Expresiones lógicas

Juan Álvarez Rubio

Las condiciones (que incluyen operadores de relación o de comparación) producen como resultado un valor Verdadero o Falso. En el lenguaje Python las condiciones entregan un valor True o False, que son los únicos dos valores del tipo predefinido bool, llamado así como un homenaje a George Boole quien aportó a los fundamentos teóricos de la computación. En otros lenguajes este tipo de datos suele llamarse boolean o logical.

Tipo bool

El tipo bool es uno de los tipos predefinidos en el lenguaje Python y se utiliza para representar y operar con los valores True y False que representan la veracidad o falsedad de las proposiciones lógicas de las matemáticas. Otros tipos de datos en Python son int para números enteros, float para números reales y str para strings o textos literales. En cada uno de los tipos de datos se reconocen constantes, expresiones, variables y funciones.

Las constantes del tipo bool son True y False, que se deben escribir de esta única manera, es decir la primera letra en mayúscula y las siguientes en minúscula y no deben encerrarse entre apóstrofes ni entre comillas porque en ese caso se considerarían del tipo str (string).

Las expresiones del tipo bool corresponden a las condiciones que se admiten en la instrucción if para expresar el criterio para seleccionar instrucciones. Por ejemplo, a<b es una expresión de tipo bool que entrega el valor True si el valor de la variable a es menor que el valor de la variable b (y False en caso contrario).

El resultado de una expresión de tipo bool puede guardarse o asignarse a una variable para ser usado posteriormente. Por ejemplo, la instrucción p = a<b guarda el valor True o False del resultado de la condición en la variable p. Al asignarse un valor de tipo bool entonces la variable también adquiere el tipo bool y puede usarse en todo contexto en que se permite una condición, por ejemplo, if p: instrucciones1 else: instrucciones2.

Condiciones compuestas

Una condición compuesta consiste de dos o más condiciones simples separadas operadores lógicos. Por ejemplo, la función para determinar el tipo de un triángulo se puede escribir:

```
#tipo: num num num -> str
#'equilatero', 'isosceles' o 'escaleno'
#ej: tipo(1,1,1)->'equilatero'
#ej: tipo(2,2,3)->'isosceles'
#ej: tipo(3,4,5)->'escaleno'
def tipo(x,y,z):
    if x==y and x==z:
```

```
        return 'equilatero'
    elif x==y or x==z or y==z:
        return 'isosceles'
    else:
        return 'escaleno'
assert tipo(1,1,1)=='equilatero'
assert tipo(2,2,3)=='isosceles'
assert tipo(3,4,5)=='escaleno'
```

La primera condición establece que, si x es igual que y, y además si x es igual que z, entonces el triángulo es equilátero porque tienes los tres lados iguales. La condición después de elif sirve para verificar si el triángulo tiene dos lados iguales. La función está escrita usando elif de manera de expresar los tres resultados posibles a través de las tres instrucciones return.

La condición `x==y and x==z` responde a la forma general condición1 operador-lógico condición2. Los operadores lógicos son **and** y **or**. El resultado de una condición compuesta se explica con una "tabla de verdad" similar a la conjunción y disyunción de las proposiciones lógicas, es decir:

condición1	condición2	condición1 **and** condición2	condición1 **or** condición2
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

El operador lógico **and** entrega el valor True si las dos condiciones son también True. El operador lógico **or** produce un resultado True si alguna de las dos condiciones es True. Por otra parte, la evaluación de una condición compuesta se realiza de una manera eficiente. Por ejemplo, en el caso de **and**, si la condicion1 es False, entonces el resultado es False independientemente del valor de la condicion2. En el operador **or**, si la condición1 es True el resultado es True, sin que sea necesario evaluar la condición2. En resumen, la condición2 se evalúa sólo si se necesita para determinar el resultado de la condición compuesta.

El operador lógico not es un operador unario, es decir requiere un solo operando, y su forma general es not condición. La semántica indica que entrega False si la condición es True, y entrega True si la condición es False. El operador lógico not es poco utilizado con una condición simple, porque puede reescribirse afirmativamente. Por ejemplo, not x>y es equivalente a x<=y que es más legible.

Habitualmente, el operador not se utiliza con una condición compuesta, y en ese caso se cumplen las leyes de Morgan: not(condicion1 and condicion2) es equivalente a not(condicion1) or not(condicion2); y not(condicion1 or condicion2) es equivalente a not(condicion1) and not(condicion2). Por ejemplo, para verificar que un valor y no está entre x y z se puede escribir not(x<=y and y<=z) que es equivalente a not(x<=y) or not(y<=z) o mejor a x>y or y>z.

