ChatGPT

Sentencias condicionales en Python

Entendiendo las instrucciones condicionales

Una **sentencia condicional** permite que un programa ejecute cierto bloque de código solo si se cumple una condición lógica, y opcionalmente ejecute un bloque alternativo si la condición no se cumple [1] [2]. En otras palabras, es una estructura de control que **toma decisiones** durante la ejecución del programa. En Python, la sentencia condicional básica es `if` (si), y se considera una *sentencia compuesta* porque abarca una línea de encabezado y un bloque indentado de una o varias líneas [3].

La sintaxis de una instrucción `if` en Python incluye los siguientes elementos [4]:

- La palabra reservada `if`, que inicia la sentencia condicional.
- Una **condición** (expresión booleana) a evaluar, por ejemplo `x > 0`. Esta expresión puede ser verdadera (`True`) o falsa (`False`).
- Un signo de **dos puntos (`:`)** al final de la línea del `if`, que indica el comienzo del bloque de código condicional.
- Un **bloque de código indentado** (sangrado) bajo el `if`, que contiene una o más instrucciones a ejecutar **solo si** la condición es verdadera. En Python, la indentación estándar es de 4 espacios por nivel [5] [6]. Todas las líneas de este bloque deben tener el mismo nivel de sangría.

Por ejemplo, consideremos el siguiente código condicional simple:

```
x = 5
if x > 0:
    print("x es positivo")
print("Fin del programa")
```

En este caso, la condición `x > 0` es verdadera si `x` es mayor que 0. Si lo es, se ejecuta la línea indentada `print("x es positivo")`; de lo contrario, ese `print` se omite. En ambos casos, después del `if` el programa continúa con la siguiente instrucción no indentada (`print("Fin del programa")`).

Al escribir condicionales en Python, recuerda que **no** se usan paréntesis alrededor de la condición (son opcionales), y que **sí** se debe usar el `:` y la indentación adecuada para delimitar el bloque [7]. Es un error común de sintaxis usar el operador de asignación `=` en lugar del operador de comparación `==` dentro de la condición; por ejemplo, `if x = 5:` es incorrecto y causará un error, mientras que la comparación correcta es `if x == 5:` [8].

Por qué usamos condicionales

Las sentencias condicionales se usan porque la mayoría de los programas necesitan **tomar decisiones** y ejecutar acciones diferentes dependiendo de ciertas condiciones. Sin condicionales, un programa sería una secuencia fija de pasos; con condicionales, el programa puede *ramificarse* y responder de manera dinámica a distintas situaciones. En la vida diaria de un programa, a menudo "si sucede X, haz

A; de lo contrario, haz B". Por ejemplo, podríamos querer verificar si un usuario ha introducido una contraseña correcta: si es correcta, permitir el acceso; si no, denegarlo.

En términos generales, los condicionales nos **permiten controlar el flujo lógico** de un programa de forma clara y flexible [1]. Como explica la documentación, para escribir programas útiles casi siempre necesitamos la capacidad de comprobar condiciones y cambiar el comportamiento del programa de acuerdo con ellas, y *las sentencias condicionales nos brindan precisamente esa capacidad* [9]. En resumen, usamos condicionales para que el programa **no haga siempre lo mismo**, sino que pueda realizar distintas acciones según los valores de entrada, el estado del sistema u otros criterios lógicos.

Siguiendo el flujo de un programa

Por defecto, un programa en Python (y en muchos lenguajes) ejecuta las instrucciones en **flujo secuencial**, es decir, en el orden en que aparecen, de arriba hacia abajo [10]. Las sentencias condicionales alteran ese flujo secuencial introduciendo **ramas**: partes del código que solo se ejecutan si se cumple cierta condición. Para seguir el flujo de ejecución de un programa con condicionales, debemos tener en cuenta qué camino toma el código en tiempo de ejecución.

Cuando el programa llega a una sentencia `if`, **evalúa la condición** booleana asociada [11]. Si la condición es verdadera (`True`), se ejecuta el bloque de código indentado bajo el `if`; si la condición es falsa (`False`), ese bloque se salta por completo [12] [13]. En el caso de una estructura `if ... else`, el flujo es mutuamente excluyente: si la condición del `if` es verdadera se ejecuta el bloque *if*, y si es falsa se ejecuta en cambio el bloque *else*, garantizando que **uno u otro** camino se ejecute siempre [14] [15]. Una vez ejecutado uno de los bloques (if o else), la estructura condicional termina y el programa continúa con la instrucción siguiente después de todo el bloque `if/else` [13].

Podemos visualizar el flujo con un ejemplo sencillo de `if-else`:

```
edad = 17
if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
print("Fin del programa")
```

En este fragmento, si `edad >= 18` resulta verdadera, el programa imprime "Eres mayor de edad" y *no* ejecuta el bloque `else`. Si la condición es falsa (por ejemplo, edad es 17), el programa saltará el bloque del `if` y ejecutará el bloque `else`, imprimiendo "Eres menor de edad". En cualquier caso, después de la bifurcación, el flujo se reúne de nuevo y se imprime "Fin del programa".

