Python II: Lógica booleana

Un *boolean* es una variable binaria, que sólo tiene 2 valores posibles: 0 y 1. En Python, las variables booleanas sólo pueden tener valores **`False`** y **`True`**. Estos dos valores pueden tratarse como cualquier otro, y pueden ser convertidos a y desde valores numéricos:

```
int(True)

> 1

bool(0)

> False
```

Lo que significa que podemos usarlos como tal:

```
5 * True

> 5

5 * False

> 0
```

Todos los valores distintos de cero (y `None`) se convierten en `True`:

```
bool('False')

> True

bool(43)

> True
```

Mientras que el número cero, **`None`** y strings vacíos se evalúan como **`False`**:

```
bool(0)

> False

bool(None)

> False

bool('')

> False
```

Podemos evaluar condiciones booleanas usando los símbolos matemáticos usuales:

```
3 > 1

> True

9 <= 4

> False

3 == int('3')

> True

3 != 1

> True
```

(Comparen esto con el uso de operadores booleanos como **`-gt`** en Bash.) Para evaluar una igualdad debemos usar dos **==**, ya que un solo **=** significa “asignar”. También podemos evaluar directamente si un valor está en un cierto rango:

```
x = 3

1 < x <= 5

> True
```

Podemos combinar condiciones booleanas usando los operadores lógicos **`&`** (AND) y **`|`** (OR):

```
(9 > len('hola')) & (3 == 2+1)

> True

(9 > 4) | False

> True
```

Para combinar condiciones debemos identificar cada condición con paréntesis, o el comportamiento no será el esperado:

```
9 > len('hola') & 3 == 2+1

> False
```

Pero más fácil de recordar que los símbolos es usar las palabras:

```
(9*3 == 27.0) and (3 == int('3'))

> True

(7 > 10) or ('b' != 'ab')
```

```
> True
```

También existe el operator NOT, que se escribe simplemente **`not`**:

```
not (3 == 5)

> True

(2 < 3) and not ('a' > 'c')

> True
```

La última comparación evalúa la posición de cada literal alfanuméricamente.

Para evaluar si un elemento existe en un iterable usamos **`in`**:

```
'c' in 'abc'

> True

4 in (1, 2, 3)

> False

4 not in (1, 2, 3)

> True
```

Por último, el operador lógico **`is`** evalúa si dos elementos se refieren *al mismo objeto en la memoria*, en lugar de comparar el valor de ambos. Veamos un ejemplo:

```
list1 = []

list2 = []

list3 = list1

list1 == list2

> True

list2 == list3

> True

list3 is list1

> True

list3 is list2

> False
```

Por motivos de eficiencia, Python tiene espacios de memoria reservados para los números enteros -5…256 (son “instancias únicas”, *singletons*), de manera que

```
a = 1

b = 1

a is b

> True
```

pero

```
a = 257

b = 257

a is b

> False
```

El uso de **`is`** se recomienda en lugar de **`==`** para “instancias únicas” como **`None`**; para todo el resto típicamente usaríamos **`==`**:

```
a = None

a is None

> True
```

Cláusulas condicionales

Podemos usar la lógica booleana para tomar decisiones sobre qué hacer con declaraciones if-elif-else, con la siguiente sintaxis:

```
if condicion:
     comandos
elif condicion:
     comandos
...
else:
     comandos
```

Por ejemplo,

```
x = 'string largo'
print(x)
if len(x) <= 3:
     print('x es corto')
elif len(x) <= 8:
     print('x es relativamente largo')
elif len(x) <= 12:
     print('x es bastante largo!')
else:
     print('x no termina nunca')
print('No más if')
> string largo
> x es bastante largo!
> No más if
```

Aquí apareció otra de las características definitorias de Python: las cláusulas se identifican simplemente indentando las líneas. La última línea, que sigue la indentación original, no es parte del bloque condicional. En otros lenguajes, típicamente se usan brackets (`{}`). Notarán que en Jupyter (así como en IPython), al terminar una línea con dos puntos la línea siguiente se indenta sola. (Si usan un editor de texto, deben fijarse que su editor de texto interprete la tecla **`TAB`** como un conjunto de espacios y no un tab, pues este último causa problemas de compatibilidad. Es práctica recomendada que la indentación sea de 4 espacios.)

Consideremos el siguiente ejemplo:

```
if 'bla' in ('bla', 'ble', 'bli'):
     print('Se cumple la condición externa')
     if 10 > 20:
          print('condición 1')
     else:
          print('no condición 1')
          print('Te equivocaste')
```

```
        print('Entre condiciones')
        if 20 > 10:
            print('condición 2')
        else:
            print('no condición 2')
            print('Te equivocaste')
        print('Fin de la condición externa')
print('Fin del bloque')
> Se cumple la condición externa
> no condición 1
> Te equivocaste
> Entre condiciones
> condición 2
> Fin de la condición externa
> Fin del bloque
```

Podemos usar condicionales if para asignar variables en una línea, reemplazando esto:

```
if condición:
    x = valor1
else:
    x = valor2
```

Por esto:

```
x = valor1 if condición else valor2
```

Por ejemplo,

```
edad = 19
categoria = 'Adulta' if edad >= 18 else 'Niña'
categoria
> 'Adulta'
```

Existe otro tipo de cláusula condicional, que sirve para “atrapar” errores (o “excepciones”). Por ejemplo, podemos atrapar el siguiente error:

```
5 > 'bla'

> TypeError: '>' not supported between instances of 'int'
and 'str'
```

de la siguiente manera:

```
try:

     5 > 'bla'

     print('Funciona!')

except TypeError as err:

     print(f'no se puede! {err}')

> no se puede! '>' not supported between instances of
'int' and 'str'

try:

     5 > 6

     print('Funciona!')

except TypeError:

     print('no se puede!')

> Funciona!
```

Luego de atrapar el error pueden seguir trabajando con normalidad. Noten que deben atrapar el error correcto:

```
try:

     5 > 'bla'

except IndexError:

     print('no se puede')

> TypeError: '>' not supported between instances of 'int'
and 'str'
```

En principio podrían atrapar todos los errores posibles simplemente escribiendo **`except`** a solas, pero en la práctica **no deben hacerlo** ya que es muy peligroso: ¡es

mejor saber qué error están cometiendo! Se pueden, en todo caso, agregar tantas cláusulas except como sea necesario:

```
try:
    5 > 'bla'
    print('Operación terminada')
except IndexError as err:
    print(f'No se puede (IndexError): {err}')
except TypeError as err:
    print(f'No se puede (TypeError): {err}')
> No se puede (TypeError): '>' not supported between instances of 'int' and 'str'
```

Hay una opción adicional al atrapar excepciones. En lugar de escribir todo el código en caso que no haya error en el primer bloque, podemos dar otro uso a la cláusula **`else`**:

```
try:
    5 > 6
except TypeError:
    print('no se puede!')
else:
    print('Operación terminada')
> Operación terminada
```

Esto nos permite aislar la línea que produce el error, lo que puede hacer una diferencia enorme al momento de identificar la fuente del error.