Expresiones booleanas y estructuras de decisión

Los lenguajes de programación permiten comparar valores. Python sigue la sintaxis y semántica de la mayoría de los lenguajes imperativos.

Ejemplo 1

```
>>> x = 10
>>>
>>> x == 10
True
>>> x != 10
False
>>>
>>> x > 5
True
>>> x < 10
False
>>>
>>> x >= 5
True
>>> x <= 10
True
>>>
```

Podemos ver que las expresiones de comparación incluyen los **operadores lógicos** de igualdad (==), desigualdad (!=), mayor que (>), menor que (<), mayor o igual que (>=) y menor o igual que (<=). También podemos notar en el ejemplo que estos operadores son **binarios** y que funcionan con operandos constantes y variables. En general, los operandos pueden ser **cualquier expresión válida** en Python.

Ejemplo 2

```
>>> 3 + 9 > 2.0 + 9.0
True
>>> 3 + 9 <= 2.0 + 9.0
False
>>> 3 + 9 == 3.0 + 9.0
True
>>>
```

¡Qué brillante es el intérprete de Python! Sabe que 12 es mayor que 11.0 y que 12 es igual a 12.0. Notemos dos cosas. Primero, que la aritmética se resuelve antes que la

comparación. Es decir, los operadores aritméticos tienen **mayor precedencia** que los operadores de comparación. Segundo, que el tipo de dato de los operandos numéricos (entero o flotante) **no afecta** las comparaciones. Esto porque ambos operandos se transforman a flotantes, tal como vimos que ocurría con los operadores aritméticos.

Las expresiones como `3 + 9 > 2.0 + 9.0`, se conocen como **expresiones booleanas** y arrojan como resultado uno de **dos posibles valores**: `True` (verdadero) o `False` (falso), conocidos como **valores booleanos**, en honor al matemático y lógico británico George Boole, quien formalizó el álgebra que gobierna estos valores (que en Álgebra I podríamos haber conocido como Álgebra de Boole, Lógica Proposicional o Lógica Matemática).

George Boole (1815 – 1864)

George Boole fue un matemático británico, procedente de una familia de escasos recursos, al punto que tuvo que dejar sus ideas de convertirse en monje al verse obligado a mantener a sus padres.

A la corta edad de dieciséis años enseñaba matemáticas en un colegio privado y más tarde fundo uno propio, a los veinticuatro años, tras publicar su primer escrito se le ofreció una vacante en la universidad de Cambrige, oferta que desestimó para cuidar de su familia. El resto de su vida trabajó como profesor de matemáticas del Queen's College, en Cork, donde permaneció el resto de su vida.

El aporte de Boole fue aplicar una serie de símbolos a operaciones lógicas y hacer que estos símbolos y operaciones, tuvieran la misma estructura lógica que el álgebra convencional. En el **Álgebra de Boole,** los símbolos podían manipularse según reglas fijas, que producían resultados lógicos, cuyos argumentos admiten sólo dos estados finales (verdadero o falso)

Si bien, en un principio la adopción del aporte de Boole al álgebra fue lenta, este aporte marca los fundamentos de la aritmética computacional moderna, lo que hace a George Boole uno de los fundadores del campo de las Ciencias de la Computación.

Recordemos que el Álgebra de Boole permite combinar resultados de expresiones de comparación, esto se realiza a través de los operadores de conjunción (`and`), disyunción (`or`) y negación (`not`).

Ejemplo 3

```
>>> 10 < 50 and 50 > 100 or not 5 > 10 and 20 == 20.0
True
>>>
```

Analizando el ejemplo 3, nos daremos cuenta que la evaluación de la expresión siguió el siguiente orden:

```
10 < 50 and 50 > 100 or not 5 > 10 and 20 == 20.0
 \   /
  True and 50 > 100 or not 5 > 10 and 20 == 20.0
            \     /
    True and False or not 5 > 10 and 20 == 20.0
        \         /
         False or not 5 > 10 and 20 == 20.0
                   \   /
          False or not False and 20 == 20.0
                     |
           False or True and 20 == 20.0
                         \     /
             False or True and True
                       \     /
                 False or True
                    \    /
                     True
```