En resumen, *seguir el flujo* de un programa con condicionales implica trazar mentalmente (o con ayuda de un diagrama) qué ruta tomará el programa dependiendo de cada condición, sabiendo que las ramas verdaderas ejecutan cierto código y las falsas simplemente continúan después del bloque condicional.

Entendiendo un diagrama de flujo

Un **diagrama de flujo** es una representación visual de la lógica de un algoritmo o programa, utilizando símbolos estándar para mostrar pasos y decisiones. Es muy útil para planificar y entender el flujo antes

de escribir código. En los diagramas de flujo, las acciones o procesos se suelen representar con **rectángulos**, y las **decisiones condicionales** con un **rombo** (diamante) del cual salen dos flechas: típicamente una marcada como "Sí" (o True) y otra como "No" (o False) [16]. Dentro del rombo se escribe la condición a evaluar, y a cada lado del rombo se indican las ramas a seguir según el resultado de la condición (por convención, muchas veces la rama que sale por la derecha es para el caso verdadero y la izquierda para el falso, aunque esto puede variar) [16].

Por ejemplo, la lógica de *"pedir un número y comprobar si es negativo"* se puede ilustrar con un diagrama de flujo así:

- **Inicio** → (Pedir número al usuario) → **[Rombo: número < 0?]** → *Si* (True) → (Mostrar mensaje "El número es negativo") → → *continúa* → **Fin**
  ↘ *No* (False) ↙ (Si no es negativo, no hacer nada especial) → **Fin**

En este diagrama, el rombo "¿número < 0?" bifurca el flujo: si la respuesta es *sí* (True), se toma una acción (avisar que es negativo); si es *no* (False), se omite esa acción. Al final, ambas ramas se unen de nuevo hacia el fin del programa.

Los diagramas de flujo ayudan a **visualizar todas las rutas posibles** de ejecución de un programa de forma clara. Cada símbolo estándar tiene un significado específico: el **óvalo** suele indicar inicio o fin, el **rectángulo** una operación o proceso, el **paralelogramo** una entrada o salida (por ejemplo leer datos o imprimir resultados), y el **rombo** una decisión condicional [17] [16]. Aprender a leer y dibujar estos diagramas facilita entender el flujo de control, especialmente cuando hay múltiples condiciones o bucles involucrados. Una vez que comprendemos el diagrama de flujo de un problema, resulta más sencillo traducir esa lógica a código Python usando condicionales y otras estructuras.

Creando subcondiciones (condiciones compuestas)

En muchos casos, necesitaremos evaluar **más de una condición** a la vez para tomar una decisión. Python proporciona los **operadores lógicos** `and` (y), `or` (o) y `not` (no) para *combinar subcondiciones* dentro de una condición más grande [18]. Una condición compuesta con `and` será verdadera solo si *todas* las subcondiciones son verdaderas, mientras que con `or` será verdadera si *al menos una* subcondición es verdadera [19]. El operador `not` invierte el valor lógico de una condición, volviéndola True si era False y viceversa [20].

Por ejemplo, si queremos verificar que un número `x` está en un rango específico, podríamos usar una condición compuesta:

```
if x > 0 and x < 10:
    print("x es un número positivo de un solo dígito")
```

Esta sentencia `if` se ejecutará **solamente** si ambas subcondiciones son verdaderas, es decir, si `x` es mayor que 0 **y** además `x` es menor que 10. En caso de que `x` no cumpla alguna de esas dos condiciones, la expresión completa se evalúa como falsa y el bloque no se ejecuta [19]. De forma similar, una condición usando `or` como `if x < 0 or x > 100:` se cumpliría si *cualquiera* de las dos subcondiciones es verdadera (en este ejemplo, si el valor de `x` es menor que 0 **o** mayor que 100).

Es importante notar que Python evalúa estas expresiones de izquierda a derecha y emplea **evaluación de cortocircuito**: en una expresión con `and`, si la primera subcondición resulta falsa, Python **no**

evalúa la segunda (pues el resultado de `and` ya no puede ser True) [21]. Con `or`, ocurre lo contrario: si la primera subcondición es verdadera, se omite la evaluación de la segunda, porque ya se sabe que el resultado será True. Esto puede ser útil para evitar errores o cálculos innecesarios; por ejemplo, en `if divisor != 0 and n/divisor > 10:`, la segunda parte (`n/divisor > 10`) solo se evaluará si `divisor` no es 0, evitando así una división por cero.

Para mejorar la legibilidad, es recomendable **usar paréntesis** cuando combinamos varias condiciones, incluso si Python no los requiere, especialmente en expresiones largas o con mezcla de `and` y `or`. Por ejemplo:

```
if (edad > 12 and edad < 20) or tiene_carnet_estudiantil:
    print("Tiene derecho a descuento juvenil")
```

Así queda claro que la primera parte `(edad > 12 and edad < 20)` es una subcondición compuesta cuyo resultado se combina con `tiene_carnet_estudiantil` mediante un `or`. Esto ayuda a evitar ambigüedades sobre el orden de evaluación.

En resumen, las *subcondiciones* nos permiten expresar lógica más compleja dentro de un mismo `if`. Usadas correctamente, hacen el código más compacto y evitan tener que anidar múltiples `if`. Sin embargo, si una condición compuesta se vuelve demasiado compleja, podría ser señal de que conviene refactorizar el código (por ejemplo, dividiéndola en condiciones más simples o utilizando variables intermedias con nombres descriptivos para cada subcondición).

