List Comprehensions en Python

Pero, ¿qué es esto?

Las **comprensiones de lista** son como varitas mágicas en Python, permitiéndonos crear y transformar listas en un abrir y cerrar de ojos. Imagina tomar una lista, darle un agite y, ¡voilà!, aparece una lista completamente nueva, lista para deslumbrarte con su contenido.

Estas concisas líneas de código son una herramienta poderosa para construir listas basadas en listas existentes, todo mientras mantienes tu código ordenado y conciso. No más bucles largos y líneas repetitivas: las comprensiones de lista hacen el trabajo pesado por ti.

Así que, arremanguémonos y sumerjámonos en el mundo encantador de las comprensiones de lista. ¡Prepárate para desbloquear nuevos niveles de elegancia y eficiencia en la codificación!

Lo que sabemos ahora:

Las **List Comprehensions** son una alternativa clara y compacta a los bucles `for` convencionales, permitiendo que construyamos nuevas listas aplicando expresiones a los elementos de una lista o cualquier otro iterable.

Sintaxis básica

La estructura general de una comprensión de lista es la siguiente:

```python [expresión for elemento in iterable]
```

```
# Definir una lista de palabras
list_of_words = ["Barcelona", "Madrid", "Girona", "Murcia"]

# Crear una lista vacía para almacenar las versiones en mayúsculas de las palabras
uppercase_cities = []

# Recorrer cada palabra en list_of_words
for word in list_of_words:
    # Convertir la palabra a mayúsculas usando el método upper() y añadirla a la nueva lista
    uppercase_cities.append(word.upper())

# Mostrar la lista de ciudades en mayúsculas
uppercase_cities
```

```
['BARCELONA', 'MADRID', 'GIRONA', 'MURCIA']
```

¿Y si lo podemos simplificar?

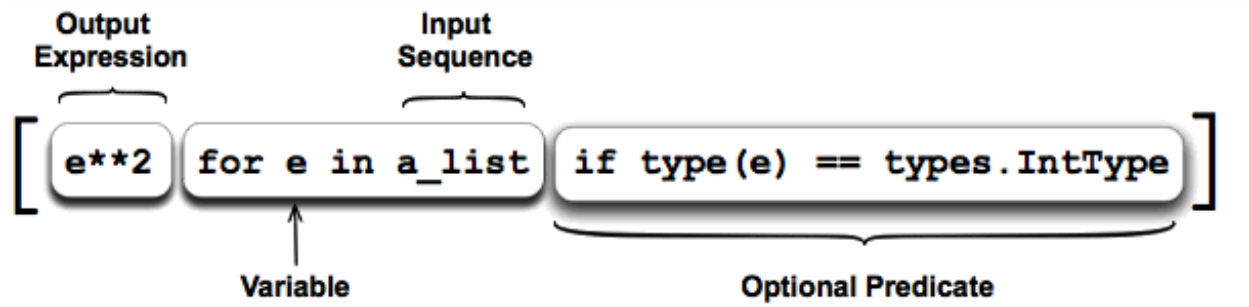
```
# Definir una lista de palabras
list_of_words = ["Barcelona", "Madrid", "Girona", "Murcia"]

# Crear una lista vacía para almacenar las versiones en mayúsculas de las palabras
uppercase_cities = [i.upper() for i in list_of_words]

# Mostrar la lista de ciudades en mayúsculas
uppercase_cities
```

```
['BARCELONA', 'MADRID', 'GIRONA', 'MURCIA']
```

Vale, pero ¿cuál es la magia detrás de esto?



Hagamos un ejemplo más **picante**:

```
# Definir una lista de palabras
list_of_words = ["Barcelona", "Madrid", "Girona", "Murcia"]

# Crear una lista vacía para almacenar palabras de más de 6 caracteres
new_list = []

# Recorrer cada palabra en list_of_words
for word in list_of_words:
    # Verificar si la longitud de la palabra es mayor que 6
    if len(word) > 6:
        # Si se cumple la condición, añadir la palabra a new_list
```

```
        new_list.append(word)

# Mostrar new_list que contiene palabras de más de 6 caracteres
new_list
```

```
['Barcelona']
```

de nuevo, podemos simplificarlo:

```
# Definir una lista de palabras
list_of_words = ["Barcelona", "Madrid", "Girona", "Murcia"]

# Alucinante:
long_cities = [i for i in list_of_words if len(i) > 6]

# Mostrar las ciudades largas
long_cities
```

```
['Barcelona']
```

