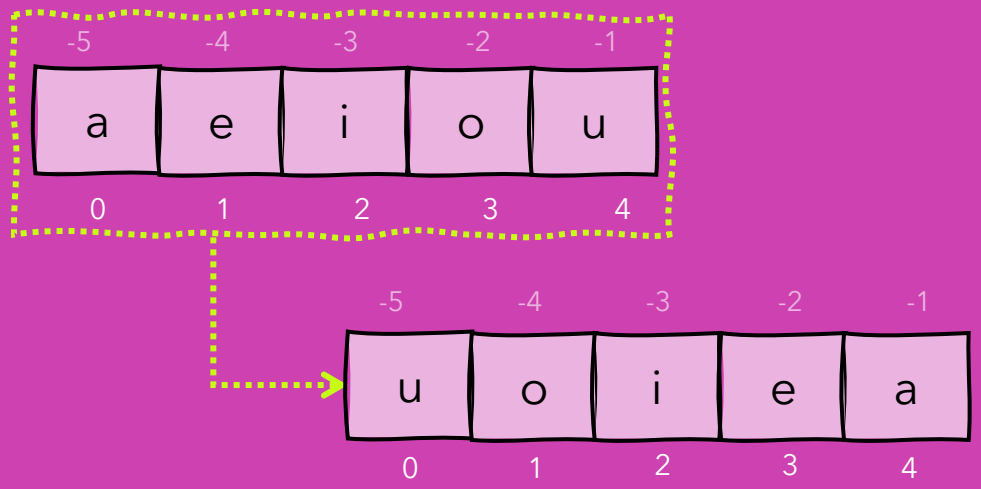
```
print(vocales[0:4:2]) #Avanza de a dos pasos
```

Entonces obtiene los elementos del 0 hasta el 4, pero ignorando algunos. Comienza en el 0, y avanza 2 pasos, así que selecciona el que tiene el índice 2. Luego, desde el 2 avanza otros 2 pasos y si ya no hay más elementos en el rango para saltar, entonces se detiene.

SEGMENTACIÓN DE LISTAS CON PASOS: INVERTIR CONTENIDO

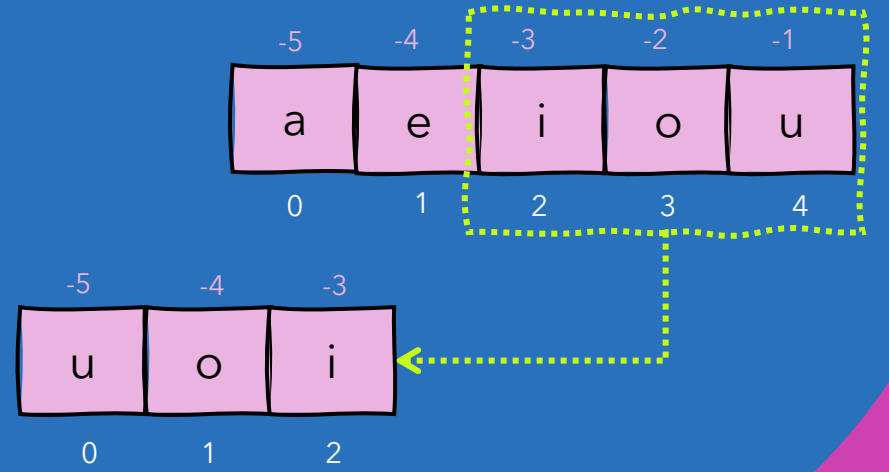
Si quisiéramos invertir el resultado de una lista, podríamos avanzar con paso -1, seleccionado el inicio hasta el fin:

```
print(vocales[::-1]) #Invierte la lista
```