Manejando condiciones de borde (casos límite)

Las **condiciones de borde** (o *casos límite*) son aquellas situaciones extremas o fronterizas en las que el comportamiento del programa podría diferir del caso "normal". Al diseñar las condiciones en nuestros programas, es fundamental tener en cuenta estos casos especiales para evitar errores lógicos. En otras palabras, debemos preguntarnos: *¿qué ocurre en el límite de mi condición?*

Por ejemplo, supongamos que un programa debe aceptar solo números positivos. ¿Qué pasa si el usuario ingresa exactamente 0? El 0 no es positivo, pero tampoco es negativo; es un caso límite que debemos manejar. Quizás nuestra condición de validación deba ser `if numero <= 0:` en lugar de `if numero < 0:`, dependiendo de si queremos excluir al cero. Del mismo modo, si una aplicación ofrece un descuento para edades "menores de 18 años", debemos decidir si a los 18 justo se aplica o no, y asegurarnos de que la condición (`edad < 18` o `edad <= 18`) refleje correctamente esa decisión.

Manejar condiciones de borde implica anticipar **valores extremos, valores inesperados o situaciones límite** y tratarlos explícitamente en el código. Algunos ejemplos comunes de casos borde son: listas vacías, valores nulos (`None` en Python), cadenas vacías, números muy grandes o muy pequeños, divisiones por cero, etc. Es buena práctica **documentar y probar** estos casos borde [22]. Por ejemplo, si escribimos una función que calcula la raíz cuadrada de un número, debemos considerar el comportamiento para números negativos (posiblemente devolviendo un error o un valor especial).

Un enfoque frecuente es incluir condicionales que capturen estos escenarios excepcionales. Por ejemplo:

```python
def calcular_raiz_cuadrada(x):
    if x < 0:
        print("Error: no se puede calcular la raíz de un número negativo")
        return None
    # ... continuar con el cálculo para x >= 0
```

Aquí añadimos una verificación inicial para un caso borde (x negativo) antes de proceder al caso general. Esto se conoce como un "guard clause" o condición de guarda, y ayuda a manejar salidas tempranas ante entradas fuera de lo esperado.

Otro caso de condiciones de borde ocurre en bucles: por ejemplo, evitar un bucle infinito estableciendo claramente la condición de finalización. Si programamos un bucle `while`, debemos asegurarnos de que eventualmente la condición se vuelva falsa o de incluir una instrucción `break` cuando se alcance cierta situación límite. **Nunca** debe diseñarse un bucle sin condición de salida; de lo contrario, el programa podría quedarse ejecutándose para siempre [23].

En resumen, para cada condición que escribamos en un programa, conviene preguntarnos: *¿Qué pasa en el límite?* y *¿hay algún valor o situación que rompa mis supuestos?* Manejando adecuadamente las condiciones de borde, nuestros programas serán más robustos y confiables incluso en escenarios extremos o entradas inválidas.

Evaluación de una expresión

En una sentencia condicional, la **expresión** que se coloca tras la palabra clave `if` (o tras `elif`) es evaluada para determinar si es verdadera o falsa. Esta evaluación sigue las reglas de *verdad* de Python: muchas veces la expresión involucra operadores de comparación (==, !=, <, >, <=, >=) que producen un resultado booleano `True` o `False` explícito [24] [25]. Por ejemplo, `5 == 5` evalúa a `True`, mientras que `5 != 5` evalúa a `False`. Las expresiones booleanas pueden también combinarse con los operadores lógicos ya mencionados (`and`, `or`, `not`) y con paréntesis para formar condiciones más complejas, cuyo resultado final igualmente será un valor booleano.

Es importante entender que **casi cualquier expresión en Python tiene un "valor de verdad"** incluso si no es explícitamente un booleano. Python considera ciertos valores como *falsos* en un contexto booleano, por ejemplo: el número 0, el valor `None`, la cadena vacía `""`, colecciones vacías (listas `[]`, tuplas `()`, diccionarios `{}`, etc.) se tratan como `False`. En cambio, cualquier valor numérico distinto de cero, cadena no vacía o colección con elementos se considera `True` al evaluar una condición [26]. Así, podemos escribir condicionales como `if nombre:` para verificar si la cadena `nombre` no está vacía (porque si está vacía se considera False y no entrará al if). Aunque esta flexibilidad existe, es recomendable usar explícitamente comparaciones o condiciones claras, ya que puede haber sutilezas. Por ejemplo, `if lista:` verifica si la lista tiene elementos (True si no está vacía); algunos programadores prefieren escribir `if len(lista) > 0:` para mayor claridad, aunque ambos funcionan igual.

Durante la evaluación de una condición, Python seguirá las **reglas de precedencia de operadores** (primero se evalúan comparaciones, luego `not`, luego `and`, luego `or`, etc.). Si no estamos seguros del orden, siempre podemos usar paréntesis para hacer la evaluación explícita.

