«Talento Tech»

Iniciación a la

Programación con Python

Clase 03

Buenos Aires Ciudad

Clase N° 3 | Condicionales I

Temario:

- Operadores lógicos y relacionales.
- Control de flujo: estructuras condicionales if y else.
- Comentarios en el código.

Objetivos de la clase

El principal objetivo de esta clase es introducir a las y los estudiantes en el uso de estructuras condicionales en Python, herramientas clave para tomar decisiones dentro de un programa. A través de la implementación de los bloques **if** y **else**, los estudiantes aprenderán a controlar el flujo de ejecución según las condiciones definidas.

Además, se presentarán los operadores lógicos y relacionales, fundamentales para realizar comparaciones y establecer condiciones en los programas. Así se comprenderá cómo combinar estas herramientas para resolver problemas prácticos y crear aplicaciones más dinámicas e inteligentes.

Finalmente, se destacará la importancia de los comentarios en el código como una práctica esencial para mejorar la legibilidad y facilitar el mantenimiento del programa. Al finalizar la clase, los y las estudiantes serán capaces de validar datos ingresados por las personas usuarias, integrar lógica condicional en sus programas y desarrollar soluciones más robustas y funcionales.

Te damos la bienvenida a tu jornada en TechLab 🎉

Al llegar, te encontrás con Mariana, la Gerente de Proyectos, quien te felicita por haber completado tu primera asignación en torno a las variables con éxito, y aprovecha para hacerte algunas sugerencias:

“Para que tu programa sea aún más robusto, me gustaría que trabajemos en validar los datos que ingresan los usuarios (por ejemplo, que los campos no queden en blanco) y agregar una lógica para indicar si un cliente es mayor o menor de edad.”

Para ello, tendrás que interiorizarte en el **concepto de condicionales** (estructuras de control de flujo) **y los operadores lógicos y relacionales.**

¡Manos a la obra!

Operadores lógicos y relacionales.

Operadores relacionales

Los **operadores relacionales** en Python, que también se conocen como operadores de comparación, sirven para algo clave: comparar valores. El resultado de esta comparación siempre es un valor booleano, es decir, **True** (verdadero) o **False** (falso), dependiendo de si la condición que estás evaluando es cierta o no.
Ahora, vayamos a lo importante: ¿qué operadores relacionales tenés a tu disposición en Python?

Operador	Significado	Descripción
==	Igual que	Verifica si el valor o los valores de dos expresiones son iguales.
!=	No igual que	Comprueba si dos valores son diferentes.
>	Mayor que	Evalúa si el valor de la izquierda es mayor que el valor de la derecha.
<	Menor que	Determina si el valor de la izquierda es menor que el valor de la derecha.
>=	Mayor o igual que	Verifica si el valor de la izquierda es mayor o igual al valor de la derecha.
<=	Menor o igual que	Comprueba si el valor de la izquierda es menor o igual al valor de la derecha.

Esto es súper importante, porque estos operadores son los que te permiten tomar decisiones en tus programas. Los vas a usar un montón cuando trabajes con estructuras como if, else y elif, o cuando necesites crear bucles con while o controlar iteraciones con for.

Veamos unos ejemplos para que te quede más claro cómo funcionan:

```
print(3 > 4)                # False
print(2 <= 4)               # True
print(2 != 22)              # True
```

```
print("Hola" == "hola")    # False
print("Carlos" < "Ada")    # False
```

Hasta acá, parece bastante sencillo, ¿no? Pero hay algo más interesante que vale la pena mencionar: ¿cómo compara Python las cadenas de caracteres?

Acá viene lo interesante. Las cadenas de texto se comparan carácter por carácter, siguiendo un orden basado en su codificación (ASCII o Unicode). Esto significa que Python va a ir chequeando cada carácter de la cadena, de izquierda a derecha, hasta que encuentra una diferencia o hasta que una de las cadenas se termine. Entonces, cuando usás operadores relacionales con cadenas de texto, Python sigue estas reglas:

- **== (igual que):** Verifica si dos cadenas son exactamente iguales.
- **!= (no igual que):** Comprueba si dos cadenas son diferentes.
- **> (mayor que), < (menor que):** Compara las cadenas según el valor de cada carácter. Por ejemplo, en ASCII, las letras mayúsculas (A-Z) tienen un valor menor que las letras minúsculas (a-z), lo que significa que, por ejemplo, "a" es mayor que "A".

