Mantengamos el flujo

¡Bienvenido de nuevo al emocionante mundo de la programación en Python! En nuestras sesiones anteriores, hemos dominado las declaraciones condicionales y explorado los bucles "for". Ahora, es momento de subir un nivel más y embarcarnos en un viaje fascinante a través del encantador reino de los bucles.

En este cuaderno, desentrañaremos los misterios de la declaración `while`, dominaremos los secretos de `pass`, `continue` y `break`, y exploraremos más a fondo el versátil bucle `for`. Aprenderemos a jugar con cadenas, movernos con listas, sincronizarnos con diccionarios y mucho más.

Así que abróchate el cinturón, ¡aventurero Pythonista! Es hora de sumergirse en el mundo de los bucles y abrazar el poder de la iteración. Ya seas un programador experimentado o apenas estés comenzando tu viaje en Python, ¡aquí hay algo para todos! ¡Zarpemos juntos en esta aventura de codificación! □□

La declaración `while`

En Python, la declaración `while` es una herramienta poderosa cuando necesitas ejecutar un bloque de código repetidamente mientras una condición específica se mantenga verdadera. Piensa en ella como tu fiel aliado para esos escenarios dinámicos donde no sabes exactamente cuántas iteraciones serán necesarias de antemano.

¿Cuándo usar `while` en lugar de `for`?

Imagina que estás construyendo una aplicación empresarial y necesitas procesar pedidos de clientes hasta que tu inventario esté vacío o hasta que se alcance un objetivo de ventas específico. Aquí es donde la declaración `while` destaca. En un contexto empresarial, usarías un bucle `while` en situaciones como:

1. **Condiciones dinámicas:** Cuando no sabes cuántas iteraciones necesitas, como al procesar pedidos hasta alcanzar un objetivo de ventas que puede cambiar día a día.

2. **Monitoreo en tiempo real:** Si necesitas supervisar un sistema o proceso en tiempo real y actuar con base en condiciones cambiantes, como ajustar la producción según la demanda.

3. **Interacción con el usuario:** Si estás desarrollando una aplicación interactiva que depende de la entrada del usuario, como un chatbot que responde a mensajes hasta que el usuario decide salir.

La declaración `while` permite que tu programa se ajuste dinámicamente a las condiciones cambiantes del mundo real. Continuará ejecutándose hasta que una condición específica se vuelva falsa, lo que la convierte en una opción ideal para escenarios donde la flexibilidad y capacidad de respuesta son esenciales.

Por otro lado, el bucle `for` sobresale cuando tienes una secuencia fija o una colección de elementos para recorrer. Sin embargo, en situaciones donde necesitas una iteración dinámica y condicionada, como en procesos empresariales en constante cambio, el bucle `while` es tu mejor aliado.

> **Nota:** ¡Cuidado con las recursiones infinitas! Asegúrate de que la condición del bucle cambie en algún momento para evitar que tu programa se ejecute indefinidamente.

Simulación de incremento de edad usando un bucle `while`

En este ejemplo, utilizamos un bucle `while` para simular el proceso de incremento de la edad de una persona desde los 10 hasta los 18 años, lo que representa el paso de la infancia a la adultez. A continuación, se presenta una explicación paso a paso del código:

1. **Inicialización:** Comenzamos con una variable `edad` establecida en 10, y definimos un `limite_edad` de 18. El bucle continuará ejecutándose hasta que la persona alcance los 18 años.

2. **Condición del Bucle:** El bucle `while` se ejecuta mientras la edad sea menor o igual al límite de edad (`18`). Si la condición es verdadera, el bucle sigue ejecutándose.

3. **Acciones Dentro del Bucle:** Dentro del bucle, imprimimos la edad actual utilizando una `f-string`, proporcionando un mensaje claro sobre el estado actual.

4. **Incremento de la Edad:** En cada iteración, incrementamos la edad en 1 utilizando la expresión `edad += 1`.
5. **Salida del Bucle:** El bucle se detiene automáticamente cuando la condición ya no se cumple (es decir, cuando la edad es mayor que 18).
6. **Mensaje Final:** Una vez que el bucle termina, imprimimos un mensaje que indica que la persona ha alcanzado la adultez.

Este ejemplo demuestra cómo usar un bucle `while` para realizar una serie de acciones repetitivas hasta que se cumpla una condición específica. Es un patrón común para tareas iterativas donde no se conoce de antemano el número exacto de iteraciones.

```
# Inicializar la variable de edad
edad = 10

# Definir el límite superior de edad para el bucle
limite_edad = 18

# Usar un bucle while para iterar mientras la edad sea menor o igual
al límite de edad
while edad <= limite_edad:
    # Imprimir la edad actual
    print(f"La edad actual es {edad}")

    # Incrementar la edad en 1 en cada iteración
    edad += 1

# El bucle se saldrá cuando la edad sea mayor que el límite de edad
print("¡Ahora eres un adulto!")
```

```
La edad actual es 10
La edad actual es 11
La edad actual es 12
La edad actual es 13
La edad actual es 14
La edad actual es 15
La edad actual es 16
La edad actual es 17
La edad actual es 18
¡Ahora eres un adulto!
```

Cálculo de Interés Compuesto usando un bucle `while`

En este ejemplo, utilizamos un bucle `while` para calcular el **interés compuesto** durante un número especificado de años. El interés compuesto es el interés calculado tanto sobre el capital inicial como sobre el interés acumulado de periodos anteriores.

Conceptos Clave:

- **Principal:** La cantidad inicial de dinero invertido o prestado.
- **Tasa de Interés Anual:** La tasa de interés anual como porcentaje.
- **Período de Tiempo:** El número de años durante los cuales queremos calcular el interés compuesto.
- **Frecuencia de Capitalización:** Indica con qué frecuencia se capitaliza el interés por año. En este ejemplo, se capitaliza **anualmente** (una vez al año).

