# 3 Condicionales

## Expresiones booleanas

Una **expresión booleana** es una expresión que es verdadera o falsa. Los siguientes ejemplos usan el operador `==`, que compara dos operandos y produce `True` si son iguales y `False` de lo contrario:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` y `False` son valores especiales que pertenecen a la clase `bool`; no son cadenas.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

El operador `==` es uno de los **operadores de comparación**; los otros son:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

Copyright 2026

Aunque es probable que estas operaciones te resulten familiares, los símbolos de Python son diferentes de los símbolos matemáticos para las mismas operaciones. Un error común es usar un único signo igual (`=`) en lugar de un doble signo igual (`==`). Recuerda que `=` es un operador de asignación y `==` es un operador de comparación. No hay tal cosa como `=<` o `=>`.

# Operadores lógicos

Hay tres **operadores lógicos**: `and`, `or`, y `not`. La semántica (significado) de estos operadores es similar a su significado en inglés. Por ejemplo,

`x > 0 and x < 10`

es cierto solo si `x` es mayor que 0 **y** menor que 10.

`n%2 == 0 or n% 3 == 0` es verdadero si **cualquiera** de las dos condiciones es verdadera, es decir, si el número es divisible por 2 **o** 3.

Finalmente, el operador `not` niega una expresión booleana, por lo que `not(x>y)` es verdadero si `x>y` es falso; es decir, si `x` es menor o igual que `y`.

Estrictamente hablando, los operandos de los operadores lógicos deben ser expresiones booleanas, pero Python no es muy estricto. Cualquier número distinto de cero se interpreta como "verdadero".

```
>>> 17 and True
True
```

Esta flexibilidad puede ser útil, pero hay algunas sutilezas que pueden ser confusas. Es posible que desee evitarlo hasta que esté seguro de saber lo que está haciendo.

# Ejecución condicional

Para escribir programas útiles, casi siempre necesitamos la capacidad de verificar las condiciones y cambiar el comportamiento del programa en consecuencia. Las **declaraciones condicionales** nos dan esta habilidad. La forma más simple es la instrucción `if`:

```
if x > 0 :
    print('x is positive')
```

La expresión booleana después de la instrucción `if` se llama la **condición**. Terminamos la instrucción `if` con un carácter de dos puntos (:) y la(s) línea(s) después de la instrucción if deben estar sangradas.

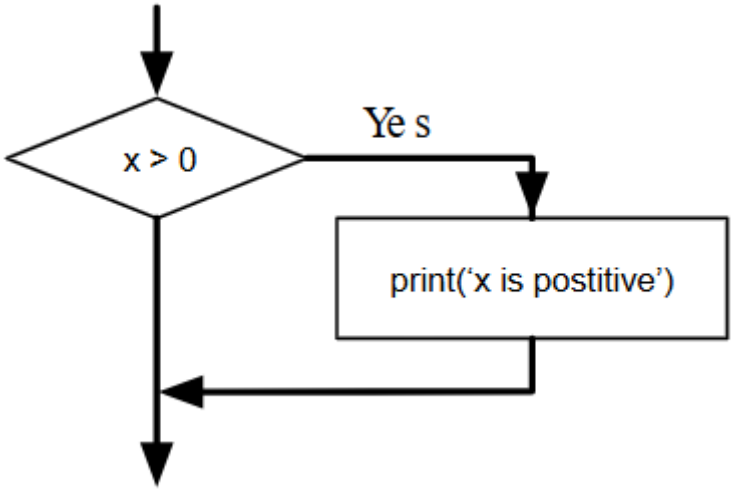



Imagen - Si logica

Si la condición lógica es verdadera, entonces se ejecuta la instrucción con sangría. Si la condición lógica es falsa, la instrucción con sangría se omite.

Las declaraciones `if` tienen la misma estructura que las definiciones de funciones o `for` loops [1]. La declaración consiste en una línea de encabezado que termina con el carácter de dos puntos (:) seguido de un bloque con sangría. Las declaraciones como esta se denominan **declaraciones compuestas** porque se extienden a lo largo de más de una línea.