Veamos un ejemplo de evaluación paso a paso. Supongamos el código:

```
a = 5
b = 3
c = 8
if a > b or c < a and not (b == 3):
    print("La condición es verdadera")
else:
    print("La condición es falsa")
```

Para determinar qué mensaje se imprimirá, Python evalúa la expresión condicional siguiendo la precedencia: primero evalúa `not (b == 3)`. `b == 3` es True (ya que b es 3), así que `not (b == 3)` resulta False. Luego evalúa `c < a` (esto es `8 < 5`, lo cual es False). Ahora la expresión queda `a > b or (False and False)`. Como el operador `and` se evalúa antes que `or`, resolvemos `False and False` primero, que da False. Finalmente tenemos `a > b or False`; `a > b` es `5 > 3`, que es True, así que queda `True or False`, lo cual es True. La condición global es verdadera, por lo tanto se ejecutará el bloque del `if` imprimiendo "La condición es verdadera". Este proceso ilustra cómo Python decide el flujo según la evaluación booleana de la expresión condicional.

En resumen, la *evaluación de una expresión* condicional implica convertir esa expresión a un resultado booleano. Como programadores, debemos asegurarnos de construir correctamente la expresión (usando los operadores adecuados) y de entender su valor de verdad en todos los casos posibles. Esto incluye conocer la diferencia entre usar `==` **para comparar** y `=` **para asignar**, reconocer valores "truthy" o "falsy", y tener claras las condiciones especiales que podrían afectar el resultado (por ejemplo, divisores cero, índices fuera de rango, etc.). Si la expresión se evalúa a True en tiempo de ejecución, Python ejecutará el bloque bajo el `if`; si es False, no lo hará (o pasará a un posible `elif` o `else` siguiente).

Creando un programa con condiciones

Veamos cómo aplicar todo lo anterior en la construcción de un programa simple. Imaginemos un problema sencillo: queremos pedir al usuario que ingrese un número y verificar si ese número es positivo. Si el número ingresado es negativo, le mostraremos una advertencia porque no era eso lo que se pidió; finalmente, siempre mostraremos el número que introdujo el usuario.

Podemos comenzar diseñando un **diagrama de flujo** para este problema:

- Inicio
- **Entrada:** pedir "Escriba un número positivo" y leer el número
- **Decisión (rombo):** ¿El número es menor que 0?
- Si **Sí** (True): **Proceso:** mostrar "¡Le he dicho que escriba un número positivo!"
- Si **No** (False): (no hacer nada especial)
- **Proceso final:** mostrar "Ha escrito el número X" (donde X es el valor ingresado)
- Fin

Ahora traduzcamos ese diagrama a código Python usando condicionales:

```
numero = int(input("Escriba un número positivo: "))
if numero < 0:
```

```
    print("¡Le he dicho que escriba un número positivo!")
print(f"Ha escrito el número {numero}")
```

En este código, primero se solicita al usuario un número y se convierte a entero con `int()`. Luego, la sentencia `if numero < 0:` comprueba la condición "¿el número es negativo?". Si esta condición es verdadera (por ejemplo, el usuario ingresó -5), el programa ejecuta el bloque indentado y muestra la advertencia correspondiente [27]. Si la condición es falsa (por ejemplo, el usuario ingresó 5), entonces el bloque del `if` se omite y no se muestra ninguna advertencia [28]. En ambos casos, después del `if`, el programa continúa y ejecuta la siguiente línea no indentada, que imprime el número introducido.

Ejemplo de ejecución 1 (número negativo):

```
Escriba un número positivo: -5
¡Le he dicho que escriba un número positivo!
Ha escrito el número -5
```

Aquí la condición `numero < 0` se evaluó como True (porque -5 < 0) [11], por eso se ve el mensaje de advertencia antes de la última línea.

Ejemplo de ejecución 2 (número positivo):

```
Escriba un número positivo: 5
Ha escrito el número 5
```

En este caso la condición fue False (5 no es < 0) [28], por lo que el bloque dentro del `if` no se ejecutó y el programa pasó directamente a imprimir el número.

Este pequeño programa ilustra cómo usar una estructura condicional para validar la entrada del usuario y manejar un caso especial (un número fuera del rango esperado). Hemos seguido buenas prácticas de estilo en Python: la condición termina con `:` y el mensaje de advertencia está indentado con 4 espacios bajo el `if`. También empleamos una **f-string** (`f"Ha escrito el número {numero}"`) para mostrar el valor de la variable de forma clara.

Cuando partimos de un diagrama de flujo, la traducción a código suele ser directa: cada **rombo** se convierte en un `if` (o `if...else` según corresponda), y los **bloques** de acciones secuenciales se convierten en bloques indentados bajo la condición adecuada. Es útil comentar el código o utilizar nombres de variables descriptivos para reflejar la lógica del diagrama de flujo, lo que facilita entender el código fuente comparándolo con el diseño inicial.

Manejando múltiples condiciones con `elif`

Hemos visto cómo funciona una estructura `if` simple y una `if...else` con dos caminos. Pero ¿qué pasa si tenemos **más de dos posibles condiciones o casos**? Por ejemplo, supongamos que un estudiante puede obtener calificaciones "Aprobado con Excelencia", "Aprobado", o "Reprobado" dependiendo de su nota numérica. Podemos necesitar chequear múltiples rangos: si la nota es mayor o igual a 90, o si es mayor o igual a 60, etc. En lugar de anidar varios `if...else` uno dentro de otro,

Python ofrece la cláusula `elif` (contracción de *else if*) para manejar múltiples condiciones secuenciales de forma más limpia.