Este tipo de comparación se llama **orden lexicográfico**. Básicamente, Python va carácter por carácter, desde el principio de las dos cadenas, comparando. Si encuentra una diferencia, ahí se detiene y decide cuál es "mayor" o "menor". Si las cadenas son idénticas hasta cierto punto pero una es más corta, entonces la más corta se considera "menor".

Vamos a ver un par de ejemplos para aclarar cómo se comportan las comparaciones de cadenas:

```
print("apple" < "banana")     # True
print("apple" > "Apple")      # True
print("apple" < "apples")     # True

print("python" == "python")   # True
print("Python" != "python")   # True

print("A" < "a")              # True
print("z" > "Z")              # True
```

Como ves, Python no sólo se fija en si las letras son iguales, sino también en si son mayúsculas o minúsculas, o si una cadena tiene más o menos caracteres. ¡Esto es súper útil cuando necesitás comparar nombres, palabras o cualquier tipo de texto en tus programas!

Operadores lógicos.

Los operadores lógicos en Python son clave cuando querés trabajar con múltiples condiciones al mismo tiempo. Imaginate que estás desarrollando una aplicación donde necesitás verificar varias cosas a la vez, como si un usuario está logueado y además tiene los permisos necesarios para acceder a cierta función. Es acá donde entran en juego los operadores lógicos, que te permiten combinar distintas afirmaciones y evaluarlas juntas. Al final, todo se reduce a si esas afirmaciones son **True** (verdaderas) o **False** (falsas).

En Python, tenés tres operadores lógicos principales: **and**, **or** y **not**. Cada uno tiene su propio comportamiento, pero todos ellos trabajan sobre expresiones que resultan en valores booleanos, es decir, True o False.

Estos operadores te permiten hacer evaluaciones complejas de una manera súper simple.

Vamos a ver cómo funciona cada uno:

Operador “and”.

El operador and es bastante directo: sólo devuelve True si todas las condiciones que estás evaluando son verdaderas. Basta con que una sola sea falsa para que el resultado total sea **False**. Es como decir: "quiero que todo lo que estoy evaluando cumpla con la condición".

Fijate en estos ejemplos:

```
resultado = (5 > 3) and (8 > 6)
# True, porque ambas condiciones son verdaderas.

resultado = (5 > 3) and (8 < 6)
# False, porque una de las condiciones es falsa.
```

En el primer caso, ambas comparaciones son verdaderas, entonces and devuelve **True**. Pero en el segundo ejemplo, aunque la primera parte (5 > 3) es verdadera, la segunda (8 < 6) es falsa, lo que hace que todo el resultado sea **False**.

Operador “or”.

El operador **or**, por otro lado, es un poco más relajado. Con él, alcanza con que una sola de las condiciones sea verdadera para que el resultado total sea **True**. Sólo va a devolver **False** si todas las condiciones son falsas. Es como decir: "si al menos una cosa cumple, está bien".

Mirá estos ejemplos:

```
resultado = (5 > 3) or (8 < 6)
# True, porque al menos una condición es verdadera.

resultado = (5 < 3) or (8 < 6)
# False, porque ambas condiciones son falsas.
```

En el primer caso, aunque la segunda condición es falsa, la primera (5 > 3) es verdadera, por lo que el resultado final es **True**. En el segundo ejemplo, las dos comparaciones son falsas, por lo que el resultado es **False**.

Operador “not”.

Finalmente, tenemos el operador **not**. Este es el que invierte el valor de verdad de una expresión. Si algo es **True**, **not** lo convierte en **False**, y viceversa. Es como un interruptor de luz: lo que está prendido lo apaga, y lo que está apagado lo prende.

Acá van algunos ejemplos para que lo veas en acción:

```
resultado = not (5 > 3)
# False, porque 5 > 3 es True, y not lo invierte.

resultado = not (5 < 3)
# True, porque 5 < 3 es False, y not lo invierte.
```

En el primer caso, aunque 5 es efectivamente **mayor que** 3 (lo que sería **True**), **not** lo da vuelta y lo convierte en **False**. En el segundo ejemplo, 5 **no es menor que** 3 (lo que sería **False**), pero **not** invierte el resultado y nos da True.

Estos operadores lógicos son esenciales para cualquier lenguaje de programación y Python los maneja de manera súper intuitiva. En tus futuros programas, vas a ver cómo te permiten tomar decisiones más complejas, desde validar varias condiciones hasta hacer que tu código responda de manera diferente dependiendo de múltiples factores.

Luis, como programador experimentado que es, te comparte unas tablas que usa para recordar qué resultado se obtiene con cada operador en cada caso:

Condición 1	Condición 2	Y (and)	O (or)
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

Condición	NO (not)
V	F
F	V

Estructuras de control.