No hay límite en el número de declaraciones que pueden aparecer en el cuerpo, pero debe haber al menos una. Ocasionalmente, es útil tener un cuerpo sin declaraciones (generalmente como un marcador de posición para el código que aún no ha escrito). En ese caso, puedes usar la instrucción `pass`, que no hace nada.

```
if x < 0 :
    pass          # need to handle negative values!
```

Si ingresa una instrucción `if` en el intérprete de Python, el indicador cambiará de tres puntos a tres puntos para indicar que se encuentra en medio de un bloque de declaraciones, como se muestra a continuación:

```
>>> x = 3
>>> if x < 10:
```

```
...     print('Small')
...
Small
>>>
```

# Ejecución alternativa

Una segunda forma de la sentencia `if` es **ejecución alternativa**, en la que hay dos posibilidades y la condición determina cuál se ejecuta. La sintaxis se ve así:

```
if x%2 == 0 :
    print('x is even')
else :
    print('x is odd')
```

Si el resto cuando `x` se divide por 2 es 0, entonces sabemos que `x` es par, y el programa muestra un mensaje a tal efecto. Si la condición es falsa, se ejecuta el segundo conjunto de sentencias.

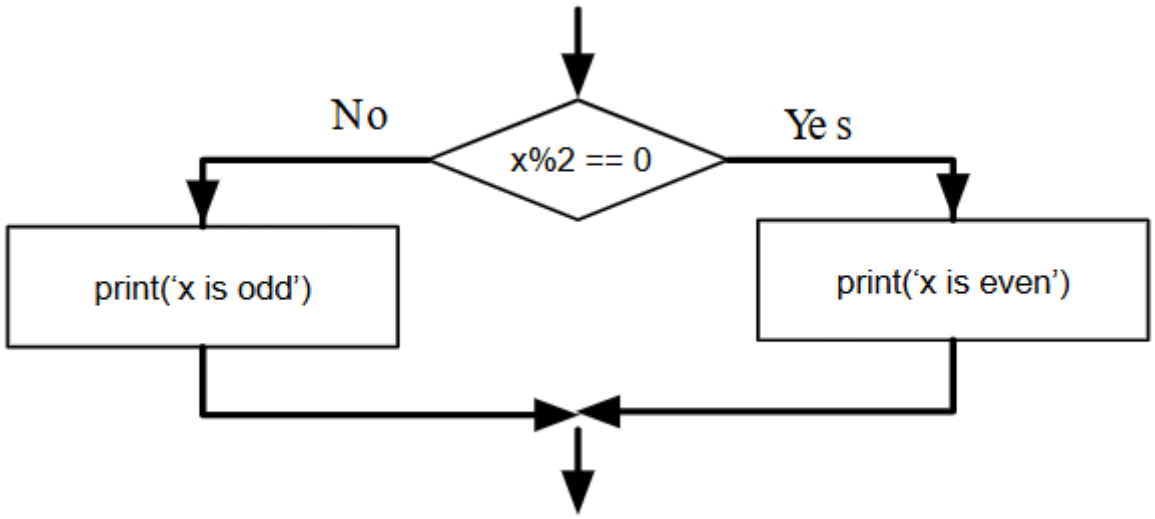



Imagen - Lógica de If-Then-Else

Dado que la condición debe ser verdadera o falsa, se ejecutará exactamente una de las alternativas. Las alternativas se denominan **ramas**, porque son ramas en el flujo de ejecución.

# Condicionales encadenados

A veces hay más de dos posibilidades y necesitamos más de dos ramas. Una forma de expresar un cálculo como ese es un **condicional encadenado**:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

"elif" es una abreviatura de "else if". De nuevo, exactamente una rama será ejecutada.