En las condiciones compuestas pueden aparecer operadores aritméticos, operadores de comparación y operadores lógicos. Para evaluar una condición compuesta se debe considerar que la mayor prioridad la tienen los operadores aritméticos (y entre ellos con las prioridades preestablecidas). En segundo lugar, los operadores de comparación. En tercer lugar, los operadores lógicos en el orden not, and y or. Por ejemplo, no necesita el uso de paréntesis la condición que entrega True si a está entre b y c: b<=a and a<=c or c<=a and a<=b no necesita el uso de paréntesis. Tampoco necesita paréntesis la condición para determinar que los números positivos a, b y c forman un triángulo: a+b>c and a+c>b and b+c>a.

Funciones lógicas

Es muy habitual escribir funciones lógicas, es decir, funciones que entregan un resultado True o False que indica si sus parámetros cumplen o no una cierta propiedad. Por ejemplo, una función que determina si un número entero es o no un número par, se puede escribir de la siguiente manera:

```
#par: int -> bool
#True si x es par (False si es impar)
#ej: par(4)->True, par(5)->False
def par(x):
  if x % 2 == 0:
    return True
  else:
    return False
assert par(4)==True
assert par(5)==False
```

Para determinar si el número es par se calcula el resto de la división por 2 y, si el resultado es cero, entonces es un número par, de lo contrario es impar. Como toda función que devuelve un resultado de tipo bool, la prueba de la función debe contemplar los dos casos posibles. Al respecto, el cuerpo y la prueba de la función se puede escribir de una manera más concisa:

```
def par(x):
  return x%2==0
assert par(4)
assert not par(5)
```

En este caso, la expresión x%2==0 produce un resultado de tipo bool, por lo tanto, la instrucción return devuelve el valor True o False, y no se necesita usar una instrucción if. Por otra parte, en la prueba con la instrucción assert, basta con invocar a la función con un número par y otro impar, y usar el resultado directamente sin necesidad de comparar con los valores True y False.

Una función lógica más interesante es una que determina si un año es o no bisiesto. Al respecto, un año es bisiesto si es divisible por 4 (por ejemplo 2020) y se exceptúan los años divisibles por 100 (por ejemplo 1900). Sin embargo, las reglas del calendario gregoriano establecen que los años

divisibles por 400 también son bisiestos (por ejemplo, el año 2000 fue bisiesto). Por lo tanto, una función para determinar si un año es bisiesto se puede escribir de la siguiente manera:

```
#bisiesto: int -> bool
#True si año es bisiesto (False si no)
#ej: bisiesto(2021) -> False (no es divisible por 4)
#ej: bisiesto(2020) -> True  (divisible por 4)
#ej: bisiesto(1900) -> False (divisible por 100)
#ej: bisiesto(2000) -> True  (divisible por 400)
def bisiesto(año):
  if año % 4   != 0: return False
  if año % 400 == 0: return True
  if año % 100 == 0: return False
  return True
assert not bisiesto(2021)
assert bisiesto(2020)
assert not bisiesto(1900)
assert bisiesto(2000)
```

La función se puede escribir también con una instrucción de selección múltiple con un if, dos elif y un else, pero como cada rama terminaría con un return, es más breve escribir 3 instrucciones if omitiendo la opción else. El orden entre las instrucciones if es muy importante, para no entregar un resultado erróneo. Por ejemplo, sería incorrecto preguntar primero si el año es divisible por 100 porque en ese caso el año 2000 no se consideraría bisiesto.

La prueba de la función debe contemplar todos los casos posibles, de manera de garantizar su correcto funcionamiento. Por otra parte, analizando las alternativas, se puede concluir que los 4 casos quedan contemplados en una sola expresión de tipo bool:

```
def bisiesto(año):
   return año%4 == 0 and año%100 != 0 or año%400 == 0
```

La expresión se puede escribir sin paréntesis debido a las reglas de prioridad entre los operadores: primero se evalúan los operadores aritméticos (%), segundo los operadores de comparación (== y !=), en tercer lugar, el operador lógico and y, finalmente, el operador lógico or. Al respecto, para enfatizar y confirmar el orden de evaluación de la expresión de tipo bool se podría escribir:

```
def bisiesto(año):
   return (((año%4) == 0) and ((año%100) != 0)) or ((año%400) == 0)
```

Se recomienda que el nombre de una función lógica sea un adjetivo de manera que exprese una propiedad de su(s) parámetro(s), por ejemplo, par o bisiesto. Nombres como numeroPar o añoBisiesto no se aconsejan porque son sustantivos que sugieren que la función entrega un número par o un año bisiesto. Alternativamente, se suele usar un nombre que insinúe la respuesta a una

pregunta. Por ejemplo, esPar, esBisiesto, esNumeroPar o esAñoBisiesto sugieren una respuesta True o False.