En otras palabras, la expresión es equivalente a `((10 < 50) and (50 > 100)) or ((not (5 > 10)) and (20 == 20))`, y no, por ejemplo, a la expresión `((10 < 50) and (50 > 100)) or (not ((5 > 10) and (20 == 20)))`. Esto ocurre porque, al igual que los operadores aritméticos, los operadores lógicos responden a una **precedencia** establecida: primero se resuelven los operadores de comparación, luego las negaciones (que, a diferencia de los otros, es un **operador unario**), luego las conjunciones, y finalmente las disyunciones.

Pregunta 1

Ahora desarrolle la pregunta 1 indicada por su profesor(a)

¿Para qué sirven las expresiones booleanas? Para responder esta pregunta, debemos fijarnos en la estructura de bloque principal de los programas que hemos hecho hasta ahora. Esta estructura ha sido **siempre lineal**, es decir las sentencias se ejecutan una a una en el **orden que aparecen**, comenzando por la sentencia 1, luego la sentencia 2, y así sucesivamente todas las sentencias hasta completar el programa.

La utilidad de las expresiones booleanas se hace evidente cuando un programa

las usa para **tomar decisiones**. Los humanos tomamos decisiones continuamente, por ejemplo, ¿Asisto a clases o me quedo estudiando en la biblioteca? ¿Sirvo té o café? ¿Este estudiante aprobó o reprobó el curso? Como la computación se usa para **resolver problemas de la vida real**, necesitamos dotar a un programa con la capacidad de **decidir entre caminos de solución distintos**, y para esto utilizamos expresiones booleanas

En general, los lenguajes de programación proveen diferentes **estructuras de decisión**. La más simple se compone de una **condición**, que no es otra cosa que una expresión booleana, y un **bloque de sentencias condicional** que **sólo se ejecutan** si la condición se evalúa con valor **verdadero**. La figura 1 muestra esta estructura. Notemos que **después** de la construcción de decisión simple vienen otras sentencias, las que **siempre toman el flujo de control** después de la decisión.

Figura 1: Diagrama de flujo de la ejecución de una estructura de decisión

La sintaxis de Python para una estructura de decisión simple es la siguiente:

Importante

```
if <condición>:
        <Bloque de sentencias condicionales>
<Bloque de sentencias que sigue>
```

La estructura de decisión simple aparece muy frecuentemente en la construcción de programas. Por ejemplo, supongamos que necesitamos la función `logz()` definida para todos los enteros:

$$\text{logz}(x) = \begin{cases} \ln(-x) & \text{si } x < 0 \\ \ln(1) & \text{si } x = 0 \\ \ln(x) & \text{si } x > 0 \end{cases}$$

Una posible solución sería: si el valor de x es negativo, entonces lo convertimos en positivo multiplicando por -1 y damos un mensaje que informe de este cambio; si x es cero, entonces lo cambiamos por el valor 1 y damos un mensaje avisando de este cambio; finalmente entregamos el valor de ln(x). Esta solución puede verse escrita en Python en el programa `Logz.py`. Probando el programa, se tiene lo siguiente:

```
>>> ========================== RESTART ============================
>>>
Ingrese un número entero: 0
Valor 0 (cero) cambiado a 1
log( 0 ) = 0.0
>>> ========================== RESTART ============================
>>>
Ingrese un número entero: -100
Valor -100 (negativo) cambiado a 100
log( -100 ) = 4.60517018599
>>> ========================== RESTART ============================
>>>
Ingrese un número entero: 10
log( 10 ) = 2.30258509299
>>>
```

Analicemos el programa paso a paso, ignorando los comentarios. La primera sentencia (línea 9) es la importación de la función logaritmo natural desde el módulo `math`. La siguiente sentencia define la función `logz()`, con un parámetro formal llamado `x` (línea 15). El cuerpo de esta función está compuesto por dos decisiones simples y una sentencia de retorno. La primera decisión simple (líneas 16-18) usa la condición `x < 0`. Si esta expresión booleana se evalúa como verdadera, entonces se ejecuta el bloque de sentencias condicionales (líneas 17-18) que muestra un mensaje en pantalla avisando que el valor negativo `x` se ha reemplazando por el valor positivo `-x` (línea 17) y luego realiza este cambio (sentencia de asignación de la línea 18).