Imaginá que estás manejando un auto. No sólo te limitás a acelerar o frenar, sino que también tomás decisiones según las señales de tránsito, girás cuando es necesario, y a veces, das vueltas por el mismo lugar varias veces (¡ojo con eso!). En la programación pasa algo parecido: las estructuras de control son esas herramientas que te permiten decidir por dónde ir, repetir una acción o tomar un desvío según las condiciones. Son el volante y los frenos de tu código.

Las estructuras de control en Python son clave porque te permiten tener un programa que no simplemente corre una lista de instrucciones de manera secuencial, sino que puede adaptarse a distintas situaciones. Dependiendo de las condiciones que se encuentren, podés hacer que el programa ejecute una parte del código o repita una acción varias veces.

El propósito principal de estas estructuras es justamente darle esa flexibilidad a tu programa para que pueda "tomar decisiones". En lugar de que todo el código se ejecute de forma lineal, las estructuras de control le dicen al programa cuándo detenerse, repetir algo o saltar a otro bloque de código dependiendo de lo que está ocurriendo en ese momento. Esto es lo que hace que un programa pueda, por ejemplo, responder de manera diferente según la entrada de un usuario, realizar tareas repetitivas automáticamente o procesar datos de manera más eficiente.

Con estas herramientas, vas a poder hacer que tu programa no sólo funcione, sino que también piense, decida y actúe por su cuenta, ajustándose a lo que necesite en cada momento. ¡El verdadero poder de la programación está en estas estructuras!

Control de flujo: estructuras condicionales

Imaginá que estás caminando por la calle y, de repente, comienza a llover. En ese momento, tenés que tomar una decisión rápida: si tenés paraguas, lo abrís; si no, salís corriendo para no mojarte. Bueno, en programación, las estructuras condicionales son algo muy parecido. Le dicen al programa qué hacer dependiendo de lo que esté ocurriendo en ese instante. En Python, usamos estas estructuras para que el código pueda tomar decisiones según lo que esté pasando o según los datos que reciba.

La forma más básica de hacer esto es con la palabra clave **if**. Con **if**, podés indicarle a Python que ejecute cierto bloque de código sólo si se cumple una condición. Si esa condición es verdadera (o sea, True), Python va a ejecutar el código que está indentado justo debajo del if. Si no se cumple (o sea, es False), Python va a ignorar ese bloque y seguirá con lo que venga después.

La estructura básica de un if es súper sencilla. Fijate en este ejemplo:

```
numero = 5

if numero > 0:
   print("El número es positivo.")
```

Acá lo que estamos haciendo es preguntarle a Python: "¿Es el número mayor que 0?" Si la respuesta es sí (o sea, True), se va a ejecutar la línea que está indentada justo después del if, que imprime "El número es positivo.". Si el número hubiera sido 0 o negativo, ese bloque de código se saltaría y Python seguiría con lo que venga después.

Esa simple verificación es como el cerebro del programa, que decide qué hacer según la información que tiene. ¡Es lo que hace que los programas sean flexibles y puedan adaptarse!

¿Cómo funciona exactamente?

Veamos más de cerca lo que pasa con un if:

1. **Condición:** Lo primero que necesita el if es una condición. Esto puede ser cualquier expresión que Python pueda evaluar como **True** o **False**. Por ejemplo, numero > 0 es una condición que le pregunta a Python si numero es mayor que 0.

2. **Dos puntos (:):** Estos dos puntos que aparecen al final de la línea if son muy importantes. Le dicen a Python que el bloque de código que sigue está relacionado con esa condición.

3. **Bloque de código indentado:** Todo el código que esté indentado debajo del if se va a ejecutar solo si la condición es True. Y ojo con esto: la indentación es obligatoria en Python.

Esa pequeña sangría al principio de la línea no es sólo por estilo; le dice a Python que esas líneas de código forman parte del bloque del if.

Vamos a poner otro ejemplo para que quede bien claro:

```
edad = 18

if edad >= 18:
   print("Sos mayor de edad.")
```

En este caso, estamos verificando si alguien tiene 18 años o más. Si la condición es True, Python va a imprimir "Sos mayor de edad". Si no, simplemente va a seguir con el resto del programa, sin hacer nada especial con esa línea.

Recordá esto. ¡Te va a servir para poder resolver el pedido que te hizo Luis en su correo!

Y esto es solo el principio. Ahora pronto vas a ver cómo podés manejar situaciones que no cumplen ninguna condición con **else**.

