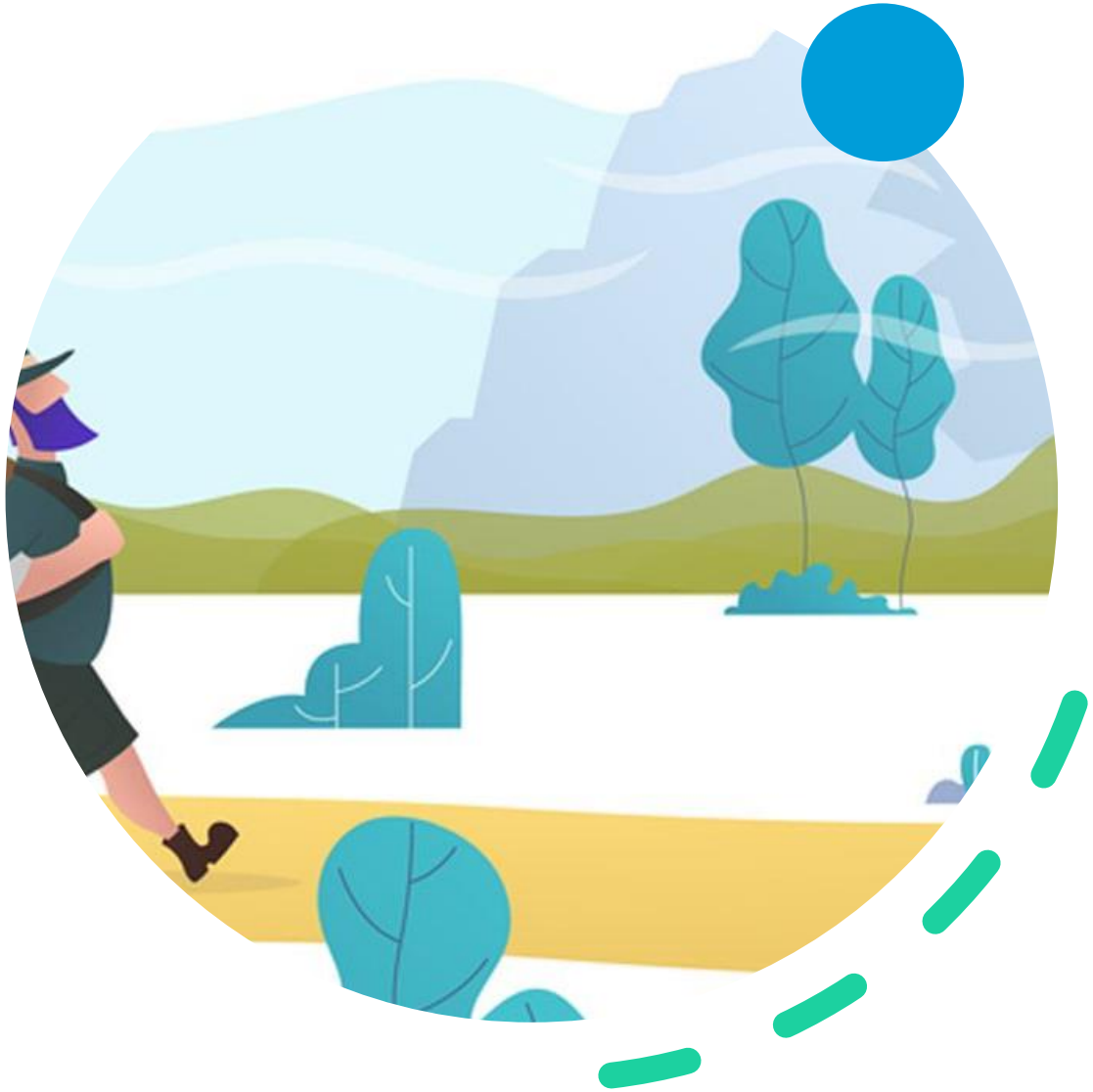
Lógica de computadora

¿Te has dado cuenta de que las condiciones que hemos usado hasta ahora han sido muy simples, por no decir, bastante primitivas? Las condiciones que utilizamos en la vida real son mucho más complejas. Veamos este enunciado:



Si tenemos tiempo libre, y el clima es bueno, saldremos a caminar.

Hemos utilizado la conjunción and (y), lo que significa que salir a caminar depende del cumplimiento simultáneo de estas dos condiciones. En el lenguaje de la lógica, tal conexión de condiciones se denomina conjunción. Y ahora otro ejemplo:

Si tu estás en el centro comercial o yo estoy en el centro comercial, uno de nosotros le comprará un regalo a mamá.

La aparición de la palabra or (o) significa que la compra depende de al menos una de estas condiciones. En lógica, este compuesto se llama una disyunción.