La sintaxis general de una cadena de condicionales con `elif` es:

```
if condición1:
    # código si condición1 es True
elif condición2:
    # código si condición1 fue False y condición2 es True
elif condición3:
    # código si las anteriores fueron False y condición3 es True
...
else:
    # código si ninguna de las condiciones anteriores fue True
```

Solo una de todas las ramas posibles se ejecutará: la primera cuyo condicional resulte verdadero; las demás serán saltadas y la estructura condicional terminará en ese punto [29]. Si ninguna condición resulta verdadera, entonces se ejecutará el bloque final `else` (si existe) [30] [31]. Cabe destacar que el bloque `else` es opcional; podemos terminar con un `elif` si no necesitamos un caso "por defecto". No hay límite teórico en el número de cláusulas `elif` que se pueden encadenar, aunque en la práctica un número muy grande de ellas podría indicar la necesidad de replantear la lógica o usar estructuras de datos (por ejemplo, un diccionario de casos).

Veamos un ejemplo con `elif`. Supongamos un sistema simplificado de evaluación de notas:

```
calificacion = 85

if calificacion >= 90:
    print("Aprobado con Excelencia")
elif calificacion >= 60:
    print("Aprobado")
else:
    print("Reprobado")
```

En este código, evaluamos secuencialmente: primero si la calificación es 90 o más; si no (False), luego si es 60 o más; si tampoco, entonces cae en el `else`. Con una calificación de 85, la primera condición (`>= 90`) es falsa, la segunda (`>= 60`) es verdadera, por lo que se ejecutará el bloque del `elif` y el programa imprimirá "Aprobado" [32]. No se considera el `else` porque ya se encontró una condición verdadera y la cadena condicional finaliza en ese punto. Es importante señalar que si hubiéramos usado **dos** `if` **separados** en lugar de `elif` en este caso, ambas condiciones que resultaran verdaderas podrían ejecutarse, lo cual *no* es el comportamiento deseado. Por ejemplo, con calificación 85, un primer `if calificacion >= 80:` imprimiría "Aprobado con Excelencia" y un segundo `if calificacion >= 60:` independiente **también** imprimiría "Aprobado", dando dos mensajes. En cambio, usando `elif`, nos aseguramos de que solo una rama se ejecute [33].

Una buena práctica al usar múltiples `elif` es **ordenar las condiciones de la más específica a la más general** [34]. Esto significa que conviene poner primero la condición que abarca el caso más restringido o crítico. En el ejemplo anterior, verificamos `>= 90` antes que `>= 60` porque los que cumplen 90

también cumplen 60; si invirtiéramos el orden y preguntáramos primero `>= 60`, esa rama capturaría a **todos** los alumnos con 60 o más (incluyendo los de 90), y nunca se llegaría a evaluar la de 90. Por lo tanto, el orden correcto nos permite distinguir correctamente entre "excelencia" y un simple "aprobado". Si tus condiciones no son mutuamente excluyentes, debes tener mucho cuidado con el orden en que las evalúas.

Otra cosa a tener en cuenta es la legibilidad: una cascada larga de `elif` puede ser correcta, pero si empieza a crecer demasiado podría ser difícil de leer. En algunos casos, quizás convenga usar estructuras más avanzadas (como diccionarios mapeando casos a acciones, o la nueva sintaxis `match-case` introducida en Python 3.10) para manejar múltiples alternativas de forma más declarativa. No obstante, para rangos numéricos o decisiones encadenadas relativamente simples, `if/elif/else` es totalmente adecuado.

Manejando condiciones anidadas

Las **condiciones anidadas** ocurren cuando colocamos una estructura condicional (`if`, `if...else`, etc.) *dentro* del bloque de otra estructura condicional. Esto permite expresar lógica donde se requiere una segunda decisión *solo después* de haber pasado una primera condición. Por ejemplo, imaginemos un sistema de autenticación donde primero verificamos si un usuario está registrado, y solo si es así procedemos a verificar su contraseña. Esa lógica podría expresarse con condicionales anidados:

```
if usuario_existe:
    if password_correcta:
        print("Acceso concedido")
    else:
        print("Contraseña incorrecta")
else:
    print("El usuario no existe")
```

En este caso, el segundo `if` (el de la contraseña) solo se evalúa si el primero (`usuario_existe`) fue verdadero. Vemos que la rama "else" exterior cubre el caso en que el usuario ni siquiera existe, y la rama "else" interior cubre el caso de un usuario válido pero contraseña inválida. Esta estructura anidada refleja un árbol lógico donde primero se toma una decisión y *dentro de esa* puede surgir otra decisión adicional.