También podríamos realizarlo utilizando índices específicos, pero el orden debe ser inverso:

```
print(vocales[4:1:-1]) #Invierte la lista
```



```
lista = [3, 6, 9, 12, 15]

#Aplicamos el método reverse()
lista.reverse()

#Imprimimos la lista invertida
print(lista) #[15, 12, 9, 6, 3]

lista = [3, 6, 9, 12, 15]

#Creamos una nueva lista invertida
listaInvertida = list(reversed(lista))

#Imprimimos la lista invertida
print(listaInvertida) #[15, 12, 9, 6, 3]

#La lista original no se ha modificado
print(lista) #[3, 6, 9, 12, 15]
```

MÉTODOS REVERSE Y REVERSED

Se utiliza para invertir el orden de los elementos de una lista de forma permanente. A diferencia de la técnica de rebanado que muestran la lista en orden inverso sin modificar la lista original, *reverse()* modifica la lista actual.

Se debe observar que después de llamar a *reverse()*, la lista original se ha modificado permanentemente.

Si es necesario invertir una lista sin modificar la lista original, es posible utilizar la función *reversed()* junto con *list()* para obtener una nueva lista invertida.

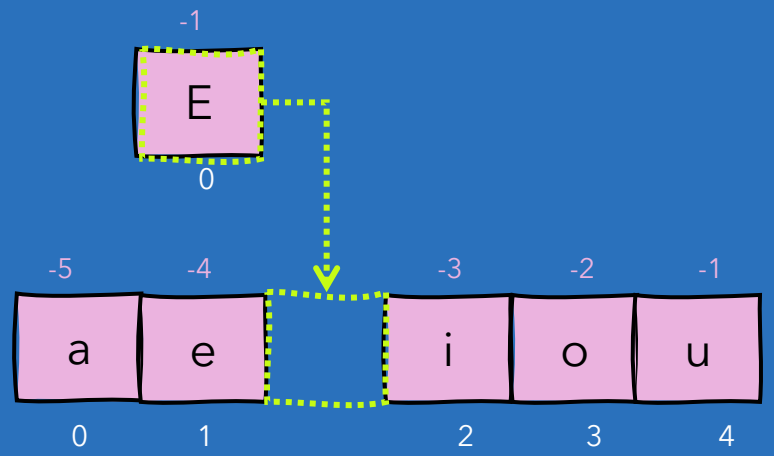
SEGMENTACIÓN DE LISTAS CON LONGITUD 0

Una rebanada de longitud cero es aquella que devuelve una lista sin elementos. Por ejemplo:

```
print(vocales[2:2]) #Retorna una lista vacía
```

El ejemplo anterior retorna todos los elementos entre el índice 2 y, sin incluir, el índice 2 también, es decir, todos los elementos comprendidos entre el índice 2 y el 1. Sin embargo, podríamos hacer una inserción de un elemento utilizando el operador de asignación (del lado derecho de la asignación siempre debe haber una lista):

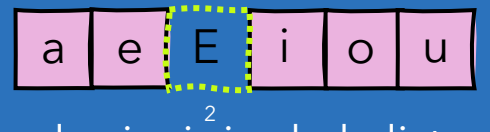
```
vocales[2:2] = ['E'] #Se inserta un elemento a la lista en esa posición
```



MÉTODO INSERT

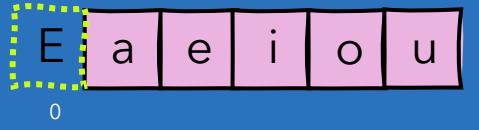
La función insert() agrega un elemento en el índice dado de una lista existente. Acepta dos parámetros, el índice a insertar y el valor a insertar:

```
vocales.insert(2, 'E') #Agrega el valor E en la posición 2
```



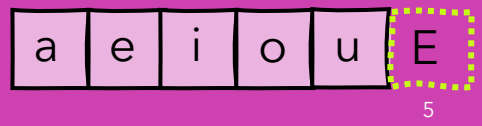
Para agregar un elemento al principio de la lista usando esta función, debemos establecer el primer argumento como 0, que denota que la inserción se realiza en el índice 0, el comienzo de la lista:

```
vocales.insert(0, 'E') #Agrega el valor E en la posición inicial
```



Si quisiéramos agregar el elemento al final, utilizaríamos la función len en la posición:

```
vocales.insert(len(vocales), 'E') #Agrega el valor E en la posición final
```