Estructura condicional if...else

Hasta ahora vimos cómo **if** le da la capacidad a tu programa de tomar decisiones cuando una condición es verdadera. Pero, ¿qué pasa cuando esa condición no se cumple? Ahí es donde entra en juego **else**. Mientras que **if se encarga de ejecutar un bloque de código solo si algo es True**, **else viene al rescate cuando esa condición resulta ser False**, permitiéndote hacer algo diferente en ese caso. Es como decir: "Si pasa esto, hacé tal cosa, y si no, hacé esto otro".

El **else** no necesita tener una condición propia. **Se ejecuta solo si el if que lo precede da como resultado False**. Imaginá que estás manejando y, si tenés combustible, seguís adelante, pero si no tenés, vas directo a cargar nafta. Esa es la idea detrás de if...else.

Veamos cómo funciona:

```
numero = -5

if numero > 0:
   print("El número es positivo.")
else:
   print("El número no es positivo.")
```

Acá, lo que está pasando es que le preguntamos a Python si numero es mayor que 0. Como en este caso numero es -5, la condición es False, entonces Python se saltea el bloque del if y salta directo al else, que imprime "El número no es positivo". Si el número hubiera sido positivo, el else ni siquiera se ejecutaría.

¿Por qué es tan útil else?

El else es esa segunda oportunidad que tenés para manejar lo que pasa cuando tu condición no se cumple. Es como tener un plan B. Al combinar if con else, podés crear programas mucho más flexibles, que reaccionen de manera diferente según las circunstancias. En lugar de simplemente ignorar las situaciones en las que la condición no se cumple, con else podés hacer que tu programa actúe de otra forma, cubriendo todos los posibles resultados.

Pensemos en otro ejemplo sencillo:

```
edad = 16

if edad >= 18:
   print("Sos mayor de edad.")
else:
```

```
   print("Sos menor de edad.")
```

Acá estamos haciendo que Python verifique si alguien es mayor o igual a 18. Si la respuesta es True, imprime "Sos mayor de edad", pero si es False, el programa no se queda sin hacer nada: imprime "Sos menor de edad". Así, el código no solo evalúa la condición, sino que también reacciona de manera diferente si la condición no se cumple.

¿Cuándo usar else?

El uso de else hace que tu código sea más claro y directo. En lugar de tener que escribir más if o manejar los casos por separado, else te permite cubrir todos los escenarios posibles en los que la condición no se cumpla, lo que hace que tu código sea más eficiente y fácil de leer. Además, le da al programa la capacidad de reaccionar de manera distinta según los datos que reciba o el estado en el que se encuentre.

Si combinás if con else, podés manejar cualquier situación que se te presente: desde validar formularios, hasta hacer que un juego responda de manera diferente según las decisiones del jugador. Todo depende de las condiciones que estés evaluando, y con else, tenés la tranquilidad de que nada va a quedar sin manejar.

El uso de else le da a tus programas una dimensión extra. Con esto, ya no se trata solo de ejecutar un bloque de código si algo es verdadero, sino también de decirle al programa qué hacer cuando no lo es. ¡Es como tener un plan B, listo para cualquier situación!

Veamos otro ejemplo con else. Supongamos que querés verificar si un producto tiene un precio menor a $500. Podrías hacer algo así:

```
precio = 450
if precio < 500:
   print("El producto es económico.")
else:
   print("El producto es costoso.")
```

Y, por supuesto, podés construir condiciones que usen operadores lógicos. Por ejemplo, si querés verificar si una persona puede votar en base a su edad y ciudadanía:

```
edad = 20
es_ciudadano = True

if edad >= 18 and es_ciudadano:
   print("Podés votar.")
else:
   print("No podés votar.")
```

Este ejemplo te muestra cómo combinar condiciones para evaluar múltiples criterios al mismo tiempo

Comentarios en el código: más allá del

En la clase anterior vimos que los comentarios en Python se crean usando el símbolo #. Los comentarios son esenciales para explicar partes del código, documentar decisiones importantes y hacer que el programa sea más fácil de entender, tanto para vos como para cualquier otra persona que pueda trabajar con ese código en el futuro.

Además de los comentarios simples que empiezan con #, Python permite usar comentarios multilínea, que sirven para explicar procesos o ideas más complejas que no entran en una sola línea. A continuación, repasamos ambos tipos:

1. Comentarios simples con

Como ya vimos, los comentarios simples se escriben anteponiendo el símbolo # a una línea o parte de una línea. Son ideales para anotar explicaciones breves o desactivar temporalmente líneas de código.