Otro ejemplo común es el de validar múltiples condiciones paso a paso. Por ejemplo, para registrar a una persona en un sistema podríamos querer chequear en orden: 1) que el email tenga formato válido, 2) que la contraseña cumpla cierta longitud, 3) que la edad sea mayor a cierta edad, 4) que haya aceptado los términos. Cada verificación puede anidarse dentro de la anterior, de modo que solo si la anterior fue satisfactoria continuamos con la siguiente:

```
if "@" in email and "." in email:
    print("✓ Email válido")
    if len(password) >= 8:
        print("✓ Contraseña suficientemente larga")
        if edad >= 18:
            print("✓ Edad válida")
            if terms_accepted:
```

```
                print("✓ Términos aceptados")
                print("Registro completado exitosamente")
            else:
                print("✗ Debe aceptar los términos")
        else:
            print("✗ Debe ser mayor de 18 años")
    else:
        print("✗ Contraseña muy corta (mínimo 8 caracteres)")
else:
    print("✗ Email inválido")
```

En este código, cada nivel de indentación representa un nivel de anidamiento, es decir, una condición dentro de otra condición [35] [36]. Solo si se pasa la validación del email, se revisa la contraseña; solo si la contraseña es válida, se revisa la edad; y así sucesivamente. Si en algún punto falla una condición, las ramas `else` correspondientes producen un mensaje de error y no se siguen evaluando las condiciones anidadas más profundas.

Las condicionales anidadas son muy poderosas, pero **abusar de ellas puede volver el código difícil de leer** [37]. A medida que aumenta el nivel de anidamiento (muchos `if` dentro de `if` dentro de `if`...), el código va quedando con mucha indentación hacia la derecha, lo cual complica seguir el flujo lógico. En general es buena idea evitar anidar más de 2 o 3 niveles profundos, si es posible [37] [38]. A veces podemos **simplificar condicionales anidadas usando operadores lógicos** para combinarlas en una sola condición cuando la lógica lo permite [38]. Por ejemplo, el caso de comprobar `0 < x < 10` visto antes se podía hacer con dos `if` anidados, pero es más claro usar `if 0 < x and x < 10:` en una sola línea [39] [40].

Otra técnica para evitar anidamientos profundos es utilizar **retornos anticipados o** `break` en vez de else redundantes, en contextos como funciones o bucles. Por ejemplo, podríamos reescribir el pseudocódigo de autenticación de usuario de forma más plana: primero `if not usuario_existe: print("no existe"); return`. Luego fuera de ese if, verificar la contraseña sabiendo que el usuario existe. De este modo evitamos un nivel extra de indentación. Esto se conoce como "guardar condiciones de salida temprano", y aunque no es exactamente una sentencia condicional distinta, sí es una manera de reorganizar la lógica para mejorar la legibilidad.

En resumen, **se pueden anidar tantos** `if` **como se necesite** (no hay un límite estricto) [41], pero siempre pregúntate si hay una forma más clara de expresar la misma lógica. Si usas anidamiento, procura que esté justificado por la naturaleza del problema (como en el ejemplo de validación por etapas) y considera añadir comentarios para aclarar cada nivel, en especial cuando las ramas comienzan a entrelazarse. Y si encuentras que tu código tiene muchos niveles de indentación, tal vez sea hora de refactorizar, dividiendo en funciones auxiliares o usando operadores lógicos para aplanar algunas condiciones.

Manejando condiciones de salida

En el contexto del control de flujo, una **condición de salida** se refiere típicamente a la condición que determina cuándo terminar un bucle o cuándo finalizar prematuramente una rutina. Aunque las condiciones de salida están más asociadas a bucles que a sentencias `if` simples, vale la pena mencionarlas porque son esenciales para escribir bucles correctos y evitar problemas como ciclos infinitos.

Cuando utilizamos bucles `while`, definimos una condición que debe permanecer verdadera para que el bucle siga ejecutándose; en consecuencia, la **condición de salida** (o de terminación) es la condición opuesta, aquella que hará que el bucle se detenga. Por ejemplo, consideremos este bucle que cuenta regresivamente:

```
contador = 5
while contador > 0:
    print(contador)
    contador -= 1
print("¡Despegue!")
```

La condición de salida aquí es *"cuando* `contador` *ya no sea mayor que 0"*. Una vez que `contador` llegue a 0, el `while` dejará de iterar. Es fundamental asegurarse de que, dentro del bucle, algo modifique las variables involucradas para que eventualmente la condición se vuelva falsa. En el ejemplo, la línea `contador -= 1` garantiza que el contador irá decreciendo y eventualmente romperá la condición `contador > 0`. Si olvidáramos esa línea, el bucle sería infinito porque la condición nunca cambiaría.

En otros casos, utilizamos la sentencia `break` dentro de un `while` (o `for`) para *salir inmediatamente* del bucle cuando se cumpla cierta condición. Por ejemplo, leer entradas del usuario hasta que ingrese "fin":

```
while True:
    texto = input("Escribe algo (o 'fin' para salir): ")
    if texto == "fin":
        break  # condición de salida: el usuario escribió 'fin'
    # ... procesar el texto ...
```

Aquí implementamos un bucle "infinito" (`while True:`) pero con una condición de salida interna: cuando `texto == "fin"`, usamos `break` para salir. La condición de salida, por tanto, es *texto igual a "fin"*. Este es un patrón común cuando no sabemos de antemano cuántas iteraciones habrá: definimos un `while True` y dentro controlamos la salida con un `if` + `break`. **Siempre** que hagas esto, asegúrate de que el `break` realmente pueda ejecutarse en algún momento; de lo contrario, habrás creado un bucle infinito.