SEGMENTACIÓN DE LISTAS CON LONGITUD 0

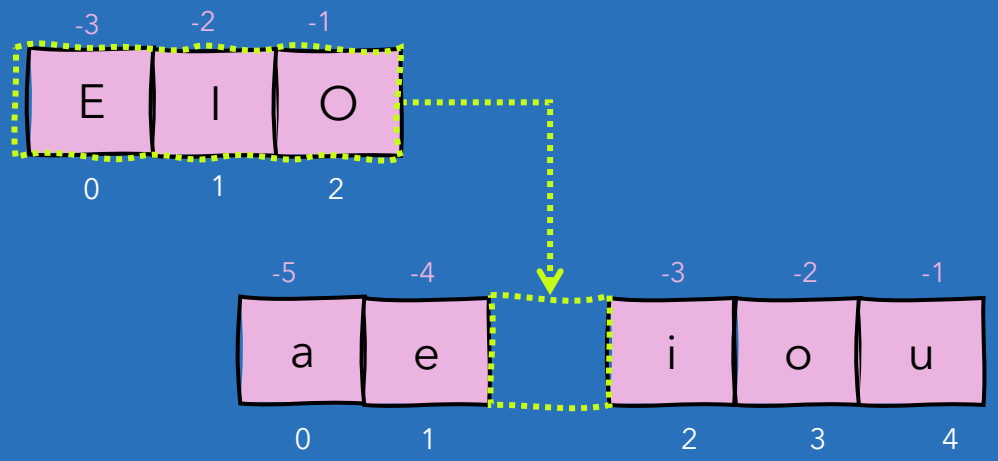
Sin embargo, la técnica de la segmentación permite, a diferencia del método *insert*, insertar más de un elemento en la misma operación.

El objeto *list* no dispone de ningún método que sea capaz de hacer esto:

```
vocales[2:2] = ['E','I','O'] #Se insertan los elementos a la lista en esa posición
```

Podemos situar también el cursor de inserción justo después del último elemento, lo que provocará que la lista se extienda por la derecha:

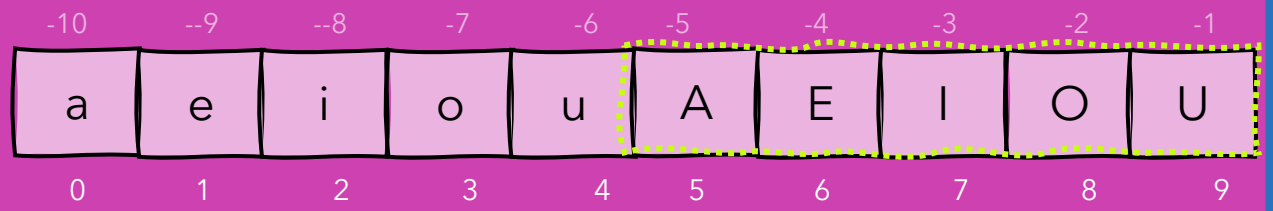
```
vocales[len(vocales):] = ['E','I','O'] #Se insertan los elementos al final
```



MÉTODO EXTEND

Nos permite añadir elementos de una lista a una lista existente, iterando sobre el argumento y agregando cada elemento a la lista. Por tanto, el argumento dado debe ser de tipo iterable (otra lista por ejemplo):

```
vocales.extend(['A', 'E', 'I', 'O', 'U']) #Agrega los elementos de la lista al final
```



Deberíamos usar este método si necesitamos agregar varios elementos a una lista como elementos individuales a partir de una secuencia.

Como se puede ver en el ejemplo, se agregan los elementos de un iterable al final de una lista en el orden en el que aparecen.

Devuelve la longitud de la lista
fibonacci.count(1) 2

Devuelve la longitud de la lista
len(fibonacci) 5

Retorna la suma de todos los elementos en la lista
sum(fibonacci) 7

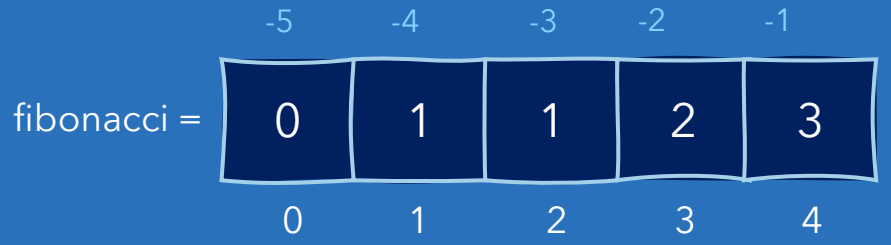
Devuelve el elemento mínimo en la lista
min(fibonacci) 0

Devuelve el elemento máximo en la lista
max(fibonacci) 3

AGREGACIÓN

Nos sumergiremos en operaciones clave de agregación que enriquecen la utilidad de las listas: *len*, *sum*, *min*, *max* y *count*.

Estas funciones no solo proporcionan información vital sobre la composición de las listas, sino que también ofrecen una manera eficiente de extraer estadísticas y resúmenes esenciales.



OPERADOR IN

```
lista = [1, 3, 5, 8, 13]

if (8 in lista):
    print("El valor se encuentra en la lista")
else:
    print("El valor no se encuentra en la lista")

if (7 not in lista):
    lista.append(7) #Agrega el elemento si no existe
```

Un operador de pertenencia se emplea para identificar la existencia de un valor dentro de una lista. Retorna *True* en caso de que exista; y *False* en caso contrario.

También podemos utilizar su negación anteponiendo el operador *not*, invirtiendo su resultado de verdad.

OPERADOR IN

Otra forma de utilizar el operador in es el recorrido de elementos de una lista utilizando la estructura de bucle *for*, sin la necesidad de acceder por sus índices; donde cada uno de los elementos de la lista se deposita en la variable a la izquierda del *in* en cada iteración.

Esta forma de recorrer sería equivalente a la que se utiliza el acceso a los índices de la lista al realizar la iteración.

```
lista = [1, 3, 5, 8, 13]

for numero in lista:
    print(numero)

for i in range(0, len(lista)):
    print(lista[i])
```

OPERADOR IN: RECORRIDO DE MATRICES

El bucle *for* junto con el operador *in* se puede utilizar para recorrer cada elemento de una matriz. Esto se logra anidando bucles *for* y utilizando la estructura de la matriz.

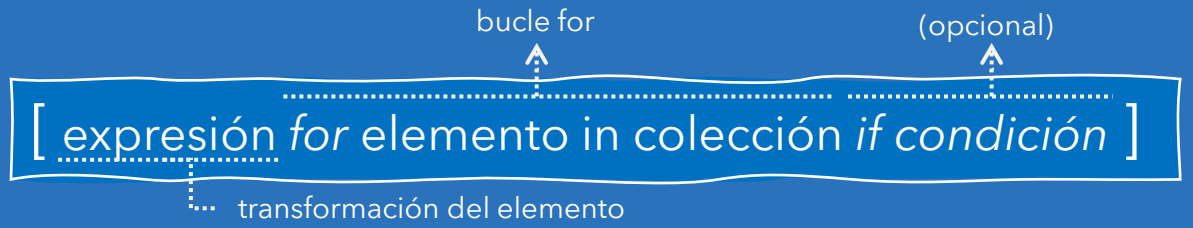
Para recorrer fila por fila, se itera sobre cada fila de la matriz utilizando un bucle *for*, mientras que para recorrer columna por columna, se itera sobre cada elemento en una fila de la matriz en cada iteración.