Desafío fácil:

Queremos una lista con los cuadrados de los números del 1 al 10

```
# Usando un bucle for

squares_for_loop = []  # Lista vacía para almacenar los resultados
for x in range(1, 11):
    squares_for_loop.append(x**2)  # Elevamos cada número al cuadrado
y lo agregamos a la lista
print("Cuadrados usando bucle for:", squares_for_loop)
```

```
Cuadrados usando bucle for: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
# Usando una comprensión de lista

squares_comprehension = [x**2 for x in range(1, 11)]  # Comprensión de
lista que genera los cuadrados
print("Cuadrados usando comprensión de lista:", squares_comprehension)
```

```
Cuadrados usando comprensión de lista: [1, 4, 9, 16, 25, 36, 49, 64,
81, 100]
```

Desafío fácil:

Crear una nueva lista, sustituyendo las "e" por "a" en cada palabra de la lista original

```
list_of_words = ["Barcelona", "Madrid", "Gerona", "Murcia"]

# Usando un bucle for

new_list_for = []  # Lista vacía para almacenar las palabras
modificadas
for word in list_of_words:
    new_word = word.replace('e', 'a')  # Reemplazamos "e" por "a" en
cada palabra
    new_list_for.append(new_word)  # Agregamos la palabra modificada a
la nueva lista
print("Usando bucle for:", new_list_for)
```

```
Usando bucle for: ['Barcalona', 'Madrid', 'Garona', 'Murcia']
```

```
# Usando una comprensión de lista

new_list_comprehension = [word.replace('e', 'a') for word in
list_of_words]  # Comprensión de lista con reemplazo
print("Usando comprensión de lista:", new_list_comprehension)
```

```
Usando comprensión de lista: ['Barcalona', 'Madrid', 'Garona',
'Murcia']
```

Condiciones (ponemos IF en la comprensión)

If / Else en Comprensiones de Lista

Las **comprensiones de lista** en Python no solo son una forma compacta de crear listas, sino que también pueden incluir declaraciones condicionales con una estructura `if-else`. Esto te permite aplicar diferentes transformaciones o filtros a los elementos en función de condiciones específicas. En esta sección, exploraremos cómo las comprensiones de lista condicionales añaden flexibilidad y elegancia a tu código Python.

Lo que sabemos:

- Las comprensiones de lista pueden ser utilizadas con condiciones para modificar los elementos de una lista de forma más dinámica y compacta.

Ejemplo

Supongamos que tenemos una lista de nombres de ciudades y queremos modificar cada nombre según su longitud: si el nombre de la ciudad tiene más de 6 caracteres, lo convertiremos a mayúsculas; si tiene 6 caracteres o menos, lo convertiremos a minúsculas.

```
# Lista original de nombres de ciudades
list_of_words = ["Barcelona", "Madrid", "Girona", "Murcia"]
```

```python
# Usando un bucle for con if-else

new_list = []  # Inicializar una lista vacía para almacenar nombres de ciudades modificados

# Iterar a través de cada nombre de ciudad en la lista original
for city in list_of_words:
    # Verificar si la longitud del nombre de la ciudad es mayor a 6 caracteres
    if len(city) > 6:
        # Si es así, convertir el nombre de la ciudad a mayúsculas y añadirlo a new_list
        new_list.append(city.upper())
    else:
        # Si no, convertir el nombre de la ciudad a minúsculas y añadirlo a new_list
        new_list.append(city.lower())

# new_list contiene los nombres de ciudades modificados
print("Lista modificada usando bucle for:", new_list)
```

```
Lista modificada usando bucle for: ['BARCELONA', 'madrid', 'girona', 'murcia']
```

```python
# Usando una comprensión de lista con if-else

new_list_comprehension = [city.upper() if len(city) > 6 else city.lower() for city in list_of_words]

# new_list_comprehension contiene los nombres de ciudades modificados
print("Lista modificada usando comprensión de lista:", new_list_comprehension)
```

```
Lista modificada usando comprensión de lista: ['BARCELONA', 'madrid', 'girona', 'murcia']
```

En el ejemplo proporcionado, la condicional (la declaración `if-else`) se coloca antes del valor que debe ser incluido en la nueva lista. Esta es una estructura común para las comprensiones de lista que permite aplicar condiciones en la construcción de nuevas listas de manera eficiente y compacta.