Para un diseño robusto de bucles, ten en cuenta las siguientes prácticas recomendadas:

- **Establece claramente la condición de salida**: Ya sea en la propia cláusula del `while` o mediante un `break`, debe quedar claro bajo qué circunstancias termina el ciclo. Por ejemplo, en un bucle de búsqueda podrías tener `while not encontrado:` como condición, y dentro actualizar la variable `encontrado` a True cuando halles lo buscado.
- **Evita condiciones de bucle que nunca cambian**: Si utilizas variables en la condición, asegúrate de actualizarlas dentro del bucle. Si la lógica de salida es más compleja (por ejemplo, múltiples condiciones), puedes combinarlas con `and` / `or` en el `while`, o usar `break` cuando corresponda.
- **Prevé una salida de emergencia**: En algunos casos, aunque esperes que el bucle termine normalmente, es buena idea tener un contador máximo de iteraciones por seguridad. Por ejemplo, si esperas que cierto proceso termine en menos de 100 intentos, podrías incluir algo

como `intent += 1` y `if intent >= 100: break` para evitar potenciales loops infinitos en caso de condiciones inesperadas.

Un caso especial de *condición de salida* ocurre en la recursión (funciones que se llaman a sí mismas). Allí se le llama **caso base**: es la condición en la que la función deja de llamarse recursivamente a sí misma. Por ejemplo, en el cálculo factorial, la condición de salida es "si n <= 1, devolver 1 (y no llamar recursivamente)". Aunque esto se sale un poco del tema de condicionales simples, es útil reconocer que en cualquier estructura repetitiva (sea un bucle o recursión) debemos planificar cómo y cuándo saldrá.

En conclusión, manejar condiciones de salida implica garantizar que **todo ciclo tenga un final previsto**. Es parte de las buenas prácticas de programación vigilar por condiciones de salida claras para evitar bucles infinitos y garantizar que nuestros programas eventualmente terminen o pasen a la siguiente tarea [23] . Al escribir un bucle, piensa: *"¿bajo qué circunstancia exacta debo detener este ciclo?"* y luego implementa esa lógica con una condición en el `while` o con un `break` en un `if` interno. Esto hará tus algoritmos más seguros y tu código más fácil de entender.

Utilizando convenciones de nombres en variables (Snake Case)

La legibilidad del código no solo depende de su lógica, sino también de cómo está escrito y formateado. Python sigue unas guías de estilo (conocidas como **PEP 8**) que recomiendan, entre otras cosas, cómo nombrar las variables y otras entidades. La convención principal para nombres de variables (y funciones) en Python es usar **snake_case**, también conocido como "minusculas_con_guiones_bajos" [42] . Esto significa que los nombres se escriben *todo en minúsculas* y, si constan de varias palabras, se separan con el carácter de subrayado `_` en lugar de usar espacios o mayúsculas. Por ejemplo:

- `contador` (correcto, una sola palabra en minúsculas)
- `total_factura` (correcto, dos palabras separadas por guion bajo)
- `NombreUsuario` (**no** recomendado para variable, parece CamelCase de clase)
- `totalFactura` (**no** recomendado, estilo camelCase no se usa para variables en Python)
- `TOTAL` (válido pero por convención se suele reservar para constantes en mayúsculas)

En Python, por convenio: - **Variables y funciones** se nombran en *snake_case* (minúsculas_con_guiones_bajos) [42] . Ej: `edad_usuario`, `calcular_total()`.
- **Clases** se nombran en *PascalCase* (también llamado CamelCase con mayúscula inicial), dondeCadaPalabraEmpiezaEnMayúscula sin guiones bajos [43] . Ej: `ClaseUsuario`, `CalculadoraCientifica`.
- **Constantes** (valores que no deberían cambiar) se escriben con *MAYÚSCULAS* y guiones bajos (UPPERCASE_SNAKE_CASE). Ej: `MAX_INTENTOS = 5`, `PI = 3.1416` [43] .

Adherirse a estas convenciones hace que tu código sea más fácil de leer para otros desarrolladores y para ti mismo. Por ejemplo, si ves una variable llamada `numero_de_clientes`, inmediatamente sabes que es una variable común. Si ves `NumeroDeClientes`, un programador Python podría confundirse pensando que es el nombre de una clase (ya que empieza con mayúscula). Y si ves `NUMERO_DE_CLIENTES`, entenderás que seguramente es una constante global. Esta consistencia en estilo actúa como un lenguaje visual que transmite información sin necesidad de comentarios extra.

Además de la convención snake_case, existen otras **buenas prácticas de nomenclatura** que vale la pena seguir [44] [45]:

- Los nombres de variables deben ser **descriptivos y claros** sobre lo que representan [44]. Por ejemplo, es mejor `total_compra` que solo `t`. Evita abreviaturas confusas; es preferible un nombre más largo pero claro.
- No uses palabras reservadas de Python como nombres de variable (p.ej. `if = 5` es inválido porque `if` es palabra clave). Si necesitas usar un nombre parecido, modifícalo, por ejemplo `mi_if` o `if_flag` [46].
- Evita caracteres especiales o acentos en los identificadores. Usa solo letras (a-z, A-Z), números y `_`. Y recuerda que no pueden comenzar con número [47] (por ejemplo, `2variable` es inválido, debería ser algo como `variable2`).
- Python es *case-sensitive*: distingue mayúsculas de minúsculas [48]. Entonces `edad` y `Edad` son variables distintas (no uses esto a tu favor, puede causar confusión; mejor mantén un estilo consistente en minúsculas).
- Sigue las convenciones de estilo de espacios: alrededor de operadores como `=` u `+` pon un espacio (ej: `a = b + 1`, no `a=b+1`), y después de comas también un espacio [49]. Aunque esto no es "nomenclatura" de variables, sí es parte de PEP8 y contribuye a la limpieza del código.