Importante

La función `logz(x)` del ejemplo, despliega mensajes en pantalla a través de la sentencia `print`, **pero esto no es una buena práctica de programación,** en general necesitamos que las funciones sólo retornen un valor de tal forma que puedan ser usadas como parte de expresiones aritméticas, como por ejemplo:

```
resultado = logz(x) + 2*logz(x)
```

Si la función pide datos o despliega mensajes en pantalla, confunde el cálculo de esta expresión.

Logz.py

```
1. # -*- coding: cp1252 -*-
2.
3. #
4. # Programa que implementa logz():
5. # logaritmo natural definido para todos los enteros,
6. # haciendo logz(x) = ln(-x) para x < 0, y logz(0) = ln(1)
7. #
8.
9. from math import log
10.
11.#
12.# Funciones
13.#
14.
15.def logz(x):
16.    if x < 0:
17.        print "Valor", x, "(negativo) cambiado a", -x
18.        x = -x
19.    if x == 0:
20.        print "Valor", x, "(cero) cambiado a", 1
21.        x = 1
22.    return log(x)
23.
24.#
25.# Bloque principal
26.#
27.
28.# Entrada de datos
29.x = input("Ingrese un número entero: ")
30.
31.# Procesamiento
32.logx = logz(x)
33.
34.# Salida de datos
35.print "log(",x, ") =", logx
```

En este punto es importante notar que se ejecute o no se ejecute el bloque condicional de las líneas 17-18, el **flujo del programa** sigue con la sentencia de la línea 19, que en este caso corresponde a otra estructura de decisión simple. Así, la ejecución de del bloque de sentencias 20-21 depende de si la condición `x == 0` es verdadera o no. Si la condición se cumple, estas sentencias despliegan un mensaje indicando el reemplazo del valor cero por el valor uno y el cambio de valor de la variable `x`.

Nuevamente, independientemente de si se han ejecutado las sentencias condicionales, el flujo del programa continúa en la línea 22, donde la función devuelve el logaritmo natural del valor almacenado en la variable `x`. Notemos que aquí el programa **no evalúa** si `x > 1`, puesto que esto ha sido asegurado por las decisiones simples anteriores y siempre se tendrá un valor entero igual o superior a uno.

Es importante notar que el código de las líneas 15-23 (incluyendo la línea en blanco sin indentar), solamente definen la función `logz()`. Todo el análisis anterior de cómo fluye el control al interior de la función, y qué se ejecuta en cada caso, es una **interpretación semántica** de la **sintaxis** que allí está escrita. Al momento de la ejecución, sólo algunas sentencias se ejecutan dependiendo del valor que trae el parámetro real asociado a la variable `x`.

Para que la función `logz()` se ejecute, debemos **invocarla** con algún parámetro formal. La primera sentencia del bloque principal (línea 29) obtiene desde el usuario un valor entero cualquiera y lo almacena en la **variable global** `x`. Recordemos que la función `input()` detiene el flujo hasta que el usuario entrega un valor y presiona la tecla <ENTRAR>.

La función `logz()` no es invocada hasta la línea 32, donde se le entrega como parámetro formal el valor contenido en la variable global `x`. El valor que devuelve la función se guarda en la variable global `logx`. Este valor es mostrado en pantalla, junto a un mensaje adecuado, en la última sentencia del programa (línea 35).