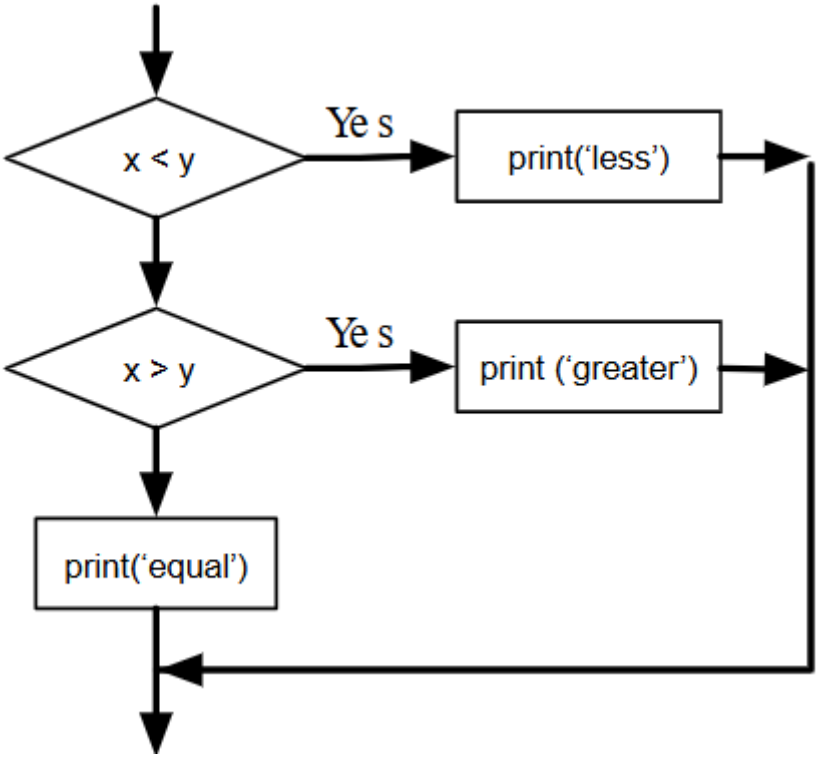



Imagen - Lógica de elif

No hay límite en el número de declaraciones `elif`. Si hay una cláusula `else`, tiene que estar al final, pero no tiene necesariamente que haber una.

```
if choice == 'a':
    print('Bad guess')
elif choice == 'b':
    print('Good guess')
elif choice == 'c':
```

```
    print('Close, but not correct')
```

Cada condición se comprueba en orden. Si el primero es falso, se marca el siguiente, y así sucesivamente. Si uno de ellos es verdadero, la rama correspondiente se ejecuta y la declaración finaliza. Incluso si más de una condición es verdadera, solo se ejecuta la primera rama verdadera.

# Condicionales anidados

Un condicional también puede ser anidado dentro de otro. Podríamos haber escrito el ejemplo de tres ramas como este:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

El condicional exterior contiene dos ramas. La primera rama contiene una declaración simple. La segunda rama contiene otra instrucción 'if', que tiene dos ramas propias. Esas dos ramas son ambas declaraciones simples, aunque también podrían haber sido declaraciones condicionales.

