Unidad 3

Operadores y expresiones

Así como existen los operadores aritméticos para realizar operaciones con valores numéricos, y los operadores de texto para operar sobre cadenas de texto, los operadores **relacionales** se utilizan para comparar la relación entre dos valores. Por ejemplo, comparar si dos valores son iguales, o si uno es mayor que el otro.

Operador	Descripción	Ejemplo
==	Igual	a == b → compara si el valor *a* es igual al valor *b*
!=	no igual	a != b → compara si el valor *a* es distinto de *b*
>	mayor que	a > b → compara si *a* es mayor que *b*
<	menor que	a < b → compara si *a* es menor que *b*
>=	mayor o igual	a >= b → compara si *a* es mayor o igual que *b*
<=	menor o igual	a <= b → compara si *a* es menor o igual que *b*

Los operadores relacionales devuelven como resultado un valor booleano, *True* o *False*.

Una **expresión** es el resultado de uno o más operadores lógicos. Por ejemplo, la expresión `a == 5` verifica si el número *a* (que puede ser un número que haya ingresado el usuario, por ejemplo) es igual a 5. Dicha expresión valdrá True o False dependiendo el caso.

En una misma expresión se pueden combinar varios operadores relacionales. Por ejemplo,

`(a >= 10) and (a % 2 == 0)`. Esta expresión valida si el número *a* es mayor o igual que 10, y además verifica si es par (utilizando el operador de módulo/resto). Solo si se cumplen ambas condiciones la expresión valdrá True.

Para combinar operaciones en una misma expresión, tenemos disponibles los operadores booleanos **and**, **or** y **not**.

Los operadores lógicos OR y AND son asociativos. El primero se evalúa de izquierda a derecha, deteniéndose al encontrar un True, y devuelve False si todos son False. De modo similar, AND se detiene al encontrar un False, y devuelve True si todos son True.

Estructuras de control selectivas

Los programas operan con diversos datos de entrada, lo que a veces requiere tratar ciertos datos de manera diferente. Para hacer esto, necesitamos identificar los casos posibles y, dependiendo del caso, optar por diferentes rutas. En consecuencia, los programas deben ser capaces de hacerse preguntas sobre los datos y determinar el tratamiento adecuado.

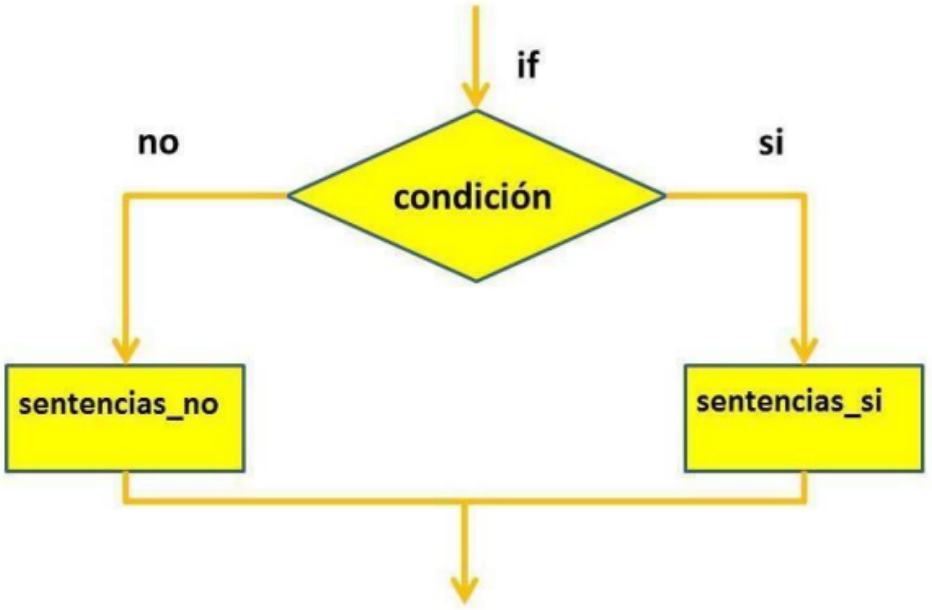
Para hacer preguntas y responder, utilizamos la lógica simbólica o bivalente. Esto implica asignar un valor de verdad a las expresiones lógicas. Los valores posibles son dos: **verdadero** y **falso**. Estos valores son disjuntos y complementarios; es decir, si algo no es verdadero, entonces es falso y viceversa, y algo no puede ser verdadero y falso al mismo tiempo.

El nombre teórico de una expresión lógica es proposición. En programación también se les llama **expresiones booleanas**.