Notemos nuevamente la sintaxis de la decisión simple: parte con la palabra reservada `if`, seguida de una expresión booleana, seguida de dos puntos para marcar el fin de la condición y el aumento de la **indentación** en un nivel para indicar que la ejecución del bloque de sentencias se está **condicionando** a que se cumpla la condición. La condición puede ser una **expresión compleja**, que incluya los operadores `and`, `or` y `not` para unir comparaciones simples. Lo importante es que el resultado de esta expresión sea finalmente un valor booleano (verdadero o falso). Podemos pedir, por ejemplo, que el número sea mayor que 10 y menor que 50 (`x > 10 and x < 50`); incluso podemos verificar si el valor de la variable cumple alguna relación matemática menos explícita, como por ejemplo que 6 veces el valor de una variable sea menor que 200 (`x * 6 < 200`).

Pregunta 2

Ahora desarrolle la pregunta 2 indicada por su profesor(a)

Ahora pensemos en una situación donde sabemos qué se tiene hacer en ambos casos, es decir, si la condición **se cumple** queremos que el programa **decida un camino** y en caso de que **no se cumpla** se decida **otro camino distinto**. Podemos dar solución a esto con dos estructuras de decisión simples, una para la **condición original** y otra para la **condición complementaria**.

Ejemplo 4

```
if x > 0  :
    # Bloque de sentencias que se ejecutan si la condición se cumple
    …
if x <= 0 :
    # Bloque de sentencias que se ejecutan si la condición opuesta se cumple
    …
```

Sin embargo, esta solución no siempre es buena, pues a veces, sobre todo cuando la expresión booleana de la condición es compleja, resulta difícil establecer la condición opuesta. Un **error común** en el caso anterior, por ejemplo, es considerar que la condición opuesta a `x > 0` es `x < 0`. Por ello Python, y muchos otros lenguajes de programación, ofrece la **estructura de decisión alternativa** que permite la ejecución condicional de **dos bloques de código sujetos a la misma condición**, como se ilustra en la figura 2.

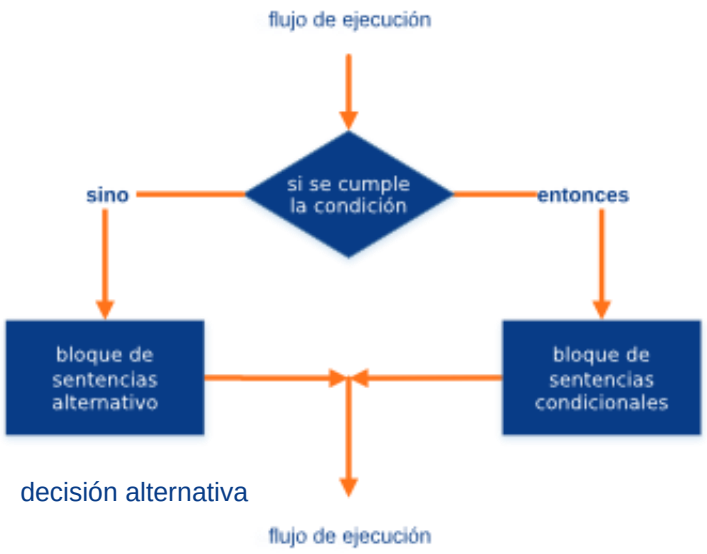

Figura 2: Diagrama de flujo de la ejecución de una estructura de

La sintaxis de Python para una estructura de decisión alternativa es la siguiente:

Importante

```
if <condición>:
        <Bloque de sentencias condicionadas a que la condición se cumple>
 else:
        <Bloque de sentencias alternativo (cuando la condición NO se cumple)>
<Bloque de sentencias que siguen>
```

Con esta construcción podemos, por ejemplo, implementar una función que nos diga si un número dado es par o impar.

MuestraParOImpar.py

```
1. # -*- coding: cp1252 -*-
2.
3. #
4. # Programa que indica por pantalla si un número ingresado
5. # por el usuario es par o impar.
6. #
7.
8. #
9. # Bloque principal
10.#
11.
12.# Entrada de datos
13.valor = input("Ingrese un número entero: ")
14.
15.# Salida de datos
16.if valor % 2 == 0:
17.    print "El valor", valor, "es par"
18.else:
19.    print "El valor", valor, "es impar"
```

Pregunta 3

Ahora desarrolle la pregunta 3 indicada por su profesor(a)