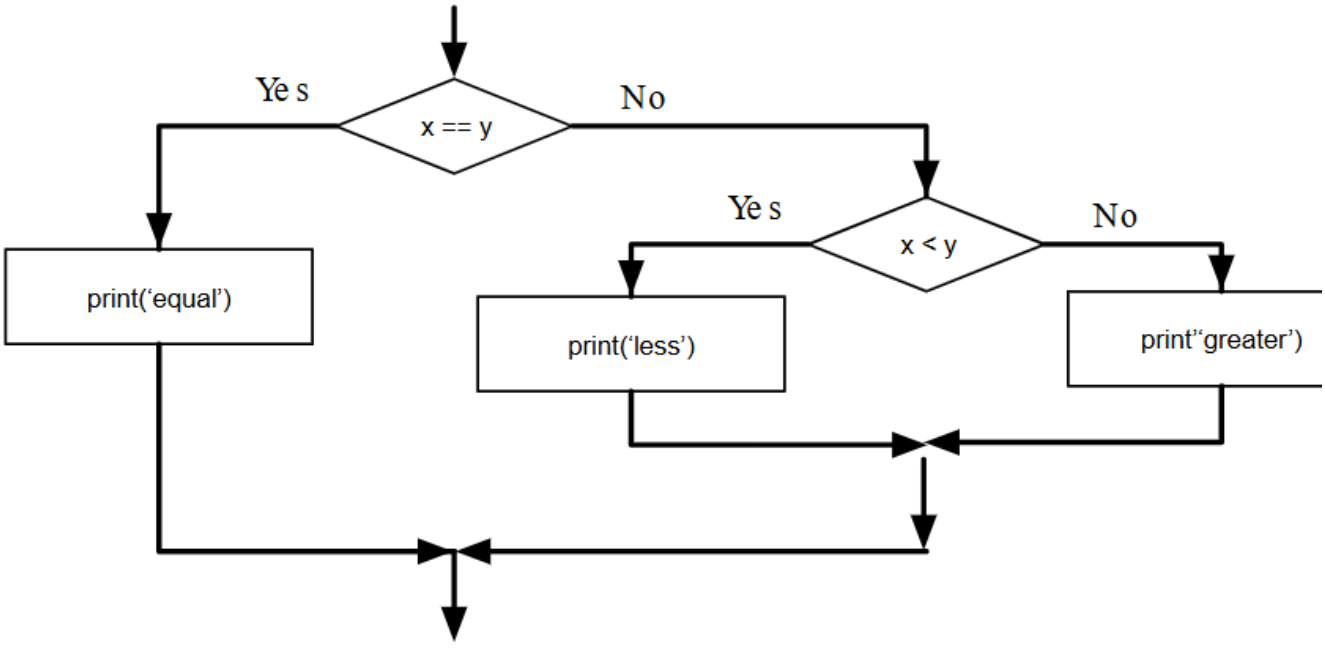


Imagen - Nested If Statements

Aunque la sangría de los enunciados hace que la estructura sea aparente, **los condicionales anidados** se vuelven difíciles de leer muy rápidamente. En general, es una buena idea evitarlos cuando puedas.

Los operadores lógicos a menudo proporcionan una forma de simplificar sentencias condicionales anidadas. Por ejemplo, podemos reescribir el siguiente código usando un solo condicional:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

La instrucción `print` se ejecuta solo si superamos ambas condiciones, por lo que podemos obtener el mismo efecto con el operador `y`:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

# Capturando excepciones utilizando try y except

Anteriormente vimos un segmento de código donde usamos las funciones `input` y `int` para leer y analizar un número entero ingresado por el usuario. También vimos lo peligroso que podría ser hacer esto:

```
>>> prompt = "What...is the airspeed velocity of an unladen swallow?\n"
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

Cuando ejecutamos estas declaraciones en el intérprete de Python, obtenemos un nuevo mensaje del intérprete, pensamos "oops" y pasamos a nuestra siguiente declaración.

Copyright 2026

Sin embargo, si coloca este código en una secuencia de comandos de Python y se produce este error, la secuencia de comandos se detiene inmediatamente dejando una traza de lo que se estaba ejecutando cuando el programa falló. No ejecuta la siguiente sentencia.

Aquí hay un programa de ejemplo para convertir una temperatura Fahrenheit a una temperatura Celsius:

https://trinket.io/embed/python3/27bbe8b7ab

Si ejecutamos este código y le damos una entrada no válida, simplemente falla con un mensaje de error hostil:

```
python fahren.py
Enter Fahrenheit Temperature:72
22.22222222222222
```

```
python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

Existe una estructura de ejecución condicional incorporada en Python para manejar estos tipos de errores esperados e inesperados llamados "try/except". La idea de `try` y `except` es que usted sabe que alguna secuencia de instrucciones puede tener un problema y que desea agregar algunas instrucciones para que se ejecuten si se produce un error. Estas declaraciones adicionales (el bloque de excepción) se ignoran si no hay ningún error.

Puede pensar en las funciones `try` y `except` en Python como una “póliza de seguro” en una secuencia de declaraciones.

Podemos reescribir nuestro convertidor de temperatura de la siguiente manera:

https://trinket.io/embed/python3/5dbec1550b

Python comienza ejecutando la secuencia de instrucciones en el bloque `try`. Si todo va bien, salta el bloque `except` y continúa. Si ocurre una excepción en el bloque `try`, Python salta del bloque `try` y ejecuta la secuencia de instrucciones en el bloque `except`.