Explicación:

1. **Inicialización:** Comenzamos con un **principal** de $1,000, una **tasa de interés anual** del 5%, y un **período de tiempo** de 10 años.

2. **Fórmula para el Cálculo del Interés Compuesto:**

   Fórmula de Interés Compuesto

   Donde:

   - **A** es el monto final (capital más interés acumulado).
   - **P** es el principal (cantidad inicial).
   - **r** es la tasa de interés anual (en forma decimal).
   - **n** es el número de veces que el interés se capitaliza por año.
   - **t** es el período de tiempo (en años).

3. **Frecuencia de Capitalización:** Establecemos `n` en 1, indicando que el interés se capitaliza **anualmente**.

4. **Uso del Bucle `while`:** Utilizamos un bucle `while` para iterar sobre cada año, desde 1 hasta el número especificado de años. En cada iteración, calculamos el monto del interés compuesto usando la fórmula.

5. **Salida del Monto Anual:** Imprimimos el monto acumulado al final de cada año, mostrando cómo crece la inversión con el tiempo.

Este ejemplo demuestra cómo un bucle `while` puede usarse para simular el crecimiento de una inversión a lo largo de varios años, considerando el interés compuesto. Es una aplicación práctica de matemáticas y programación en el ámbito financiero.

```
# Inicializar la cantidad principal, la tasa de interés y el período
de tiempo en años
principal = 1000
rate = 5  # Tasa de interés anual del 5%
years = 10

# Calcular el interés compuesto usando la fórmula
# A = P(1 + r/n)^(nt)
# donde A es la cantidad final, P es el capital inicial, r es la tasa
de interés anual, n es el número de veces que el interés se compone
```

```
por año, y t es el tiempo en años.
n = 1  # Compuesto anualmente

# Inicializar el contador de años
current_year = 1

while current_year <= years:
    # Calcular el interés compuesto para el año actual
    amount = principal * (1 + (rate / (100 * n))) ** (n *
current_year)

    # Imprimir el resultado para el año actual
    print(f"Year {current_year}: ${amount:.2f}")

    # Incrementar el año actual
    current_year += 1
```

```
Year 1: $1050.00
Year 2: $1102.50
Year 3: $1157.63
Year 4: $1215.51
Year 5: $1276.28
Year 6: $1340.10
Year 7: $1407.10
Year 8: $1477.46
Year 9: $1551.33
Year 10: $1628.89
```

Programa para validar una letra entre 'a' y 'd' usando un bucle `while`

En este ejemplo, utilizamos un bucle `while` para solicitar repetidamente al usuario que ingrese una opción hasta que proporcione una válida. El objetivo es asegurar que la entrada esté dentro de un rango específico de letras aceptadas.

Conceptos Clave:

- **Entrada del Usuario:** Le pedimos al usuario que ingrese una letra.
- **Validación:** Aseguramos que la letra ingresada sea una de las opciones válidas, es decir, entre "a" y "d".
- **Bucle `while`:** Utilizamos el bucle `while` para seguir pidiendo la entrada hasta que se proporcione una opción correcta.

Explicación:

1. **Entrada Inicial:** Pedimos al usuario que ingrese una letra usando la función `input()`. La entrada se almacena en una variable llamada `letra`.

2. **Opciones Válidas:** Creamos una lista `letras_aceptadas` que contiene las opciones válidas, que en este caso son `'a'`, `'b'`, `'c'` y `'d'`.

3. **Bucle `while`:** Usamos un bucle `while` para comprobar si la letra ingresada no está en la lista de letras aceptadas. Si no lo está, el bucle sigue ejecutándose y solicita al usuario otra letra.

4. **Validación Repetida:** Dentro del bucle, volvemos a pedir la letra hasta que el usuario ingrese una opción correcta, es decir, una letra que esté en la lista `letras_aceptadas`.

5. **Salida del Bucle:** Una vez que el usuario ingresa una letra correcta, el bucle termina y el programa puede proceder con los pasos siguientes.

Este ejemplo demuestra cómo crear un mecanismo de validación simple para asegurar que la entrada del usuario cumpla con ciertos criterios antes de permitir que el programa continúe.

```
# Inicializa la variable letter
letra = input("Letra entre a-d: ")

# Define las letras aceptadas
letras_aceptadas = ["a", "b", "c", "d"]

# Utiliza un bucle while para solicitar repetidamente una entrada
hasta que se proporcione una opción válida
while letra not in letras_aceptadas:
    print("Entrada inválida. Por favor, introduce una letra entre 'a'
y 'd'.")
    letra = input("Letra entre a-d: ")

# El bucle se sale cuando se proporciona una letra válida
print(f"Has introducido la letra: {letra}")
```

```
Letra entre a-d:  a

Has introducido la letra: a
```

Desafío Empresarial: Identificar Oportunidades de Inversión Rentables

Tarea: Escribe un programa en Python para ayudar a un analista financiero a identificar oportunidades de inversión potencialmente rentables basadas en datos históricos.

Instrucciones:

1. **Primero que todo, proporciona el pseudocódigo de este ejercicio.**
2. Pide al usuario que ingrese dos enteros: el número de oportunidades de inversión para analizar y el margen de beneficio mínimo aceptable (como porcentaje).

3. Para cada oportunidad de inversión, solicita al usuario que ingrese el nombre de la inversión y su margen de beneficio asociado (como porcentaje).
4. Verifica e imprime los nombres de las oportunidades de inversión que cumplan o superen el margen de beneficio mínimo aceptable.
5. Asegúrate de que el usuario ingrese un número válido de oportunidades y un margen de beneficio válido.
6. Utiliza un bucle `for` para iterar a través de cada oportunidad de inversión.
7. Implementa una condición para verificar si el margen de beneficio es mayor o igual al margen aceptable mínimo.
8. Imprime los nombres de las oportunidades de inversión rentables.

Consejos:

- Asegúrate de que el usuario ingrese un número válido de oportunidades de inversión y margen de beneficio (ambos deben ser positivos).
- Utiliza cadenas formateadas para mostrar los resultados claramente.

Ejemplo de Salida:

```python ¡Bienvenido al Analizador de Oportunidades de Inversión! Por favor, introduce el número de oportunidades de inversión: 3 Por favor, introduce el margen de beneficio mínimo aceptable (como porcentaje): 10

Introduce los detalles de la oportunidad de inversión 1: Nombre de la Inversión: Stock XYZ Margen de Beneficio (como porcentaje): 15

Introduce los detalles de la oportunidad de inversión 2: Nombre de la Inversión: Proyecto Inmobiliario Margen de Beneficio (como porcentaje): 8

Introduce los detalles de la oportunidad de inversión 3: Nombre de la Inversión: Startup Tecnológica Margen de Beneficio (como porcentaje): 12

Oportunidades de inversión rentables (con un margen de beneficio del 10% o superior):

1. Stock XYZ
2. Startup Tecnológica
```

```python
# Bienvenida al programa
print("¡Bienvenido al Analizador de Oportunidades de Inversión!")

# Solicitar número de oportunidades y margen de beneficio mínimo, con validación
while True:
    try:
        num_oportunidades = int(input("Por favor, introduce el número de oportunidades de inversión: "))
        margen_minimo = float(input("Por favor, introduce el margen de beneficio mínimo aceptable (como porcentaje): "))
```

```
        if num_oportunidades > 0 and margen_minimo > 0:
            break
        else:
            print("Ambos valores deben ser mayores que 0. Inténtalo de
nuevo.")
    except ValueError:
        print("Entrada no válida. Por favor, introduce números enteros
y decimales correctamente.")
```

```
¡Bienvenido al Analizador de Oportunidades de Inversión!

Por favor, introduce el número de oportunidades de inversión:  3
Por favor, introduce el margen de beneficio mínimo aceptable (como
porcentaje):  xyz

Entrada no válida. Por favor, introduce números enteros y decimales
correctamente.
```

```
# Lista para almacenar oportunidades rentables
oportunidades_rentables = []

# Bucle para analizar cada oportunidad de inversión
for i in range(1, num_oportunidades + 1):
    print(f"\nIntroduce los detalles de la oportunidad de inversión
{i}:")
    nombre_inversion = input("Nombre de la Inversión: ")

    while True:
        try:
            margen_beneficio = float(input("Margen de Beneficio (como
porcentaje): "))
            if margen_beneficio >= 0:
                break
            else:
                print("El margen de beneficio debe ser un valor
positivo. Inténtalo de nuevo.")
        except ValueError:
            print("Entrada no válida. Por favor, introduce un número
válido para el margen de beneficio.")

    # Verificar si la inversión es rentable
    if margen_beneficio >= margen_minimo:
        oportunidades_rentables.append(nombre_inversion)

# Mostrar las inversiones rentables
print(f"\nOportunidades de inversión rentables (con un margen de
beneficio del {margen_minimo}% o superior):")

if oportunidades_rentables:
    for idx, inversion in enumerate(oportunidades_rentables, 1):
        print(f"{idx}. {inversion}")
```

```
else:
    print("No hay oportunidades de inversión que cumplan con el margen de beneficio mínimo.")
```

Pass, continue y break

En Python, contamos con declaraciones de control de flujo que nos permiten gestionar el flujo de nuestros programas. Estas declaraciones son esenciales para tomar decisiones, controlar bucles y manejar diversas situaciones. Vamos a explorar tres declaraciones de control de flujo importantes: `pass`, `continue` y `break`. Puedes consultar más detalles en la documentación oficial.

Pass

La declaración `pass` sirve como un marcador de posición en Python. Cuando estás escribiendo código y necesitas una declaración en un lugar determinado pero no quieres que realice ninguna acción aún, puedes usar `pass`. Es una forma de decirle a Python: "Sé que debería haber algo aquí, pero aún no estoy listo para escribirlo."

Se utiliza comúnmente cuando estás diseñando la estructura de tu código y necesitas completar los detalles más tarde. Por ejemplo, al definir una función o una clase, podrías usar `pass` hasta que estés listo para escribir el código real dentro de ellas.

Continue

La declaración `continue` se utiliza dentro de los bucles (como `for` o `while`) para saltar el resto de la iteración actual y pasar a la siguiente. Imagina que estás iterando sobre una lista de números y, cuando se cumple una condición específica, deseas saltarte ese número y proceder al siguiente. Puedes usar `continue` para lograr esto. Es como decir: "He terminado con esta parte del bucle; vamos al siguiente elemento."

Break

La declaración `break` también se utiliza dentro de los bucles, pero es más poderosa. Cuando se encuentra, sale inmediatamente del bucle, incluso si las condiciones del bucle no se cumplen. Es como decir: "He terminado con este bucle; salgamos de aquí." Podrías usar `break` cuando buscas un elemento específico en una lista. Una vez que lo encuentres, puedes salir del bucle en lugar de continuar iterando innecesariamente.

Resumen

Estas declaraciones de control de flujo te ayudan a gestionar el flujo de tu programa y a tomar decisiones basadas en condiciones, asegurando que tu código se comporte de la manera que pretendes. Son herramientas esenciales para cualquier programador y pueden simplificar la lógica compleja.

- **`pass`**: ignora la ejecución y permite que el programa continúe sin realizar ninguna acción en ese punto.

- **`continue`**: salta a la siguiente iteración del bucle, ignorando el resto del código en la iteración actual.
- **`break`**: termina el bucle de inmediato y sale de él, independientemente de las condiciones.

Estas herramientas son fundamentales para mejorar la legibilidad y la eficiencia de tu código.

Ejemplo 1:

```
# Ejemplo usando la declaración pass dentro de un bloque condicional
x = 2

if x > 5:
    print("x es mayor que 5")
elif x < 5:
    pass # trabajo en el futuro
else:
    print('revisión hecha')

# Explicación:
# En este ejemplo, tenemos una variable x con un valor de 10.
# Utilizamos una declaración condicional (if-elif-else) para comprobar
el valor de x.
# - Si x es mayor que 5, imprime el mensaje "x es mayor que 5."
# - Si x es menor que 10 pero no mayor que 5, la declaración pass se
utiliza como un marcador de posición.
# - En el bloque "else", imprime el mensaje "revisión hecha."
```

Ejemplo 2:

```
# Ejemplo usando la declaración pass dentro de un bucle
list_of_names = ["Alice", "Bob", "Charlie", "Dave", "Eve"]

for i in list_of_names:
    if i.startswith("C"):
        pass
    else:
        print("No comienza con una C")

# Explicación:
# Tenemos una lista de nombres almacenada en la variable
'list_of_names'.
# Usamos un bucle 'for' para iterar a través de cada nombre en la
lista.
# Dentro del bucle, verificamos si el nombre comienza con la letra 'C'
usando el método 'startswith'.
# Aún no sabemos qué hará la condición, pero nos permite verificar si
el bucle está funcionando.
```

- `continue`: para y va a la siguiente iteración.

Ejemplo 3:

```
list_of_names = ["Clara", "Albert", "Laura", "Carla"]

for i in list_of_names:
    if i.startswith("C"):
        # Continúa con la SIGUIENTE iteración si el nombre comienza
con "C"
        continue  # Esta línea instruye al programa a saltar el código
restante en esta iteración
        print(i)  # Esta línea no se ejecuta para nombres que
comienzan con "C"
    else:
        # Imprime los nombres que no comienzan con "C"
        print(i)
```

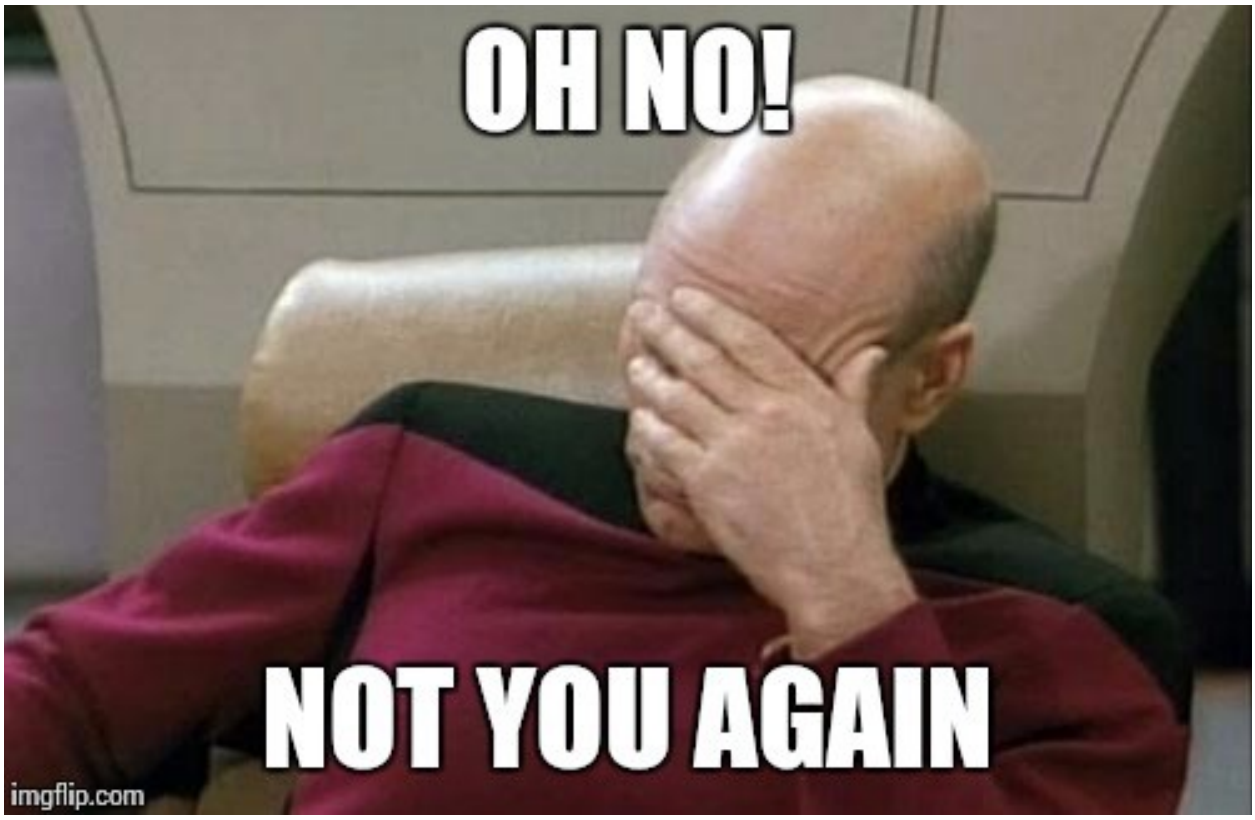
- `break` para el loop

Ejemplo 4:

```
# Lista de nombres
list_of_names = ["Albert", "Clara", "Laura", "Carla"]

# Iterar a través de la lista
for i in list_of_names:
    # Verificar si el nombre comienza con "C"
    if i.startswith("C"):
        # Si es así, salir del bucle inmediatamente usando "break"
        break
        # La siguiente línea no se ejecutará para nombres que
comiencen con "C"
        print(i)
    else:
        # Imprimir nombres que no comienzan con "C"
        print(i)
```

For



En nuestro viaje para dominar Python, ya nos hemos familiarizado con los conceptos básicos de los bucles `for`, que nos permiten iterar a través de secuencias de datos. Ahora es momento de adentrarnos en aspectos más avanzados de los bucles `for` que nos proporcionan un mayor control y versatilidad.

El Poder del Rango

Imagina tener la capacidad de generar secuencias de números sin esfuerzo, proporcionando una forma estructurada de controlar tus bucles. Ahí es donde entra en juego la función `range()`. Exploraremos cómo usarla para definir el inicio, el fin y el paso de tus secuencias. Esto te permitirá, por ejemplo, iterar desde un número específico hasta otro, ajustando el incremento según tus necesidades.

Iterando a Través de Listas

Aunque iterar a través de una lista es un caso de uso común, descubriremos técnicas avanzadas. Aprenderemos a acceder a los elementos tanto por su valor como por su índice, lo que nos permitirá realizar operaciones más complejas de manera eficiente. Usar la función `enumerate()` será clave para esto, ya que nos proporciona tanto el índice como el valor del elemento en cada iteración.

Bucle Simultáneo con Zip

A veces, necesitamos trabajar con múltiples listas o secuencias en paralelo. La función `zip()` de Python nos permite emparejar elementos correspondientes de diferentes secuencias, simplificando las tareas que involucran trabajar con datos correlacionados. Esto es especialmente útil en situaciones donde quieres procesar dos o más listas de manera conjunta, manteniendo su relación.

Aprovechando el Poder de los Diccionarios

Los diccionarios, que son las tiendas de valores clave de Python, son estructuras de datos versátiles. Exploraremos cómo iterar a través de las claves, los valores o ambos de un diccionario. Aprender a manipular y extraer datos dentro de este contenedor estructurado te proporcionará valiosas perspectivas sobre la gestión de información en tus programas.

Conclusión

A través de estos conceptos avanzados, nos equiparemos con las herramientas necesarias para abordar desafíos de programación aún más complejos. ¡Embarquémonos en este viaje para desbloquear el potencial completo de los bucles `for` en Python!

Ejemplo 1:

```
number = "5678"

# Descomenta la siguiente línea para convertir 'number' en un entero
# number = int(number)

# Itera a través de cada carácter en la cadena 'number'
for i in number:
    # Imprime cada carácter
    print(i)
```

Ejemplo 2:

```
# No se pueden iterar números
number = 5678.567
number_str = str(number)  # Convierte el número a una cadena
for digit in number_str:
    print(digit)
```

Ejemplo 3:

```
greetings_list = ["Hello", "How", "are", "you"]

# Itera a través de cada elemento en greetings_list
for _ in greetings_list:
    # Imprime "¡Hola!" para cada elemento (el guion bajo _ se usa
```

```
cuando no necesitamos el elemento actual)
    print("¡Hola!")
```

Cuando necesitamos un contador que se incremente con cada iteración, Python proporciona la función `range()`, que genera secuencias de enteros. Esta función es versátil y puede ser invocada de diferentes maneras:

- **`range(int)`**: Cuando se llama con un solo argumento, genera una secuencia de enteros que comienza desde 0 hasta (pero sin incluir) el entero especificado. Por ejemplo, `range(5)` generará la secuencia `[0, 1, 2, 3, 4]`.

Además de esta forma básica, `range()` también se puede utilizar con más argumentos para mayor flexibilidad:

- **`range(start, stop)`**: Esta forma permite especificar un valor inicial (inclusive) y un valor final (exclusivo). Por ejemplo, `range(2, 6)` generará la secuencia `[2, 3, 4, 5]`.
- **`range(start, stop, step)`**: En esta variante, puedes definir un valor de paso que determina la diferencia entre cada número en la secuencia. Por ejemplo, `range(0, 10, 2)` generará la secuencia `[0, 2, 4, 6, 8]`, incrementando de 2 en 2.

La función `range()` es una herramienta poderosa y flexible para manejar bucles en Python, permitiéndote personalizar el comportamiento de tus iteraciones de manera efectiva.

```
# Define una lista de saludos
greetings_list = ["Hello", "How", "are", "you"]

# Usa un bucle for para iterar sobre un rango de índices de la lista
for i in range(len(greetings_list)):
    # Imprime "¡Hola!" en cada iteración
    print("¡Hola!")

# Usa un bucle for para iterar sobre un rango de números desde 0 hasta
3 (exclusivo)
for i in range(4):
    # Imprime el valor actual de 'i' durante cada iteración
    print(i)
```

- **`range(start, stop)`**: Esta variante de la función `range()` se llama con dos argumentos. El primer parámetro es el valor inicial (inclusive) y el segundo es el valor final (exclusivo). Esto significa que la secuencia generada incluirá el valor de `start` pero no incluirá el valor de `stop`.

  Por ejemplo, `range(3, 8)` generará la secuencia `[3, 4, 5, 6, 7]`, que incluye 3 y termina justo antes de 8.

```python
# Usa un bucle for para iterar sobre un rango de números desde 10
hasta 20 (exclusivo)
for i in range(10, 21):
    # Imprime el valor actual de 'i' durante cada iteración
    print(i)
```

- **`range(start, stop, step)`**: Esta variante de la función `range()` se utiliza con tres argumentos. El primer parámetro es el valor inicial (inclusive), el segundo es el valor final (exclusivo), y el tercer parámetro indica el incremento que ocurre de un elemento al siguiente.

  Por ejemplo, `range(1, 10, 2)` generará la secuencia `[1, 3, 5, 7, 9]`, comenzando en 1, incrementando de 2 en 2, y deteniéndose antes de alcanzar 10.

  También puedes usar un paso negativo para contar hacia atrás, como en `range(10, 0, -2)`, que generará la secuencia `[10, 8, 6, 4, 2]`.

```python
# Usa un bucle for para iterar sobre un rango de números desde 10
hasta 20 (exclusivo), con un incremento de dos
for i in range(10, 21, 2):
    # Imprime el valor actual de 'i' durante cada iteración
    print(i)
```

Iterando a través de cadenas

Cuando trabajas con cadenas en Python, a menudo necesitas acceder a caracteres individuales o subcadenas dentro de una cadena. El proceso de iterar a través de una cadena implica recorrer cada carácter uno por uno. Python proporciona formas simples y eficientes de iterar a través de cadenas, lo que te permite realizar diversas operaciones en ellas.

Vamos a explorar cómo iterar a través de cadenas en Python y aprovechar esta capacidad para una amplia gama de tareas de procesamiento de texto.

Ejemplo 1:

```python
# Variable inicial
greeting = "Hello How Are youuUUuu"

# qué crees que hará
for i in greeting:
    print(i)
```

Ejemplo 2:

```python
# Crear una lista vacía para almacenar los caracteres en mayúsculas de
la cadena 'greeting'.
new_list = []

# Iterar a través de cada carácter (letra) en la cadena 'greeting'.
```

```
for i in greeting:
    # Verificar si el carácter actual 'i' es mayúscula.
    if i.isupper():
        # Si el carácter es mayúscula, añadirlo a 'new_list'.
        new_list.append(i)

# Eliminar duplicados convirtiendo 'new_list' en un conjunto y luego de vuelta a una lista.
new_list = list(set(new_list))

# 'new_list' ahora contiene caracteres únicos en mayúsculas de la cadena 'greeting'.
new_list
```

Iterando a través de listas por su índice

Cuando trabajas con listas en Python, puedes iterar a través de los elementos de una lista utilizando sus índices. Este enfoque te permite acceder tanto al índice como al elemento en sí dentro del bucle. En esta sección, exploraremos cómo iterar a través de listas por su índice, junto con algunas funciones integradas y la función `enumerate()`, que puede ser particularmente útil en tales escenarios.

Para obtener más detalles, puedes referirte a la documentación de Funciones Integradas de Python y la función enumerate().

Uso de `range()` para Iterar

Una forma común de iterar a través de una lista por su índice es utilizando la función `range()` junto con `len()`. Aquí tienes un ejemplo:

Ejemplo 1:

```
fruits = ["apple", "banana", "cherry", "date"]

# Iterando a través de la lista usando un bucle for y la función range()
for i in range(len(fruits)):
    print(f"Índice {i}: {fruits[i]}")
```

Ejemplo 2:

```
fruits = ["apple", "banana", "cherry", "date"]

# Iterando a través de la lista usando la función enumerate()
for index, fruit in enumerate(fruits):
    print(f"Índice {index}: {fruit}")
```

Iterar a través de dos listas al mismo tiempo

En Python, a menudo te encuentras en situaciones donde necesitas trabajar con múltiples listas simultáneamente. Ya sea que estés comparando elementos, realizando cálculos o extrayendo datos de dos listas, Python proporciona una manera conveniente de iterar a través de ellas juntas usando la función `zip`.

La función zip empareja elementos de dos o más listas, creando un iterable que combina elementos correspondientes. Esto te permite acceder a elementos de cada lista simultáneamente durante el proceso de iteración.

Es importante tener en cuenta que la función `zip` funciona sin problemas incluso si las listas son de diferentes tamaños. Sin embargo, la iteración continuará hasta que la lista más corta se haya agotado.

Ejemplo 1:

```
# Dos listas conteniendo nombres y ciudades
list_of_names = ["Albert", "Clara", "Laura"]
list_of_cities = ["Mataró", "Bcn", "Mataró"]

# Bucle a través de las listas simultáneamente usando zip
for name, city in zip(list_of_names, list_of_cities):
    # Imprime un mensaje formateado para cada par de elementos
    print(f"Me llamo {name} y soy de {city}")
```

Ejemplo 2:

```
# Listas conteniendo nombres, ciudades y edades
list_of_names = ["Albert", "Clara", "Laura"]
list_of_cities = ["Mataró", "Bcn", "Mataró"]
list_of_ages = [30, 30, 30]

# Bucle a través de las listas simultáneamente usando enumerate y zip
for index, (name, city, age) in enumerate(zip(list_of_names,
list_of_cities, list_of_ages)):
    # Imprime un mensaje formateado para cada persona
    print(f"Persona {index + 1}: 'Me llamo {name}, y soy de {city}.
Tengo {age} años'")
```

- `zip_longest`:que pas si quiero mantener la lista más larga?

Ejemplo 3:

```
from itertools import zip_longest

# Listas con diferentes longitudes
list1 = [1, 2, 3, 4, 5]
list2 = ['a', 'b', 'c']
```

```
# Usando zip_longest para mantener la lista más larga
result = list(zip_longest(list1, list2, fillvalue=None))

# Imprimiendo el resultado
print(result)
```

Ejemplo 4:

```
from itertools import zip_longest

list1 = [1, 2, 3, 4, 5]
list2 = ['a', 'b', 'c']

# Usando zip_longest para mantener la lista más larga
result = list(zip_longest(list1, list2, fillvalue=None))

# Iterando a través del resultado combinado en un bucle for
for item in result:
    print(item)
```

Iterar sobre los elementos de un diccionario

Iterar sobre los elementos de un diccionario te permite acceder y procesar cada par clave-valor (ítem) almacenado en el diccionario uno por uno. Python proporciona varios métodos y técnicas para lograr esto, facilitando el trabajo con datos de diccionario.

Puedes iterar sobre un diccionario de diferentes maneras, dependiendo del aspecto específico del diccionario con el que quieras trabajar:

1. **Iterar Sobre las Claves:**
   Puedes usar un bucle `for` para iterar sobre las claves de un diccionario. En este caso, iteras a través de las claves del diccionario, y luego puedes acceder a los valores correspondientes usando estas claves.