Instrucción assert

Una condición es una expresión de tipo bool. Por lo tanto, la sintaxis de la instrucción assert es assert condición. Su semántica establece que primero hay que evaluar la expresión. Si el resultado es True se pasa a la instrucción siguiente, y si es False el programa termina indicando un error ("AssertionError").

Un uso de la instrucción assert es la prueba o testing de una función. Por ejemplo, para calcular el porcentaje que representa un número respecto de otro se puede escribir la función:

```
#porcentaje: num num -> float
#porcentaje de x respecto de y
#ej: porcentaje(1,8)->12.5, porcentaje(1,3)->33.3...
def porcentaje(x,y):
    return x/y*100
assert porcentaje(1,8)==12.5
#prueba de un resultado real no exacto
#assert porcentaje(1,3)==33.3 produce AssertionError
assert round(porcentaje(1,3),1)==33.3
assert porcentaje(1,3)-33.3<0.1
assert abs(33.3-porcentaje(1,3))<0.1
```

En la prueba o testing de una función habitualmente se pregunta si el resultado es igual a un valor esperado, pero no siempre se puede garantizar un resultado exacto. Por ejemplo, el porcentaje de 1 respecto de 3 es el racional 33.3… con período 3. Por lo tanto, si se pregunta si el porcentaje es igual a 33.3 se produce un error de aserción. En este caso, se puede aproximar el resultado a un decimal o realizar la comparación porcentaje(1,3)-33.3<0.1 (alternativamente en valor absoluto) para asegurar que la diferencia entre 33.3… y 33.3 sea menor que 0.1.

Precondiciones

La instrucción assert se puede usar en cualquier lugar en que se admite una instrucción y no sólo para probar una función. De hecho, es conveniente utilizarla para comprobar la validez de los parámetros de una función. En otras palabras, la instrucción assert se utiliza para verificar las condiciones previas (precondiciones) que deben cumplir los parámetros de acuerdo a lo especificado en el contrato y el objetivo de la receta de diseño de la función.

Por ejemplo, la función que determina si un número entero es o no par, de acuerdo al contrato tiene que asegurarse que el parámetro sea un número entero. Al respecto, la función predefinida type devuelve el nombre del tipo de su argumento (int, float, str o bool):

```
def par(x):
  assert type(x)==int
```

```
  return x % 2 == 0
```

En la función bisiesto, la precondición asegura que el año sea un número entero positivo:

```
def bisiesto(año):
  assert type(año)==int and año>0
  return año%4==0 and año%100!=0 or año%400==0
```

Un módulo para operar con triángulos

Una aplicación de los usos de las expresiones lógicas en precondiciones y pruebas se puede apreciar en un módulo con funciones para manejar triángulos. Se incluye una función para determinar si 3 números forman un triángulo, las funciones para calcular el perímetro y el área, y otra función para determinar si un triángulo es equilátero, isósceles o escaleno, es decir, si tiene los tres lados iguales, dos iguales o todos distintos.

La función lógica que verifica que tres números forman un triángulo puede utilizarse en las precondiciones de las otras funciones. Tres números definen un triángulo si son positivos y si la suma de dos cualesquiera de ellos sea mayor que el tercero.

```
#esTriangulo: num num num -> bool
#True si x, y, z forman un triangulo
#ejs: esTriangulo(3,4,5)->True, esTriangulo(1,2,3)->False
def esTriangulo(x,y,z):
    assert type(x)==int or type(x)==float
    assert type(y)==int or type(y)==float
    assert type(z)==int or type(z)==float
    return x>0 and y>0 and z>0 and x+y>z and x+z>y and y+z>x
assert esTriangulo(3,4,5)
assert not esTriangulo(1,2,3)
```

El contrato de la función indica que los 3 parámetros deben ser números y por lo tanto hay que asegurar que efectivamente lo sean, en otras palabras, se debe verificar que cada uno de ellos sea del tipo int o del tipo float. Cualquiera de los parámetros que no sea un número producirá que la expresión (condición) entregue un resultado False y, por lo tanto, la instrucción assert terminará el programa. Solo si los 3 parámetros cumplen la precondición, es decir, si los 3 son números, entonces se evalúa la condición para examinar si forman o no un triángulo.