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.22222222222222
```

```
python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

Manejar una excepción con una declaración `try` se llama **capturar** una excepción. En este ejemplo, la cláusula `except` imprime un mensaje de error. En general, detectar una excepción le da la oportunidad de solucionar el problema, o volver a intentarlo, o al menos finalizar el programa con gracia.

# Evaluación de cortocircuito de expresiones lógicas

Cuando Python está procesando una expresión lógica como `x>= 2 and (x/y)>2`, evalúa la expresión de izquierda a derecha. Debido a la definición de `y`, si `x` es menor que 2, la expresión `x>= 2` es `False` y, por lo tanto, toda la expresión es `False` independientemente de si `(x/y)>` se evalúa como "True" o "False".

Cuando Python detecta que no hay nada que ganar al evaluar el resto de una expresión lógica, detiene su evaluación y no realiza los cálculos en el resto de la expresión lógica. Cuando la evaluación de una expresión lógica se detiene porque ya se conoce el valor general, se llama **cortocircuitar** la evaluación.

Si bien esto puede parecer un punto delicado, el comportamiento de cortocircuito conduce a una técnica inteligente llamada **patrón guardián**. Considere la siguiente secuencia de código en el intérprete de Python:

```
>>> x = 6
>>> y = 2
```

```
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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` se evaluó como `Falso` por lo que el `(x/y)` nunca se ejecutó debido a que **se cortocircuitó** la evaluación y no hubo error.

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

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

En la primera expresión lógica, `x>=2` es `False`, por lo que la evaluación se detiene en `y`. En la segunda expresión lógica, `x >= 2` es `True` pero `y!=0` es `False` por lo que nunca alcanzamos `(x/y)`.

Copyright 2026

En la tercera expresión lógica, el `y!=0` es **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 **guard** para asegurarnos de que solo ejecutamos `(x/y)` si `y` no es cero.

# Depuración

El seguimiento que Python muestra cuando se produce un error contiene mucha información, pero puede ser abrumador. Las partes más útiles suelen ser:

- Qué tipo de error fue, y
- Donde ocurrió.

Los errores de sintaxis son generalmente fáciles de encontrar, pero hay algunos enrevesados. Los errores de espacios en blanco pueden ser complicados porque los espacios y las pestañas son invisibles y estamos acostumbrados a ignorarlos.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

En este ejemplo, el problema es que la segunda línea está sangrada por un espacio. Pero el mensaje de error apunta a `y`, lo que es engañoso. En general, los mensajes de error indican dónde se descubrió el problema, pero el error real puede ser anterior en el código, a veces en una línea anterior.

En general, los mensajes de error te indican dónde se descubrió el problema, pero a menudo no es donde se causó.

# Ejercicios (voluntario sin corrección)

**Ejercicio 1** reescribe tu cálculo de pago para darle al empleado 1.5 veces la tarifa por hora por las horas trabajadas por encima de las 40 horas.

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

**Ejercicio 2** reescribe tu programa de pago usando `try` y `except` para que su programa maneje la entrada no numérica correctamente imprimiendo un mensaje y saliendo del programa. A continuación se muestran dos ejecuciones del programa:

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input
```

```
Enter Hours: forty
Error, please enter numeric input
```

**Ejercicio 3** Escribe un programa para solicitar una puntuación entre 0.0 y 1.0. Si la puntuación está fuera de rango, imprime un mensaje de error. Si la puntuación está entre 0.0 y 1.0, imprime una calificación usando la siguiente tabla:

```
Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
< 0.6    F
~~~
```

Introduce la puntuación: 0,95 A ~ ~

```
Enter score: perfect
Bad score
```

```
Enter score: 10.0
Bad score
```

```
Enter score: 0.75
C
```

```
Enter score: 0.5
F
```

Ejecute el programa repetidamente como se muestra arriba para probar los diferentes valores de entrada.

---

> [1]. Aprenderemos sobre las funciones en el Capítulo 4 y los bucles en el Capítulo 5. ↩

---

Revision #1
Created 2025-04-05 08:58:37 CEST by Juanjo Mejías
Updated 2025-04-05 09:01:17 CEST by Juanjo Mejías