2. **Iterar Sobre Pares Clave-Valor:**
   Puedes usar el método `items()` para iterar tanto sobre las claves como sobre los valores simultáneamente. Este método devuelve un objeto de vista que contiene tuplas de pares clave-valor, los cuales puedes desempaquetar fácilmente dentro del bucle.

3. **Iterar Sobre los Valores:**
   Si solo necesitas iterar a través de los valores en el diccionario, puedes usar el método `values()`. Esto te permite acceder y procesar los valores sin necesidad de las claves asociadas.

Iterar sobre los elementos de un diccionario es una técnica fundamental para trabajar con datos estructurados en Python. Dependiendo de tus necesidades específicas, puedes elegir el método apropiado para acceder y manipular los datos dentro del diccionario.

- **Iterar Sobre las Claves:** Puedes usar un bucle `for` para iterar sobre las claves de un diccionario. En este caso, iteras a través de las claves del diccionario, y luego puedes acceder a los valores correspondientes usando estas claves. Aquí tienes un ejemplo:

```
# Definir un diccionario de calificaciones de estudiantes
student_grades = {
    "Alice": 95,
    "Bob": 89,
    "Charlie": 92,
    "David": 88
}

# Iterar sobre las claves e imprimir los nombres de los estudiantes y
sus calificaciones
for student in student_grades:
    print(f"{student}: {student_grades[student]}")
```

- **Iterar Sobre Pares Clave-Valor:**
  Puedes usar el método `items()` para iterar tanto sobre las claves como sobre los valores de un diccionario simultáneamente. Este método devuelve un objeto de vista que contiene tuplas de pares clave-valor, los cuales puedes desempaquetar fácilmente dentro del bucle. Esto te permite procesar tanto la clave como el valor en una sola iteración, lo que es muy útil para realizar operaciones que dependen de ambos elementos.

```
# Definir un diccionario de calificaciones de estudiantes
student_grades = {
    "Alice": 95,
    "Bob": 89,
    "Charlie": 92,
    "David": 88
}

# Iterar sobre pares clave-valor e imprimir los nombres de los
estudiantes y sus calificaciones
for student, grade in student_grades.items():
    print(f"{student}: {grade}")
```

- **Iterar Sobre los Valores:**
  Si solo necesitas iterar a través de los valores en un diccionario, puedes usar el método `values()`. Este método devuelve un objeto de vista que contiene todos los valores del diccionario. Al iterar sobre estos valores, puedes acceder y procesar la información sin necesidad de las claves asociadas, lo que simplifica el código cuando solo te interesan los valores.

```
# Definir un diccionario de calificaciones de estudiantes
student_grades = {
    "Alice": 95,
    "Bob": 89,
    "Charlie": 92,
```

```
    "David": 88
}

# Iterar sobre los valores e imprimir las calificaciones de los
estudiantes
for grade in student_grades.values():
    print(grade)
```

Business Challenge: Gestión de Datos de una Compañía de Taxis

Escenario empresarial:
Se te ha encargado desarrollar un programa en Python para una compañía de taxis en Barcelona. El programa debería ayudar a la compañía a gestionar datos relacionados con sus taxis, conductores y viajes. La compañía de taxis necesita almacenar y manipular estos datos de manera eficiente para optimizar sus operaciones.

Instrucciones:

