Conferencia 3: Control de Flujo Iterativo

Introducción

En sesiones anteriores, se han cubierto los fundamentos de la ejecución secuencial y la ramificación del flujo de control mediante estructuras condicionales (`if`/`elif`/`else`). El código desarrollado hasta ahora sigue una trayectoria lineal, ejecutándose una única vez de principio a fin.

El objetivo de esta sesión es introducir las **estructuras de control iterativas**, comúnmente conocidas como **ciclos** o **bucles**. Estas construcciones son esenciales para la computación, ya que permiten la ejecución repetida de un bloque de código, un requisito para procesar colecciones de datos, implementar algoritmos complejos y automatizar tareas a escala.

---

1. Modelo de Ejecución del CPU

La Unidad Central de Procesamiento (CPU) opera siguiendo un modelo de ejecución secuencial. Un puntero de instrucción interno avanza a través del código máquina, ejecutando una instrucción tras otra. Las estructuras de control de flujo son directivas que alteran este avance lineal. Los ciclos, específicamente, instruyen al CPU para que, tras completar un bloque de código, reposicione su puntero de instrucción a una ubicación anterior, reiniciando así la ejecución de dicho bloque.

---

2. El Ciclo `while`: Iteración Condicional

El ciclo `while` es la estructura de iteración más fundamental. Su operación está gobernada por una condición booleana. El bloque de código subordinado a `while` se ejecuta repetidamente mientras la evaluación de su condición resulte en `True`.

Sintaxis:

```
while condicion:
    # Bloque de código a ejecutar.
    # Debe contener lógica que eventualmente
    # altere el resultado de la 'condicion'.
```

Ejemplo: Algoritmo de cuenta regresiva

```
n = 5
while n > 0:
    print(n)
    n = n - 1 # Actualización de la variable de control

print("Terminado.")
```

Análisis de la ejecución:

1. **Inicialización:** Se inicializa una variable de control, `n`, en 5.
2. **Evaluación:** Se evalúa la condición `n > 0`. Inicialmente `5 > 0` es `True`, por lo que se accede al bloque.
3. **Ejecución:** Se ejecuta el cuerpo del ciclo. El valor de `n` se imprime y posteriormente se decrementa.
4. **Iteración:** El control retorna al encabezado `while` para una nueva evaluación de la condición.

5. **Terminación:** El proceso se repite hasta que `n` es 0. En ese punto, la condición `0 > 0` se evalúa como `False`, el ciclo termina y la ejecución continúa en la primera instrucción posterior al bloque `while`.

Error común: Ciclos infinitos La omisión de la actualización de la variable de control (`n = n - 1`) resultaría en un ciclo infinito, un error lógico donde la condición de terminación nunca se alcanza. Es imperativo que la lógica interna del ciclo garantice la eventual falsedad de la condición.

---

3. El Ciclo `for`: Iteración sobre Secuencias

Mientras `while` itera basado en una condición, el ciclo `for` está diseñado para iterar sobre los elementos de un objeto iterable (como una `list` o `tuple`). Es la construcción idiomática en Python para procesar cada elemento de una colección.

Sintaxis:

```
for variable_iteracion in iterable:
    # Bloque de código que opera sobre 'variable_iteracion'.
```

Ejemplo: Procesamiento de una lista de datos

```
datos = [10, 20, 30, 40, 50]

for elemento in datos:
    # 'elemento' toma el valor de cada ítem de 'datos' secuencialmente.
    print(f"Procesando: {elemento}")

print("Procesamiento completado.")
```

El ciclo `for` gestiona automáticamente el avance sobre la colección y la terminación cuando se han agotado los elementos, eliminando el riesgo de ciclos infinitos.

La Función `range()`: Generación de Secuencias Numéricas

Para iteraciones que no se basan en una colección preexistente, sino en un número definido de repeticiones, se utiliza la función `range()`. Este objeto genera una secuencia inmutable de números enteros de forma eficiente.

- `range(stop)`: Genera enteros desde `0` hasta `stop - 1`.
- `range(start, stop)`: Genera enteros desde `start` hasta `stop - 1`.
- `range(start, stop, step)`: Genera enteros desde `start` hasta `stop - 1`, con un incremento de `step`.