Ejemplo:

```
# Este programa calcula el área de un rectángulo
base = 5  # Base del rectángulo
altura = 10  # Altura del rectángulo
area = base * altura
print("El área es:", area)  # Mostramos el resultado
```

En este ejemplo, cada comentario aclara el propósito de las variables y la salida del programa, haciendo el código más fácil de entender.

2. Comentarios multilínea

Python no tiene un símbolo específico para comentarios multilínea, pero se pueden simular utilizando cadenas de texto de varias líneas (tres comillas simples o dobles). Aunque estas cadenas están diseñadas para definir textos largos dentro del código, se utilizan comúnmente como comentarios para explicar procesos más complejos.

Ejemplo:

```
"""
Este programa calcula el área de un rectángulo.
Primero define la base y la altura,
luego multiplica ambos valores para obtener el área.
Finalmente, muestra el resultado en pantalla.
"""
base = 5
altura = 10
area = base * altura
print("El área es:", area)
```

Si bien estas cadenas no son técnicamente comentarios (ya que Python las interpreta como texto), son ignoradas cuando no están asignadas a una variable, por lo que se comportan como comentarios en la práctica.

Los comentarios multilínea son útiles para documentar módulos, funciones o partes extensas del código. Y dado que estos comentarios son una cadena de texto, se pueden reemplazar las triples comillas dobles por triples comillas simples.

Buenas prácticas al usar comentarios.

Sé claro y preciso: Los comentarios deben explicar el propósito del código, no describir lo obvio. Por ejemplo, en lugar de:

```
x = 5  # Asignamos 5 a x
```

Es mejor:

```
x = 5  # Cantidad inicial de productos en stock
```

Actualizá los comentarios al modificar el código: Comentarios desactualizados generan confusión y contradicen el objetivo de facilitar la comprensión.

Usá comentarios para documentar decisiones importantes: Por ejemplo:

```
# Elegimos este algoritmo porque es más eficiente para listas pequeñas
```

Evitá comentarios redundantes: El código debe ser lo suficientemente claro como para no necesitar explicaciones obvias.

Los comentarios son una herramienta importante para escribir código comprensible y mantenible. Usar comentarios de manera adecuada no sólo mejora la legibilidad del código, sino que también facilita su mantenimiento y colaboración en proyectos.

Ejercicio práctico.

Luego de un día intenso en **TalentoLab** , y gracias a la ayuda inestimable de Luis, ya podés intentar resolver la tarea que te encargó Mariana. Cómo siempre, te escribió un correo explicando detalladamente que necesita que hagas:

¡Hola!

Necesito que desarrolles un pequeño programa en Python que haga exactamente lo siguiente:

- Solicite al cliente su nombre, apellido, edad y correo electrónico.
- Compruebe que el nombre, el apellido y el correo no estén en blanco, y que la edad sea mayor de 18 años.
- Muestre los datos por la terminal, en el orden que se ingresaron. Si alguno de los datos ingresados no cumple los requisitos, sólo mostrar el texto “ERROR!”.

¡Espero ver el resultado de tu trabajo pronto!

Saludos,
Mariana

Materiales y recursos adicionales

Artículos:

El libro de Python: Bloques de código y sintaxis en Python
Oregoom: If y else en Python
Programación.net: Condicionales en Python
Pythones: Aprende los condicionales en Python con ejemplos

Videos:
Tutoriales sobre ciencia y tecnología: Estructura if y estructura if…else

---

Preguntas para reflexionar

1. ¿De qué manera los operadores lógicos y relacionales amplían las posibilidades de resolver problemas en programación?
2. ¿Por qué creés que es importante escribir código que no sólo funcione, sino que también sea fácil de leer y mantener?
3. ¿Cómo pensás que el uso de estructuras condicionales puede integrarse con otras herramientas de programación para resolver problemas complejos?

Próximos pasos

En la próxima clase, vamos a profundizar en las estructuras condicionales, introduciendo herramientas más avanzadas como **elif** y **match**, que nos permitirán manejar múltiples escenarios en un solo bloque de código. Estas estructuras serán clave para tomar decisiones más complejas y optimizar nuestros programas.

También trabajaremos con cadenas de texto, explorando cómo acceder a caracteres específicos, concatenar textos, medir su longitud y aplicar métodos como **.lower()** o **.upper()** para transformar su formato. Aprenderemos a utilizar estas herramientas en conjunto con **f-Strings** para generar salidas claras y profesionales, mejorando la interacción con los usuarios.

Todo esto nos ayudará a continuar desarrollando programas más dinámicos y adaptados a las necesidades reales de posibles clientes.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad