El futuro digital es de todos

MinTIC

Lógica de programación - condicionales

OPERADO POR:

Universidad Tecnológica de Pereira

Mision TIC 2022

RUTA DE APRENDIZAJE 1

Lógica de programación - condicionales

En programación es necesario utilizar diferentes conceptos para asegurarnos de que la maquina entienda que procesos debe realizar.

Dentro de la lógica de programación tenemos también que utilizar o traducir las condiciones de nuestro problema o sus limitaciones. Para eso podemos hacer uso de los condicionales.

Los condicionales o el uso de condicionales es uno de los elementos mas importantes de nuestro código, con esto describimos fenómenos de nuestro problema que solo aparecen bajo ciertas características.

Por ejemplo, podemos indicarle a la maquina que realice un procedimiento o un calculo, únicamente si se cumplen ciertas condiciones.

Con esto es posible limitar nuestro código y realizar solo las tareas necesarias

Para comprender mejor como manejar esta nueva lógica en programación, vamos revisar un concepto que nos ayudara en el proceso.

Tabla de verdad:

Las tablas de verdad o tabla de valores de verdad, son tablas que muestran el valor de verdad de una proposición compuesta, para cada combinación de valores de verdad que se pueda asignar a sus componentes.

La tabla de los "valores de verdad", es usada en la lógica de programación, para obtener la verdad (V) o falsedad (F). Además sirven para determinar si un determinado esquema es válido como un argumento, llegando a la conclusión de si es verdadero o falso.

Qué son las tablas de la verdad:

Las tablas de verdad son, por una parte, uno de los métodos más sencillos y conocidos de la lógica, pero al mismo tiempo también uno de los más poderosos. Entender bien las tablas de verdad es, en gran medida, entender bien a la lógica misma.

Fundamentalmente, una tabla de verdad es un dispositivo para demostrar ciertas propiedades lógicas y semánticas de enunciados del lenguaje natural o de fórmulas del lenguaje del cálculo:

•Cuáles son sus condiciones de verdad
•Cuál es su rol inferencial, es decir, cuáles son sus conclusiones lógicas y de qué otras proposiciones se siguen lógicamente.

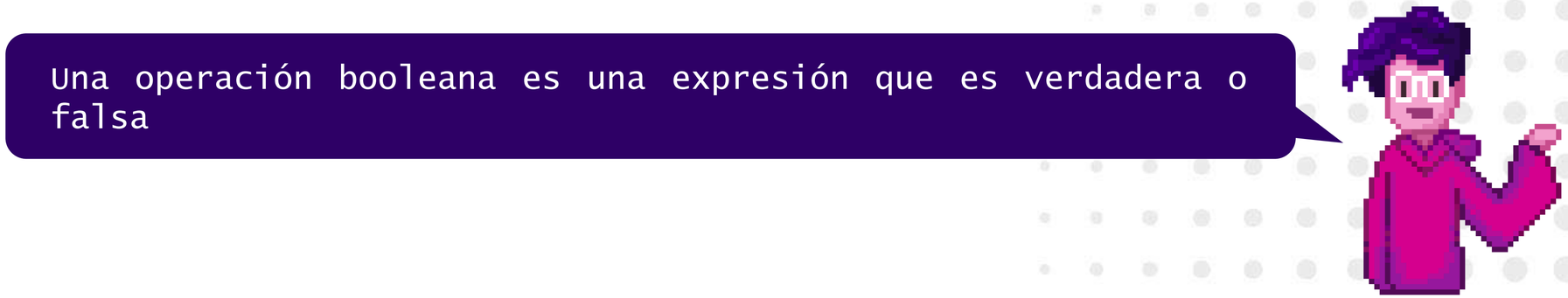
Finalmente, debemos recalcar que los únicos dos valores de verdad son **verdadero y falso**, independientemente de cómo se representen. Por ejemplo, es usual que estos valores se representen con expresiones como V y F, True y False, T y F o incluso números como 1 y 0. Sin embargo, en todos estos casos estamos hablando de los valores de verdad y no de expresiones de tipo numérico o de cadenas de caracteres.

Recordemos que:

•**Negación**. La negación (¬, not, no) es la operación Booleana que toma un valor de verdad y lo convierte en el otro valor.

•**Conjunción**. La conjunción (∧, and, y) es una operación Booleana binaria que tiene valor verdadero sólo cuando ambos operandos tienen valor verdadero.

•**Disyunción**. La disyunción (∨, or, o) es una operación Booleana binaria que tiene valor verdadero cuando por lo menos uno de los operandos tiene valor verdadero.



Consideremos la tabla más sencilla posible, en la cual sólo tenemos la proposición P y donde al lado derecho tengamos la negación de P.

P	¬ P
T	F
F	T

Aunque sencilla, esta tabla es interesante porque nos muestra todos los posibles valores de la expresión P y los valores correspondientes que tendría la expresión ¬ P (negación de P). También nos muestra que las dos expresiones no pueden ser verdaderas simultáneamente, sino que tienen valores opuestos.

Si extendemos la tabla con un par de expresiones encontraremos dos principios muy importantes:

P	¬ P	P ∧ ¬ P	P ∨ ¬ P
T	F	F	T
F	T	F	T

•Primero, vemos que la expresión P ∧ ¬ P siempre es falsa. En este caso estamos diciendo que, si P y el negado de P son verdaderas la respuesta es verdadero, sin embargo si P y el negado de P son falsas, entonces el resultado es falso. Si ambos valores son diferentes entonces siempre será falso. Esto quiere decir que no es posible que una proposición sea simultáneamente verdadera y falsa, lo cual usualmente se conoce como el principio de no contradicción.

•Segundo, vemos que la expresión P ∨ ¬ P siempre es verdadero. En esta caso la solución e siempre verdadera, excluyendo el caso de que ambos valores sean falsos, donde la solución seria falso. Esto quiere decir que siempre bien sea una proposición o su negación tienen que ser verdaderas. Como no hay una tercera opción, eso se conoce como el principio del tercero excluido.

Podemos ver este comportamiento si realizamos mas análisis entro los valores de dos proposiciones: P y Q.

P	Q	P∧Q	P∨Q
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Aquí podemos ver, de otra manera, lo que ya sabíamos sobre la conjunción y la disyunción:

- Una conjunción es verdadera sólo cuando los dos operandos son verdaderos
- Una disyunción es falsa sólo cuando los dos operandos son falsos.

Leyes fundamentales

Así como en el álgebra elemental hay algunas leyes muy importantes y conocidas (conmutatividad, asociatividad, distribución, precedencia de operadores, etc.), en el álgebra Booleana. También hay algunas leyes importantes que se deben tener en cuenta y pueden servir para replantear expresiones de formas que sean más sencillas.

•**Conmutatividad**: Tanto la conjunción como la disyunción son **conmutativas**, así que A ∨ B es equivalente a B ∨ A. También es cierto que A ∧ B es equivalente a B ∧ A.

•**Asociatividad**: Tanto la conjunción como la disyunción son **asociativas**, así que A ∨ (B ∨ C) es equivalente a (A ∨ B) ∨ C. Además, A ∧ (B ∧ C) es equivalente a (A ∧ B) ∧ C.

•**Distribución**: en el álgebra Booleana, la conjunción distribuye sobre la disyunción y viceversa. Esto quiere decir que:

- A ∧ (B ∨ C) ≡ (A ∧ B) ∨ (A ∧ C)
- A ∨ (B ∧ C) ≡ (A ∨ B) ∧ (A ∨ C)

- **Identidad de la conjunción**: si se hace una conjunción con el valor verdadero, el resultado es el mismo. Es decir que A ∧ Verdadero ≡ A.

- **Identidad de la disyunción**: si se hace una disyunción con el valor falso, el resultado es el mismo. Es decir que A ∨ Falso ≡ A.

- **Dominación de la conjunción**: si se hace una conjunción con el valor falso, el resultado es falso. Es decir que A ∧ Falso ≡ Falso.

- **Dominación de la disyunción**: si se hace una disyunción con el valor verdadero, el resultado es verdadero. Es decir que A ∨ Verdadero ≡ Verdadero.

- **Negación y equivalencia a falso**: es lo mismo decir que una proposición tiene un valor falso que decir que su negación tiene un valor positivo. Es decir que A ≡ Falso es lo mismo que decir ¬ A ≡ Verdadero.

Más aún, B ≡ Verdadero es lo mismo que decir B, así que A ≡ Falso se puede reescribir como ¬ A.

Leyes de de Morgan

Ahora veremos dos teoremas muy importantes que se deben tener muy en cuenta cuando se estén reescribiendo operaciones lógicas.

Las leyes o teoremas de Morgan dicen que:

- $\neg (P \wedge Q) \equiv \neg P \vee \neg Q$
- $\neg (P \vee Q) \equiv \neg P \wedge \neg Q$

Volviendo a las proposiciones de ejemplo que usamos anteriormente, la expresión $\neg (P \wedge Q)$ podría “traducirse” como “no es cierto que (hoy está haciendo frio y hoy es lunes)”. Hemos agregado los paréntesis para hacer notar que las palabras “no es cierto que” hacen referencia a las dos cláusulas subordinadas.

Sabemos que la expresión completa será verdadera sólo cuando la parte que está entre paréntesis sea falsa para que la negación invierta su valor. Es decir que la expresión completa será verdadera cuando sea falso que “hoy está haciendo frio y hoy es lunes”.

Ahora que tenemos mas claridad con las tablas de verdad, vamos a revisar como manejar esta logia en programación:

Operadores lógicos:

Para crear condiciones, por muy simples que estas sean, se necesitan los operadores lógicos. Las tablas de la verdad nos ayudan a identificar mejor algunos casos y sus particularidades, sin embargo los operadores lógicos nos permiten indicarle al código con que tipo de condición deseamos trabajar y como escribirla:

== significa “igual”. If x==y, significa “si x es igual a y”

> significa “mayor que”. If x>y, significa “si x es mayor que y”

< significa “menor que”. If x<y, significa “si x es menor que y”

!= significa “si es distinto”. If x!=y, significa “si x es distinto de y”

&& significa “Y”, la conjunción copulativa; es decir: If (x==y) && (x==z), significa “si x es igual a y Y x igual a z”

|| significa “O”, la conjunción adversativa; es decir, If (x==y) || (x==z), significa “si x es igual a y O x igual a z”

Con estos **operadores lógicos** podemos definir en conjunto con otras funciones mas complejas en programación, diferentes tareas o instrucciones que nos permitan facilitar la forma en la que le indicamos a la maquina como resolver nuestro problema.

Ejemplos:

Hay tres operadores lógicos: y (and), o (or), y no (not). La semántica (significado) de estos operadores es similar a su significado en inglés.

```
x = #escribe algo acá

x > 0 and x < 10
```

Vamos a definir que esto es cierto sólo si x es mayor que 0 y menor que 10.

```
n%2 == 0 or n%3 == 0
```

Es cierto si cualquiera de las condiciones es verdadera, es decir, si el número es divisible entre 2 o 3. Finalmente, el operador **not** niega una expresión booleana, por lo que no **(x > y)** es verdadero si x > y es falso; es decir, si x es menor o igual que y.

En sentido estricto, los operandos de los operadores lógicos deberían ser expresiones booleanas, pero Python no es muy estricto. Cualquier número que no sea cero se interpreta como "verdadero".

```
17 and True
```

Esta flexibilidad puede ser útil, pero hay algunas sutilezas en ella que pueden ser confusas. Puede que quieras evitarlo hasta que estés seguro de lo que estás haciendo.

Evaluación de las expresiones lógicas

Cuando Python está procesando una expresión lógica como **x >= 2 and (x/y) > 2**, se evalúa la expresión de izquierda a derecha. Debido a la definición de y, si x es menos de 2, la expresión x >= 2 es Falsa y por lo tanto toda la expresión es Falsa sin importar si (x/y) > 2 evalúa a Verdadero o Falso.

Cuando Python detecta que no hay nada que ganar evaluando el resto de una expresión lógica, detiene su evaluación y no hace los cálculos en el resto de la expresión lógica.

Cuando la evaluación de una expresión lógica se detiene porque el valor global ya se conoce, se llama cortocircuito de la evaluación.

Mientras que esto puede parecer un buen punto, el comportamiento de cortocircuito lleva a una inteligente técnica llamada el patrón del guardián. Considere la siguiente secuencia de códigos en el Intérprete de Python

In [1]:
```
x = 6
y = 2

x >= 2 and (x/y) > 2
```

Out[1]: True

In [2]:
```
x = 1
y = 0
x >= 2 and (x/y) > 2
```

Out[2]: False

In [3]:
```
x = 6
y = 0
x >= 2 and (x/y) > 2
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-43a82c4c87c1> in <module>
      1 x = 6
      2 y = 0
----> 3 x >= 2 and (x/y) > 2

ZeroDivisionError: division by zero
```

El tercer cálculo falló porque Python estaba evaluando (x/y) y "y" era cero, lo que causa un error de tiempo de ejecución. Pero el segundo ejemplo no falló porque la primera parte de la expresión x >= 2 evaluada da Falso por lo que el (x/y) nunca se ejecutó y no hubo regla de cortocircuito (no hubo error).

Podemos construir la expresión lógica para colocar estratégicamente una evaluación del guardia justo antes de la evaluación que podría causar un error de la siguiente manera:

```
In [4]:
x = 1
y = 0
x >= 2 and y != 0 and (x/y) > 2
```

```
Out[4]: False
```

```
In [5]:
x = 6
y = 0
x >= 2 and y != 0 and (x/y) > 2
```

```
Out[5]: False
```

```
In [6]:
x >= 2 and (x/y) > 2 and y != 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-b3ac81785519> in <module>
----> 1 x >= 2 and (x/y) > 2 and y != 0

ZeroDivisionError: division by zero
```

En la primera expresión lógica, x >= 2 es Falso, por lo que la evaluación se detiene en el y. En la segunda expresión lógica, x >= 2 es Verdadero pero y != 0 es Falso por lo que nunca llegamos a (x/y).

En la tercera expresión lógica, la y != 0 está después del cálculo (x/y) por lo que la expresión falla con un error. En la segunda expresión, decimos que y != 0 actúa como un guardián para asegurar que sólo ejecutamos (x/y) si y no es cero

Nombre	Tipo	Descripción
id	int	El identificador de la presentacion cuyo tipo de chocolate se quiere calcular.

Tipo del retorno	Descipción del retorno
str	Si el número es Palíndromo e impar, la presentacion corresponde a un chocolate amargo, y se retorna "BITTER". Si el número es Palíndromo y par, la presentacion corresponde a un chocolate dulce, y se retorna "SWEET". Si el número es par, pero no palíndromo, la presentación corresponde a un chocolate con canela y clavos, y se retorna "CINNAMON". Si el número es impar, pero no palíndromo,la presentacion corresponde a un chocolate bajo en grasa, y se retorna "LIGHT"

In [7]:

```python
def palidromo(numero:int)->bool:
    centena=numero//100
    decena=(numero-centena*100)//10
    unidad=numero-centena*100-decena*10

    if centena == unidad:
        resultado = True
    else:
        resultado = False
    return resultado

def espar(numero:int)->bool:
    if numero % 2 == 0:
        resultado = True
    else:
        resultado = False
    return resultado

def clasificar_chocolate(codigo:int)->str:
    if palidromo(codigo):
        if espar(codigo):
            resultado = "SWEET"
        else:
            resultado = "BITTER"
    else:
        if espar(codigo):
            resultado = "CINNAMON"
        else:
            resultado = "LIGHT"
    return resultado

print(clasificar_chocolate(123))
print(clasificar_chocolate(222))
print(clasificar_chocolate(111))
print(clasificar_chocolate(505))
print(clasificar_chocolate(576))
```

```
LIGHT
SWEET
BITTER
BITTER
CINNAMON
```