Ejemplo: Tabulación de una función

```
# Calcular y^2 para y en [0, 5)
for y in range(5):
    resultado = y ** 2
    print(f"y = {y}, y^2 = {resultado}")
```

---

4. Patrón de Iteración: Acumuladores

Un patrón algorítmico recurrente es el del **acumulador**. Consiste en una variable inicializada antes de un ciclo que se actualiza en cada iteración para computar un resultado agregado.

Estructura del patrón:

1. **Inicialización:** Declarar y asignar un valor inicial a la variable acumuladora.
2. **Actualización:** Dentro del ciclo, modificar el valor del acumulador basándose en el elemento actual de la iteración.
3. **Resultado:** Una vez finalizado el ciclo, el acumulador contiene el resultado final.

Ejemplo: Cálculo de una sumatoria (reimplementación de `sum()`)

$$S = \sum_{i=1}^{n} x_i$$

```
valores = [15.5, 8.2, 4.0, 12.3]
sumatoria = 0.0 # 1. Inicialización

for x in valores:
    sumatoria += x # 2. Actualización (sintaxis compacta para sumatoria = sumatoria + x)

# 3. Resultado
print(f"La sumatoria de los valores es: {sumatoria}")
```

Este patrón es extensible a otras operaciones, como el cálculo de productos (productoria), la concatenación de cadenas o el conteo condicional de elementos.

---

5. Terminación Prematura de Ciclos y Búsqueda Secuencial

Es posible terminar un ciclo antes de que su condición de finalización natural se cumpla. La sentencia `break` causa una salida inmediata del ciclo más interno en el que se encuentra.

Una aplicación directa de `break` es en algoritmos de búsqueda. La **búsqueda secuencial** es un método para localizar un elemento en una lista, que consiste en examinar cada elemento en orden hasta encontrar una coincidencia o agotar la lista.

Implementación del algoritmo:

```
catalogo_productos = [101, 234, 456, 198, 502]
producto_buscado = 198

encontrado = False # Variable de estado o "bandera" (flag)

for producto in catalogo_productos:
    if producto == producto_buscado:
        encontrado = True
        break # Terminar el ciclo; la búsqueda fue exitosa.

if encontrado:
    print(f"El producto {producto_buscado} existe en el catálogo.")
```

```
else:
    print(f"El producto {producto_buscado} no fue encontrado.")
```

El uso de `break` optimiza el proceso al evitar iteraciones innecesarias una vez que se ha alcanzado el objetivo de la búsqueda.

Junto a `break` está la palabra clave `continue` que omite la ejecución de las líneas siguientes en el bloque pero no detiene el ciclo.

---

6. ¿Cuán eficiente computacionalmente es esto?

La eficiencia de un algoritmo es una métrica fundamental en ciencias de la computación. Se puede obtener una noción intuitiva de esta eficiencia analizando el número de operaciones críticas que realiza. En la búsqueda secuencial, la operación crítica es la comparación (`==`).

Consideremos una lista de tamaño N.

- **Mejor caso:** El elemento buscado es el primero de la lista. Se realiza **1** comparación.
- **Peor caso:** El elemento buscado es el último, o no se encuentra en la lista. Se realizan **N** comparaciones.

La conclusión principal es que el tiempo de ejecución de la búsqueda secuencial en el peor caso es directamente proporcional al tamaño de la lista. Esta relación se describe como **coste lineal**. A medida que N aumenta, el tiempo de ejecución aumenta linealmente. Este concepto de escalabilidad es crucial para el diseño de software eficiente.

Resumen

En esta sesión se han presentado los conceptos fundamentales del control de flujo iterativo en programación:

Estructuras de control:

- **Ciclo `while`**: Para iteración condicional basada en una expresión booleana, con énfasis en la importancia de actualizar la variable de control para evitar ciclos infinitos.
- **Ciclo `for`**: Para iteración sobre secuencias y colecciones, incluyendo el uso de la función `range()` para generar secuencias numéricas.

Patrones algorítmicos:

- **Patrón acumulador**: Técnica para computar resultados agregados mediante una variable que se actualiza en cada iteración.
- **Búsqueda secuencial**: Algoritmo para localizar elementos en listas, con implementación usando la sentencia `break` para optimización.

Conceptos de eficiencia:

- **Análisis de coste**: Introducción al análisis de la complejidad computacional, específicamente el coste lineal de la búsqueda secuencial.