Sentencia if

Todo lenguaje de programación procedural posee herramientas para trabajar con lógica y condicionales. Este tipo de herramientas se denominan sentencias condicionales, y permiten alterar el flujo de control del programa.

Cuando el flujo de control del programa alcanza una sentencia condicional, primero evaluará la condición asociada a ella. Si la condición es verdadera, entonces ejecutará las sentencias asociadas a esa elección, e ignorará las sentencias asociadas al caso de que la condición sea falsa.



Python dispone de la sentencia if, que permite tomar decisiones y hacer bifurcaciones en la ejecución del programa. La estructura básica de la sentencia if es simple. Se inicia con la palabra clave 'if' seguida de la expresión que queremos evaluar y dos puntos. A continuación, se escribe el bloque de código que se ejecutará si la expresión es verdadera.

Por ejemplo:

```
if x > 10:
        print("x es mayor que 10")
else:
        print("x no es mayor que 10")
```

En este caso, si la variable x es mayor que 10, la consola imprimirá el mensaje "x es mayor que 10". En caso contrario, imprimirá “x no es mayor que 10”.

💡 La sentencia if es una **sentencia estructurada**, que contiene un encabezado (en este caso, la condición) y un cuerpo (el código que se ejecuta cuando se cumple). El cuerpo se debe escribir con indentación, esta es la forma en la que Python identifica qué sentencias pertenecen a él.

Estructuras de control iterativas

En algunos programas, es necesario ejecutar las mismas instrucciones varias veces, pero con diferentes datos. En lugar de duplicar las instrucciones manualmente (una cantidad que podría cambiar o incluso ser desconocida), podemos usar **estructuras de control iterativas**, también conocidas como **ciclos** o **bucles**.

Un bucle es una herramienta que repite ciertos pasos en un programa. Si necesitamos repetir una parte de nuestro programa, podemos colocarla dentro del bucle, que realizará la repetición de manera automática hasta terminar.

Un bucle o ciclo está compuesto principalmente por una **condición** que decidirá si el cuerpo del bucle se ejecuta o no, y un **cuerpo**, es decir la lista de sentencias que se ejecutará de forma repetida.

Cuando el flujo de control del programa encuentra un ciclo, lo primero que hace es evaluar la condición, si esta resulta verdadera, ejecuta el cuerpo, y al terminar de ejecutarse, vuelve a evaluar la condición. Si vuelve a dar verdadero, se ejecuta el cuerpo, y así sucesivamente mientras la condición sea verdadera. Cuando la condición sea falsa, sale del ciclo y continúa ejecutando el resto del programa.

Ciclo while

El ciclo `while` permite ejecutar una serie de sentencias mientras la condición sea verdadera. Se escribe de la siguiente manera:

```
while condicion:
        instruccion_1
        instruccion_2
        ...
```

💡 Al igual que con la sentencia if, el cuerpo se escribe con indentación.

Dado que el ciclo por sí solo no tiene un mecanismo para contar las iteraciones, necesitaremos, en algún momento, hacer que la condición sea falsa, o de otro modo se generará un ciclo infinito.

Ejemplo: programa para imprimir los números del 1 al 10

```
cont = 1
while cont <= 10:
        print(cont)
        cont = cont + 1 # Actualizar la variable para que el
```

Ciclo for

El ciclo `for` permite ejecutar una serie de sentencias mientras la condición sea verdadera. A diferencia del ciclo `while`, sí tiene un mecanismo para contar las iteraciones y actualizar dicho contador al finalizar cada iteración. Esto lo hace útil para iterar sobre conjuntos de datos.

```
# Imprimir los números del 1 al 10
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numeros:
        print(num)

# Para no definir manualmente el conjunto podemos usar la fun
for num in range(1, 11):
        print(num)
```

Por lo general, el bucle **for** resulta más útil cuando conocemos el número de veces que necesitamos repetir un proceso, ya que nos evita el tener que controlar manualmente las iteraciones. Por otro lado, cuando desconocemos el número de iteraciones y necesitamos continuar hasta que se cumpla una condición (como por ejemplo, si solicitamos al usuario que introduzca un número par y queremos repetir la petición en caso de que no sea par), es más conveniente usar **while**.

Instrucciones break y continue

La instrucción `break` permite salir inmediatamente de un bucle antes de que se complete su iteración. Al encontrarse con un break, el programa salta fuera del bucle y continuará con las instrucciones que estén después del bucle.

La instrucción `continue` sirve para saltarse una iteración del bucle y continuar a la siguiente. El programa se salta el resto de código del bucle, y vuelve al inicio, donde verificará la condición y volverá a ejecutar el bucle si corresponde.