1. **Crear un diccionario para almacenar información sobre los conductores de taxi.**
   Cada conductor debe tener un ID de conductor único (un valor numérico) y los siguientes detalles:
   - Nombre completo del conductor
   - Número de contacto
   - Número total de viajes realizados
2. **Crear una lista para almacenar información sobre los viajes en taxi.**
   Cada viaje debe incluir los siguientes detalles:
   - ID del viaje (un identificador único)
   - ID del conductor (para asociar el viaje con un conductor)
   - Nombre del cliente
   - Distancia del viaje (en kilómetros)
   - Tarifa del viaje (en euros)
3. **Implementar las siguientes funcionalidades dentro de tu programa en Python:**
   - Proporcionar el pseudocódigo.
   - Añadir nuevos conductores al diccionario de conductores.
   - Registrar nuevos viajes en taxi con los detalles relevantes.
   - Calcular y actualizar el número total de viajes realizados por cada conductor.
   - Calcular los ingresos totales generados por la compañía de taxis.
   - Encontrar al conductor con el mayor número de viajes.
   - Encontrar al conductor con los ingresos totales más altos.
   - Listar todos los viajes realizados por un conductor específico.

```
# Pseudocódigo
# 1. Crear un diccionario vacío llamado 'conductores' para almacenar
la información de cada conductor
# 2. Crear una lista vacía llamada 'viajes' para almacenar los
detalles de cada viaje realizado
# 3. Añadir nuevos conductores al diccionario 'conductores' con un ID
```

```
único y sus datos:
#     - Nombre completo
#     - Número de contacto
#     - Total de viajes realizados (inicialmente 0)
# 4. Registrar nuevos viajes en la lista 'viajes', cada viaje debe
incluir:
#     - ID del viaje (un identificador único)
#     - ID del conductor (para asociar el viaje con el conductor
correspondiente)
#     - Nombre del cliente que solicitó el viaje
#     - Distancia del viaje en kilómetros
#     - Tarifa del viaje en euros
# 5. Para cada viaje registrado, actualizar el total de viajes
realizados por el conductor correspondiente:
#     - Incrementar el total de viajes del conductor según el ID del
viaje
# 6. Calcular los ingresos totales generados por la compañía sumando
las tarifas de todos los viajes
# 7. Encontrar al conductor que ha realizado el mayor número de
viajes:
#     - Usar la función 'max' sobre el diccionario 'conductores' para
obtener el conductor con más viajes
# 8. Calcular los ingresos totales generados por cada conductor:
#     - Crear un diccionario temporal para almacenar los ingresos
acumulados por cada conductor
#     - Para cada viaje, sumar la tarifa al total del conductor
correspondiente
# 9. Encontrar al conductor con los ingresos totales más altos:
#     - Usar la función 'max' sobre el diccionario de ingresos por
conductor
# 10. Listar todos los viajes realizados por un conductor específico:
#     - Filtrar la lista 'viajes' para obtener solo aquellos que
coincidan con el ID del conductor

# 1. Crear un diccionario para almacenar información sobre los
conductores de taxi
conductores = {}

# 2. Crear una lista para almacenar información sobre los viajes en
taxi
viajes = []

# 3. Añadir nuevos conductores al diccionario de conductores
conductores[1] = {
    "nombre": "Juan Pérez",
    "numero_contacto": "123456789",
    "total_viajes": 0
}

conductores[2] = {
```

```
    "nombre": "María Gómez",
    "numero_contacto": "987654321",
    "total_viajes": 0
}

# 4. Registrar nuevos viajes en taxi con los detalles relevantes
viaje_1 = {
    "id_viaje": 101,
    "id_conductor": 1,
    "nombre_cliente": "Carlos Ruiz",
    "distancia": 5,  # en kilómetros
    "tarifa": 15.0   # en euros
}

viaje_2 = {
    "id_viaje": 102,
    "id_conductor": 1,
    "nombre_cliente": "Lucía Martínez",
    "distancia": 3,
    "tarifa": 10.0
}

viaje_3 = {
    "id_viaje": 103,
    "id_conductor": 2,
    "nombre_cliente": "Antonio López",
    "distancia": 10,
    "tarifa": 25.0
}

# Agregar los viajes a la lista de viajes
viajes.append(viaje_1)
viajes.append(viaje_2)
viajes.append(viaje_3)

# 5. Actualizar el número total de viajes realizados por cada
conductor
conductores[1]["total_viajes"] += 2  # Conductor 1 tiene 2 viajes
conductores[2]["total_viajes"] += 1  # Conductor 2 tiene 1 viaje

# 6. Calcular los ingresos totales generados por la compañía de taxis
ingresos_totales = 0
for viaje in viajes:
    ingresos_totales += viaje["tarifa"]

# 7. Encontrar al conductor con el mayor número de viajes
max_viajes = -1
conductor_max_viajes_id = None
for id_conductor, datos in conductores.items():
    if datos["total_viajes"] > max_viajes:
```

```
        max_viajes = datos["total_viajes"]
        conductor_max_viajes_id = id_conductor

# 8. Encontrar al conductor con los ingresos totales más altos
ingresos_por_conductor = {}
for viaje in viajes:
    id_conductor = viaje["id_conductor"]
    if id_conductor not in ingresos_por_conductor:
        ingresos_por_conductor[id_conductor] = 0
    ingresos_por_conductor[id_conductor] += viaje["tarifa"]

max_ingresos = -1
conductor_max_ingresos_id = None
for id_conductor, total_ingresos in ingresos_por_conductor.items():
    if total_ingresos > max_ingresos:
        max_ingresos = total_ingresos
        conductor_max_ingresos_id = id_conductor

# 9. Listar todos los viajes realizados por un conductor específico
id_conductor_especifico = 1  # Por ejemplo, el conductor con ID 1
viajes_conductor_especifico = []
for viaje in viajes:
    if viaje["id_conductor"] == id_conductor_especifico:
        viajes_conductor_especifico.append(viaje)

# Salidas de resultados
print(f"Ingresos totales generados: €{ingresos_totales}")
print(f"Conductor con más viajes:
{conductores[conductor_max_viajes_id]['nombre']} (ID:
{conductor_max_viajes_id})")
print(f"Conductor con más ingresos:
{conductores[conductor_max_ingresos_id]['nombre']} (ID:
{conductor_max_ingresos_id})")
print(f"Viajes realizados por el conductor
{conductores[id_conductor_especifico]['nombre']}:
{viajes_conductor_especifico}")
```

Resumen

Es tu turno, ¿qué hemos aprendido hoy?

CONDICIONES

- `if/elif/else`
  - if vs elif
    - if & if: más de 1 condición a la vez: no exclusivas
    - if & elif: condiciones exclusivas
  - else: útil pero ten cuidado de no ser demasiado general
- `booleans: tipo de dato`

  - verdaderos/falsos
    - verdadero: el resto
    - falso: vacío, 0, False, None
  - bool()
  - True / False

BUCLE

- while (mientras haya una condición que no se cumpla)
  - hasta que se cumpla una condición: mantener las cosas funcionando hasta que suceda
- for (mientras haya elementos para iterar)
  - iterables:
    - tipos de datos: cadenas
    - estructuras de datos: listas, tuplas, conjuntos, diccionarios
      - diccionario: i, .items, .keys, .values
  - elementos Y/O el índice
    - `for elemento in iterable:`
      - elemento: elemento
    - `for indice in range(len(iterable)):`
      - `indice`
      - elemento: iterable[indice]
  - rangos
    - range(4): 0, 1, 2, 3
    - range(10, 21, 2): 10, 12, 14, 16, 18, 20
  - enumerate: índice y elemento
    - `for indice, elemento in iterable`
  - zip: más de un iterable a la vez
    - `for el1, el2, el3 in zip(iterable1, iterable2, iterable3)`
  - zip y enumerate pueden usarse juntos
  - los bucles pueden anidarse: [[[[]]]]
    - for i in iterable
      - for j in i
        - for k in j - for l in k ....
- pass/continue/break
  - pass: ignora
  - continue: ignora lo de abajo y continúa con la siguiente iteración
  - break: detiene todo el bucle