Sintaxis General

La sintaxis general para usar `if-else` en una comprensión de lista es la siguiente:

```python new_list = [value_if_true if condition else value_if_false for element in iterable]

## Compresiones de Listas Anidadas

Las **comprensiones de listas** son una característica poderosa en Python que permite crear listas a partir de iterables existentes de una manera concisa y legible. Las **comprensiones de listas**
```

anidadas llevan este concepto más allá al permitirte crear listas de listas, comúnmente referidas como "listas anidadas", de forma igualmente concisa y expresiva.

¿Qué son las Comprensiones de Listas Anidadas?

En las comprensiones de listas anidadas, puedes tener una o más cláusulas `for` e incluso incluir expresiones condicionales. Esta flexibilidad te permite construir estructuras de datos complejas y realizar operaciones avanzadas sobre datos anidados.

Las comprensiones de listas anidadas son particularmente útiles cuando necesitas trabajar con datos multidimensionales o transformar la estructura de listas anidadas de manera eficiente. En esta sección, exploraremos cómo usar las comprensiones de listas anidadas y demostraremos sus aplicaciones a través de ejemplos.

Ejemplo

```
nested_list = [[1, 2], [3,4], [5, 6]]
#output esperado, flat_list = [1, 2, 3, 4, 5, 6]

# Usando un bucle for

# Inicializar una lista vacía para almacenar los elementos planos
flat_list = []

# Iterar a través de cada sublista en la lista anidada
for sublist in nested_list:
    # Iterar a través de cada elemento en la sublista
    for item in sublist:
        flat_list.append(item)  # Agregar el elemento a la lista plana

# Mostrar el resultado
print("Lista plana usando bucle for:", flat_list)
```

```
Lista plana usando bucle for: [1, 2, 3, 4, 5, 6]
```

```
# Usando comprensión de lista anidada

flat_list_comprehension = [item for sublist in nested_list for item in
sublist]

# Mostrar el resultado
print("Lista plana usando comprensión de lista:",
flat_list_comprehension)
```

```
Lista plana usando comprensión de lista: [1, 2, 3, 4, 5, 6]
```

Desaplanando una lista anidada con una comprensión de lista

A veces, es necesario desaplanar listas anidadas que contienen múltiples niveles de anidación. En este ejemplo, vamos a trabajar con una lista anidada que tiene varios niveles de profundidad y aplanarla.

```python
# Lista anidada de múltiples niveles
nested = [[[[1, 2], [3, 4]]]]

# Usando un bucle for

flat_list = []  # Inicializar una lista vacía para almacenar los elementos aplanados

# Iterar a través de cada sublista en la lista anidada
for sublist1 in nested:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for item in sublist3:
                flat_list.append(item)  # Agregar el elemento a la lista plana

# Mostrar el resultado usando bucle for
print("Lista plana usando bucle for:", flat_list)
```

```
Lista plana usando bucle for: [1, 2, 3, 4]
```

```python
# Usando comprensión de lista anidada

flat_list_comprehension = [item for sublist1 in nested for sublist2 in sublist1 for sublist3 in sublist2 for item in sublist3]

# Mostrar el resultado usando comprensión de lista
print("Lista plana usando comprensión de lista:", flat_list_comprehension)
```

```
Lista plana usando comprensión de lista: [1, 2, 3, 4]
```

recordatorio amable

Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.[a]
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.[b]
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

¿Crees que estamos respetando el zen de Python?

Comprensiones de Diccionarios

Las **comprensiones de diccionarios** son una manera concisa de crear diccionarios en Python. Te permiten generar diccionarios usando una sola línea de código, haciendo tu código más legible y eficiente. Con las comprensiones de diccionarios, puedes iterar sobre objetos iterables, como listas, y especificar tanto las claves como los valores para tu diccionario. Esta característica es particularmente útil cuando necesitas transformar o filtrar datos mientras construyes un diccionario.

¿Por Qué Usar Comprensiones de Diccionarios?

- **Legibilidad**: Las comprensiones de diccionarios permiten expresar operaciones de manera más clara y concisa que los bucles tradicionales.

- **Eficiencia**: Generar un diccionario en una línea puede ser más rápido que usar un bucle, especialmente con grandes volúmenes de datos.
- **Flexibilidad**: Puedes aplicar condiciones para filtrar elementos o transformar datos mientras creas el diccionario.

En esta sección, exploraremos cómo usar las comprensiones de diccionarios para crear diccionarios con facilidad y claridad. Ya sea que quieras construir diccionarios basados en datos existentes o realizar transformaciones de datos, las comprensiones de diccionarios son una herramienta valiosa en tu kit de programación en Python.

Ejemplo: Creación de un Diccionario de Longitudes de Nombres

Supongamos que tenemos una lista de nombres y queremos crear un diccionario donde cada nombre sea una clave y su longitud sea el valor correspondiente.

```
# Lista de nombres
names = ["Alice", "Bob", "Charlie", "David"]

# Usando un bucle for
name_length_dict = {}  # Inicializar un diccionario vacío

# Iterar a través de la lista de nombres
for name in names:
    name_length_dict[name] = len(name)  # Asignar la longitud del
nombre como valor

# Mostrar el resultado usando bucle for
print("Diccionario de longitudes de nombres usando bucle for:",
name_length_dict)
```

```
Diccionario de longitudes de nombres usando bucle for: {'Alice': 5,
'Bob': 3, 'Charlie': 7, 'David': 5}
```

```
# Usando comprensión de diccionario
name_length_dict_comprehension = {name: len(name) for name in names}

# Mostrar el resultado usando comprensión de diccionario
print("Diccionario de longitudes de nombres usando comprensión de
diccionario:", name_length_dict_comprehension)
```

```
Diccionario de longitudes de nombres usando comprensión de
diccionario: {'Alice': 5, 'Bob': 3, 'Charlie': 7, 'David': 5}
```

Aplicaciones Prácticas de las Comprensiones de Diccionarios

1. **Transformación de Datos**: Puedes transformar datos en un formato diferente, como convertir una lista de tuplas en un diccionario.

```
tuples_list = [("a", 1), ("b", 2), ("c", 3)]
dict_from_tuples = {key: value for key, value in tuples_list}
print(dict_from_tuples)
```

```
{'a': 1, 'b': 2, 'c': 3}
```

1. **Filtrado de Datos**: Puedes filtrar elementos mientras construyes el diccionario.

```
numbers = [1, 2, 3, 4, 5, 6]
even_dict = {num: num**2 for num in numbers if num % 2 == 0}
print(even_dict)

{2: 4, 4: 16, 6: 36}
```

1. **Contar Elementos**: Puedes contar ocurrencias de elementos en una lista y almacenarlas en un diccionario.

```
print(set(["apple", "banana", "orange", "apple", "banana", "banana"]))

{'banana', 'apple', 'orange'}

fruits = ["apple", "banana", "orange", "apple", "banana", "banana"]
fruit_count = {fruit: fruits.count(fruit) for fruit in set(fruits)}
print(fruit_count)

{'banana': 3, 'apple': 2, 'orange': 1}
```

Desafios

1. **Análisis de Frases**: Dada una lista de frases, escribe un diccionario que contenga la cantidad de palabras en cada frase y la longitud promedio de esas palabras.

```
# Lista de frases
phrases = ["El gato está en el tejado", "El perro corre rápido", "La
lluvia es hermosa y refrescante"]

# Usando un bucle for

phrase_info = {}  # Inicializar un diccionario vacío

# Iterar a través de la lista de frases
for phrase in phrases:
    words = phrase.split()  # Dividir la frase en palabras
    num_words = len(words)  # Contar el número de palabras
    avg_length = sum(len(word) for word in words) / num_words  #
Calcular la longitud promedio de las palabras
    phrase_info[phrase] = {'word_count': num_words, 'average_length':
avg_length}  # Agregar la información al diccionario

# Mostrar el resultado
print(phrase_info)

{'El gato está en el tejado': {'word_count': 6, 'average_length':
3.3333333333333335}, 'El perro corre rápido': {'word_count': 4,
'average_length': 4.5}, 'La lluvia es hermosa y refrescante':
{'word_count': 6, 'average_length': 4.833333333333333}}
```

```
# Usando comprensión de diccionario
phrase_info_comprehension = {phrase: {'word_count': len(words :=
phrase.split()), 'average_length': sum(len(word) for word in words) /
len(words)} for phrase in phrases}

# Mostrar el resultado
print(phrase_info_comprehension)
```

```
{'El gato está en el tejado': {'word_count': 6, 'average_length':
3.3333333333333335}, 'El perro corre rápido': {'word_count': 4,
'average_length': 4.5}, 'La lluvia es hermosa y refrescante':
{'word_count': 6, 'average_length': 4.833333333333333}}
```

1. **Clasificación de Números en Diccionarios**: Dada una lista de números, crea un diccionario donde las claves sean los tipos (par, impar) y los valores sean listas de números que pertenezcan a cada tipo.

```
# Lista de números
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Usando un bucle for

classified_numbers = {'even': [], 'odd': []}  # Inicializar un
diccionario vacío

for num in numbers:
    if num % 2 == 0:
        classified_numbers['even'].append(num)  # Agregar a pares
    else:
        classified_numbers['odd'].append(num)  # Agregar a impares

print(classified_numbers)
```

```
{'even': [2, 4, 6], 'odd': [1, 3, 5]}
```

```
# Usando comprensión de diccionario

classified_numbers_comprehension = {
    t: [num for num in numbers if num % 2 == (0 if t == 'even' else
1)]
    for t in ['even', 'odd']
}

print(classified_numbers_comprehension)
```

```
{'even': [2, 4, 6], 'odd': [1, 3, 5]}
```

1. **Consolidación de Datos de Productos**: Dada una lista de diccionarios que representan productos con su nombre y precio, crea un diccionario que contenga el nombre del producto como clave y su precio total (sumando los precios si hay duplicados) como valor.

```python
# Lista de productos
products = [
    {'name': 'manzana', 'price': 1.5},
    {'name': 'banana', 'price': 1.0},
    {'name': 'manzana', 'price': 2.0},
    {'name': 'pera', 'price': 1.2},
]

# Usando un bucle for

product_totals = {}  # Inicializar un diccionario vacío

for product in products:
    name = product['name']
    price = product['price']
    if name in product_totals:
        product_totals[name] += price  # Sumar el precio al producto existente
    else:
        product_totals[name] = price  # Inicializar el precio para el nuevo producto

print(product_totals)
```

```
{'manzana': 3.5, 'banana': 1.0, 'pera': 1.2}
```

```python
# Usando comprensión de diccionario

product_totals_comprehension = {
    name: sum(product['price'] for product in products if product['name'] == name)
    for name in set(p['name'] for p in products)
}

print(product_totals_comprehension)
```

```
{'banana': 1.0, 'manzana': 3.5, 'pera': 1.2}
```

Caso de Negocio: Análisis de Retroalimentación del Cliente

Escenario Empresarial:

Trabajas para una empresa de venta al por menor que valora la retroalimentación de los clientes. La empresa recibe comentarios de los clientes sobre sus productos y servicios. Tu tarea es analizar estos comentarios para identificar palabras clave positivas y negativas comunes mencionadas por los clientes.

Instrucciones:

1. Se te proporciona una lista de comentarios de retroalimentación de los clientes. Cada comentario es una cadena de texto.

```
feedback_comments = [
    "¡El producto es excelente! Me encanta.",
    "Servicio al cliente terrible. Nunca volveré a comprar.",
    "Gran calidad y entrega rápida.",
    "Experiencia decepcionante. El artículo llegó dañado.",
    "Soberbio valor por el precio.",
    "El envío fue rápido y el producto es excelente.",
    "La atención al cliente es muy mala, no volveré.",
    "Me encantó el diseño del producto, es genial.",
    "La calidad del material es decepcionante.",
    "Un servicio increíble, definitivamente compraré de nuevo.",
    "El producto llegó en perfectas condiciones, ¡muy feliz!",
    "La experiencia de compra fue terrible y confusa.",
    "El valor es justo para la calidad ofrecida, soberbio.",
    "Desafortunadamente, el artículo llegó dañado.",
    "¡Excelente compra! Estoy muy satisfecho."
]
```

1. Proporciona el pseudocódigo.
2. Crea dos listas:
   - `palabras_clave_positivas`: Usa una comprensión de lista para extraer palabras clave positivas de los comentarios de retroalimentación. Las palabras clave positivas son palabras como "excelente", "encanta", "genial" y "soberbio".
   - `palabras_clave_negativas`: Usa otra comprensión de lista para extraer palabras clave negativas de los comentarios de retroalimentación. Las palabras clave negativas son palabras como "terrible", "decepcionante" y "dañado".