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

for fila in matriz:
    print('|', end='')
    for columna in fila:
        print(columna, end='|')
    print()
```

COMPRENSIÓN DE LISTAS

La comprensión (*Comprehension*) es una construcción sintáctica para crear listas a partir de los elementos de otros elementos iterables (como otras listas) de una forma rápida de escribir, muy legible y funcionalmente eficiente:



A menudo la expresión (aquello que terminará insertado en la lista resultante) es igual al elemento que se recorre y la condición es opcional. Entre corchetes se escribe una expresión seguida de un bucle *for* sobre el que se itera, para finalmente escribir una condición de ser necesaria.

Lic. Yaps David P.

COMPRENSIÓN DE LISTAS

Un ejemplo trivial para comenzar a trabajar con listas por comprensión es la operación de inicialización de una lista con una secuencia de valores. Regularmente, a tal fin, tendríamos un código similar al siguiente:

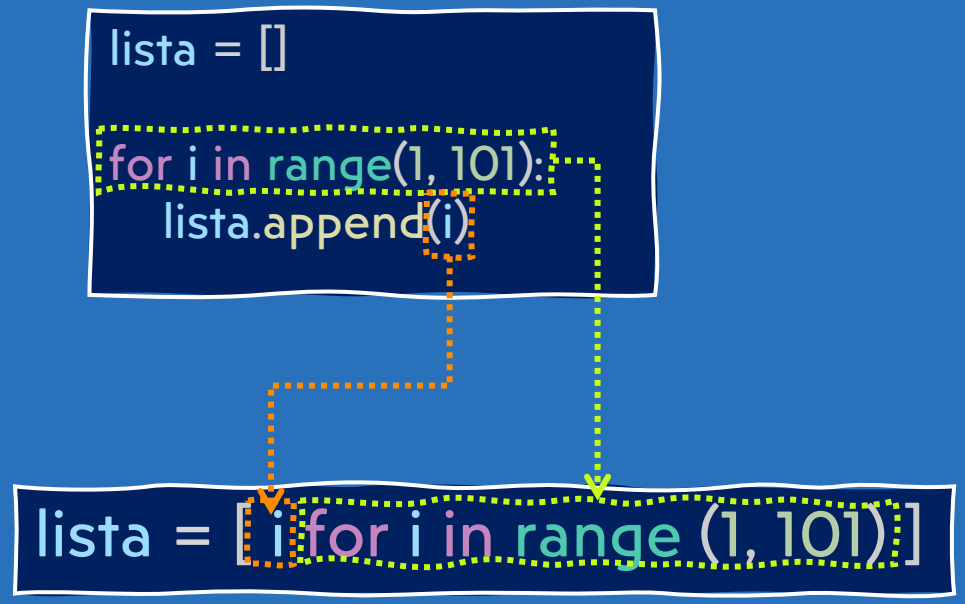
```
lista = []

for i in range(1, 101):
    lista.append(i)
```

Esta operación la podríamos resumir en una sola línea utilizando la comprensión:

```
lista = [ i for i in range (1, 101) ]
```

Obteniéndose el mismo resultado solamente con menos código.



COMPRENSIÓN DE LISTAS

```
lista = [1, 2, 3, 4, 5]
listaCuadrados = []

for elemento in lista:
    listaCuadrados.append(elemento ** 2)
```

```
listaCuadrados = [ elemento ** 2 for elemento in lista ]
```

Pasemos a otro ejemplo en el que se recorre una lista existente y, mediante la misma, se desea crear una nueva, pero con los valores elevados al cuadrado. Podríamos escribir un código similar al siguiente.

COMPRENSIÓN DE LISTAS

Continuando con el ejemplo anterior, supongamos que solamente queremos crear la nueva lista con los elementos pares de la lista original.

En esta situación, también podemos escribirlo por comprensión, utilizando la condición.

```
lista = [1, 2, 3, 4, 5]
listaCuadrados = []

for elemento in lista:
    if (elemento % 2 == 0):
        listaCuadrados.append(elemento ** 2)
```

```
listaCuadrados = [elemento ** 2 for elemento in lista if elemento % 2 == 0]
```

Lic. Yaps David P.

AUTOEVALUACIÓN

La autoevaluación es crucial para identificar mejoras y fortalezas de forma objetiva, facilitando la fijación de metas y la toma de acciones para alcanzarlas.