Está claro que Python debe tener operadores para construir conjunciones y disyunciones. Sin ellos, el poder expresivo del lenguaje se debilitaría sustancialmente. Se llaman operadores lógicos.

and

Un operador de conjunción lógica en Python es la palabra *and*. Es un operador binario **con una prioridad inferior a la expresada por los operadores de comparación**. Nos permite codificar condiciones complejas sin el uso de paréntesis como este:

```
counter > 0 and value == 100
```

El resultado proporcionado por el operador and se puede determinar sobre la base de la tabla de verdad.

Si consideramos la conjunción de A and B, el conjunto de valores posibles de argumentos y los valores correspondientes de conjunción se ve de la siguiente manera:

Argumento `A`	Argumento `B`
`False`	`False`
`False`	`True`
`True`	`False`
`True`	`True`

or

Un operador de disyunción es la palabra or. Es un operador binario con una prioridad más baja que and (al igual que + en comparación con *). Su tabla de verdad es la siguiente:

Argumento `A`	Argumento `B`
`False`	`False`
`False`	`True`
`True`	`False`
`True`	`True`

not

Además, hay otro operador que se puede aplicar para condiciones de construcción. Es un operador unario que realiza una negación lógica. Su funcionamiento es simple: convierte la verdad en falso y lo falso en verdad

Este operador se escribe como la palabra not, y su prioridad es muy alta: igual que el unario + y -. Su tabla de verdad es simple:

Argumento	`not` Argumento
`False`	`True`
`True`	`False`

Expresiones lógicas

Creemos una variable llamada var y asignémosle 1. Las siguientes condiciones son equivalentes a pares:

```
# Ejemplo 1:
print(var > 0)
print(not (var <= 0))


# Ejemplo 2:
print(var != 0)
print(not (var == 0))
```

- Puedes estar familiarizado con las leyes de De Morgan. Dicen que:
- *La negación de una conjunción es la separación de las negaciones.*
- *La negación de una disyunción es la conjunción de las negaciones.*

Escribamos lo mismo usando Python:

```
not (p and q) == (not p) or (not q)
not (p or q) == (not p) and (not q)
```

Observa como se han utilizado los paréntesis para codificar las expresiones: las colocamos allí para mejorar la legibilidad.

Deberíamos agregar que ninguno de estos operadores de dos argumentos se puede usar en la forma abreviada conocida como op=. Vale la pena recordar esta excepción.

Leyes de De Morgan

$p \wedge \sim (q \vee \sim p)$

$p \wedge (\sim q \wedge p)$ Leyes de De Morgan

$p \wedge (p \wedge \sim q)$ Propiedad Conmutativa

$(p \wedge p) \wedge \sim q$ Propiedad Asociativa

$p \wedge \sim q$ Idempotencia

Valores lógicos frente a bits individuales

Los operadores lógicos toman sus argumentos como un todo, independientemente de cuantos bits contengan. Los operadores solo conocen el valor: cero (cuando todos los bits se restablecen) significa False; no cero (cuando se establece al menos un bit) significa True.

El resultado de sus operaciones es uno de estos valores: False o True. Esto significa que este fragmento de código asignará el valor True a la variable j si i no es cero; de lo contrario, será False.

```
i = 1

j = not not i
```

Operadadores bit a bit

Sin embargo, hay cuatro operadores que le permiten manipular bits de datos individuales. Se denominan operadores bit a bit.

Cubren todas las operaciones que mencionamos anteriormente en el contexto lógico, y un operador adicional. Este es el operador xor (significa o exclusivo), y se denota como ^ (signo de intercalación).

Aquí están todos ellos:

& (ampersand) - conjunción a nivel de bits.
| (barra vertical) - disyunción a nivel de bits.
~ (tilde) - negación a nivel de bits.
^ (signo de intercalación) - o exclusivo a nivel de bits (xor).

Operaciones bit a bit (`&` , `|` , y `^`)

Argumento `A`	Argumento `B`	`A & B`	`A | B`
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Operaciones bit a bit (~)

Argumento	~	Argumento
0		1
1		0

Hagámoslo más fácil:

& requieres exactamente dos 1s para proporcionar 1 como resultado.

| requiere al menos un 1 para proporcionar 1 como resultado.

^ requiere exactamente un 1 para proporcionar 1 como resultado.

Comentario: los argumentos de estos operadores deben ser enteros. No debemos usar flotantes aquí.

La diferencia en el funcionamiento de los operadores lógicos y de bits es importante: los operadores lógicos no penetran en el nivel de bits de su argumento. Solo les interesa el valor entero final.

Los operadores bit a bit son más estrictos: tratan con cada bit por separado. Si asumimos que la variable entera ocupa 64 bits (lo que es común en los sistemas informáticos modernos), puede imaginar la operación a nivel de bits como una evaluación de 64 veces del operador lógico para cada par de bits de los argumentos. Su analogía es obviamente imperfecta, ya que en el mundo real todas estas 64 operaciones se realizan al mismo tiempo (simultáneamente).