Consejos:

- Puedes usar el método split() para dividir un comentario en palabras.
- Asegúrate de que las palabras clave estén en minúsculas para capturar variaciones en el texto.
- Utiliza comprensiones de listas o diccionarios para realizar la tarea de manera eficiente.

```
#pseudocodigo

#1. Definir una lista que contenga comentarios de clientes sobre un
producto.
#   - Ejemplo de comentarios: "El producto es excelente", "Servicio
terrible", etc.

#2. Definir una lista de palabras que consideramos positivas.
#   - Ejemplo de palabras positivas: "excelente", "encanta", "genial",
"soberbio".
```

```
#3. Definir una lista de palabras que consideramos negativas.
#   - Ejemplo de palabras negativas: "terrible", "decepcionante",
"dañado".

#4. Inicializar un contador para las palabras positivas y otro
contador para las palabras negativas, ambos comenzando en cero.

#5. Para cada comentario en la lista de comentarios:
#   a. Convertir el comentario a minúsculas para que todas las
palabras se comparen de manera uniforme.
#   b. Para cada palabra en la lista de palabras positivas:
#      - Si la palabra está presente en el comentario, incrementar el
contador de palabras positivas.
#   c. Para cada palabra en la lista de palabras negativas:
#      - Si la palabra está presente en el comentario, incrementar el
contador de palabras negativas.

#6. Al finalizar, mostrar el total de palabras positivas y negativas
encontradas en los comentarios.

# Lista de comentarios de retroalimentación de los clientes
feedback_comments = [
    "¡El producto es excelente! Me encanta.",
    "Servicio al cliente terrible. Nunca volveré a comprar.",
    "Gran calidad y entrega rápida.",
    "Experiencia decepcionante. El artículo llegó dañado.",
    "Soberbio valor por el precio.",
    "El envío fue rápido y el producto es excelente.",
    "La atención al cliente es muy mala, no volveré.",
    "Me encantó el diseño del producto, es genial.",
    "La calidad del material es decepcionante.",
    "Un servicio increíble, definitivamente compraré de nuevo.",
    "El producto llegó en perfectas condiciones, ¡muy feliz!",
    "La experiencia de compra fue terrible y confusa.",
    "El valor es justo para la calidad ofrecida, soberbio.",
    "Desafortunadamente, el artículo llegó dañado.",
    "¡Excelente compra! Estoy muy satisfecho."
]

# Definir listas de palabras clave positivas y negativas
palabras_clave_positivas = ["excelente", "encanta", "genial",
"soberbio"]
palabras_clave_negativas = ["terrible", "decepcionante", "dañado"]

# Crear un diccionario para contar la frecuencia de palabras clave
frecuencia_palabras = {
    "positivas": sum(1 for comentario in feedback_comments for palabra
in palabras_clave_positivas
                     if palabra in comentario.lower()),  # Contar
palabras positivas
```

```
    "negativas": sum(1 for comentario in feedback_comments for palabra
in palabras_clave_negativas
                     if palabra in comentario.lower())  # Contar
palabras negativas
}

# Mostrar los resultados
print("Frecuencia de palabras clave:", frecuencia_palabras)
```

```
Frecuencia de palabras clave: {'positivas': 7, 'negativas': 6}
```

RECAP

Comprensiones de Lista

Las comprensiones de lista son una herramienta poderosa para crear listas basadas en listas existentes, todo en una única línea de código legible. Son especialmente útiles cuando quieres transformar o filtrar datos de una lista a otra.

Sintaxis Regular vs. Comprensión

Al trabajar con listas, tienes dos opciones principales: usar la sintaxis regular o usar comprensiones de lista. Cada enfoque tiene sus propias ventajas y casos de uso.

Sintaxis Regular

- **Más Fácil de Escribir:** Los bucles regulares usando declaraciones for y if son más flexibles y te permiten tener un control completo sobre lo que estás haciendo. Puedes añadir declaraciones de impresión dentro de los bucles para propósitos de depuración.
- **Más Control:** Puedes usar bucles regulares para escribir lógicas y condicionales más complejas.
- **Imprimir Dentro de los Bucles:** La depuración es más fácil con bucles regulares ya que puedes insertar declaraciones de impresión para ver resultados intermedios.

Comprensión

- **Más Eficiente:** Las comprensiones de lista son más eficientes y computacionalmente menos costosas, lo que las convierte en una buena elección para operaciones simples. Pueden mejorar el rendimiento del código.
- **Más Simple:** Las comprensiones de lista simplifican el código reduciendo el número de pasos requeridos. No hay necesidad de crear una nueva lista y usar declaraciones de añadir.
- **Más Rápido:** Las comprensiones de lista son más rápidas de escribir y a menudo resultan en código más conciso.

- **Versátiles:** Se pueden usar con listas, diccionarios, conjuntos y otros objetos iterables.

En resumen, las comprensiones de lista son una forma concisa y eficiente de crear listas, especialmente cuando la transformación de datos o filtrado es sencillo. Pueden mejorar tanto la legibilidad del código como el rendimiento en tales casos.