Para no repetir tres veces la precondición para cada uno de los tres parámetros, la función esTriangulo puede reescribirse de la siguiente manera:

```
#esTriangulo: num num num -> bool
#True si x, y, z forman un triangulo
#ejs: esTriangulo(3,4,5)->True, esTriangulo(1,2,3)->False
def esTriangulo(x,y,z):
```

```
    def esNum(n): return type(n)==int or type(n)==float
    assert esNum(x) and esNum(y) and esNum(z)
    return x>0 and y>0 and z>0 and x+y>z and x+z>y and y+z>x
assert esTriangulo(3,4,5)
assert not esTriangulo(1,2,3)
```

La precondición de la función esTriangulo invoca a la función interna esNum que devuelve True si su argumento es un número. Así se evita escribir 3 veces la expresión type(parámetro)==int or type(parámetro)==float. Una función interna solo puede ser usada dentro de la función que la contiene, se puede escribir hacia el lado y no necesita escribirse con receta de diseño porque no será utilizada en otros programas.

```
#area: num num num -> float
#area de triangulo de lados x, y, z
#ejs: area(3,4,5) -> 6.0, area(1,1,1)->0.4...
def area(x,y,z):
    assert esTriangulo(x,y,z)
    s=(x+y+z)/2
    return math.sqrt(s*(s-x)*(s-y)*(s-z))
assert area(3,4,5)==6.0
assert abs(area(1,1,1)-0.4)<0.1
```

```
#perimetro: num num num -> num
#perimetro de triangulo de lados x, y, z
#ej: triangulo(3,4,5) -> 12
def perimetro(x,y,z):
    assert esTriangulo(x,y,z)
    return x+y+z
assert perimetro(3,4,5)==12
```

```
#tipo: num num num -> str
#'equilatero', 'isosceles' o 'escaleno' en caso que
#x, y, z formen un triángulo equilátero, isósceles o escaleno
#ej: tipo(1,1,1)->'equilatero'
#ej: tipo(2,2,3)->'isosceles'
#ej: tipo(3,4,5)->'escaleno'
def tipo(x,y,z):
    assert esTriangulo(x,y,z)
    if x==y and x==z:
        return 'equilatero'
    elif x==y or x==z or y==z:
        return 'isosceles'
    else:
        return 'escaleno'
assert tipo(1,1,1)=='equilatero'
```

```
assert tipo(2,2,3)=='isosceles'
assert tipo(3,4,5)=='escaleno'
```

Las precondiciones de las funciones área, perímetro y tipo invocan a la función lógica esTriangulo evitando reescribir la condición que asegura que 3 números formen un triángulo. En la función área se incluye una prueba para el caso que el valor del área no sea un número exacto. El uso de la función abs asegura que valor de la raíz sea mayor o menor que el valor esperado en menos de una décima. Por otra parte, se incluyen tres pruebas en la función tipo, una por cada uno de los tipos de triángulo.

Un programa que usa el módulo triángulo para leer 3 números y escribir el tipo de triángulo que forman y los valores de su área y perímetro puede escribirse de la siguiente manera:

```
from  triangulo import *
print('Tipo, área y perímetro de triángulo de lados a,b,c')
a=float(input('a?'))
b=float(input('b?'))
c=float(input('c?'))
if esTriangulo(a,b,c):
  print("tipo:", tipo(a,b,c))
  print("perímetro:", perimetro(a,b,c))
  print("área:", area(a,b,c))
else:
  print("no forman un triángulo")
```

El programa se asegura que los tres números formen un triángulo antes de invocar a las funciones tipo, perimetro y area. Si los números no forman un triángulo se escribe un mensaje informando la situación y evitando un error de aserción. Un ejemplo de ejecución del programa establece el siguiente diálogo con un usuario:

```
Tipo, área y perímetro de triángulo de lados a,b,c
a?3
b?4
c?5
tipo: escaleno
perímetro: 12
área: 6.0
```