En el contexto de snake_case, un detalle: a veces surgen nombres con siglas o acrónimos. Por convención, incluso esas se suelen poner en minúscula. Por ejemplo, para contar usuarios VIP (Very Important Person), se preferiría `numero_usuarios_vip` en lugar de `numero_usuarios_VIP`. La PEP8 recomienda que incluso los acrónimos se traten como palabras normales (todo minúscula) cuando son parte de un nombre de variable. Sin embargo, en algunos casos verás código con mayúsculas para acrónimos largos por legibilidad (`id_USA` vs `id_usa`). Lo importante es ser consistente en un mismo proyecto.

¿Por qué es importante todo esto? Porque el **código se lee muchas más veces de las que se escribe**. Un código con buenas convenciones es autoexplicativo en gran medida. Si declaramos `limite_velocidad = 120`, cualquier lector entiende que es un límite de velocidad, posiblemente en km/h. En cambio, un nombre críptico como `lv` obligaría al lector a buscar dónde se asigna o a adivinar su significado. Usar snake_case y nombres claros mejora la mantenibilidad.

Resumiendo, utiliza `snake_case` para tus variables y funciones en Python como parte de las buenas prácticas de estilo [42]. Esto significa minúsculas y guiones bajos, nombres significativos y sin abreviaturas innecesarias. Al adoptar estas convenciones de nomenclatura desde el inicio, te acostumbras a escribir código *pythónico* que otros podrán entender con facilidad y que cumple los estándares de la comunidad.

Fuentes: Las recomendaciones anteriores se basan en la guía de estilo PEP 8 de Python [42], en adaptaciones didácticas como la de freeCodeCamp [4] y en las prácticas generales promovidas en la comunidad Python. Cada punto del desarrollo de condicionales ha sido complementado con referencias a documentación y tutoriales reconocidos, como *Python para Todos* de Charles R. Severance (traducción al español) [9] [38], el tutorial de *Mclibre* de Bartolomé Sintes Marco [50] [27], artículos de *freeCodeCamp* [1] [7], *El Pythonista* [37] [32], *Abby's Digital Cafe* [22] [23], entre otros. Estas fuentes proporcionan ejemplos y explicaciones detalladas sobre el uso de condicionales en Python, desde aspectos básicos de sintaxis hasta consejos de buenas prácticas y manejo de casos especiales. Hemos mantenido la integridad de las citas para que puedas consultar el material original si deseas profundizar aún más en cada tema.

---

1 4 6 7 30 31 Sentencia If Else de Python: Explicación de las sentencias condicionales.
https://www.freecodecamp.org/espanol/news/sentencia-if-else-de-python-explicacion-de-las-sentencias-condiciones/

2 Control de flujo condicional — Fundamentos de Programación en Python
https://www2.eii.uva.es/fund_inf/python/notebooks/04_Control_de_flujo_condicional/Control_de_flujo_condicional.html

3 8 9 18 19 20 24 25 26 29 38 39 40 PY4E-ES - Python para todos
https://es.py4e.com/html3/03-conditional

5 11 12 13 27 28 41 50 if ... elif ... else ... Python. Bartolomé Sintes Marco. www.mclibre.org
https://www.mclibre.org/consultar/python/lecciones/python-if-else.html

10 Sentencias Condicionales en Python: Control de Flujo | LabEx
https://labex.io/es/tutorials/python-control-program-flow-with-conditional-statements-in-python-585758

14 15 Aprende Sintaxis de la Declaración If-Else | Declaración If-Else en Python
https://codefinity.com/es/courses/v2/9ac87b53-133a-4974-8f1d-a9761888723b/4c60394a-be5d-4c29-8be7-288270f03776/cc63df14-1c0e-4791-937b-555c6cd8b4e1

16 17 Diagrama de flujo - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/Diagrama_de_flujo

21 Python Scouts - Palabras clave en Python: and
https://pythonscouts.com/python-and/

22 23 45 Condicionales y Bucles en Programación · Abby's Digital Cafe
https://adigitalcafe.com/articles/condicionales-y-bucles

32 33 34 35 36 37 ▷ if, elif y else en Python: Guía de Condicionales [+Ejemplos] 2025
https://elpythonista.com/if-elif-y-else-en-python-guia-de-condicionales-2025-ejemplos

42 43 49 Estilo de Código PEP 8 en Python: Guía y Formateo Automático | LabEx
https://labex.io/es/tutorials/python-apply-pep-8-code-style-in-python-585757

44 46 47 48 Rules for naming variables in Python
https://www.luisllamas.es/en/rules-for-naming-variables-python/