«Talento Tech»

Iniciación a la Programación con Python

CLASE 5

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 5 | Condicionales

Temario:

- Cadenas: acceso a caracteres, concatenación, longitud, etc.
- Operadores lógicos.
- Control de flujo: condicionales (if, else, elif).

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Cadenas de caracteres.

Imaginemos que estás escribiendo un mensaje o tal vez guardando una lista de cosas que necesitás comprar en el almacén. Ese texto que escribís, en el mundo de la programación, se guarda en algo llamado ***cadena de caracteres*** o simplemente ***string***.
En **Python**, trabajar con estas cadenas es súper intuitivo porque ya desde el comienzo, con nuestras primeras funciones como **print()** e **input()**, hemos interactuado con ellas. Ahora, vamos a conocer un poco más sobre lo que podés hacer con estas cadenas, ¿cómo se manejan? y algunas características que las hacen especiales.

Las cadenas tienen algo importante que debemos entender: una vez que se crean, no se pueden modificar. Esto es lo que llamamos ***inmutabilidad***. Pero no te preocupes, aunque una cadena en **Python** no se pueda cambiar directamente, siempre podés generar una nueva que incluya esos cambios que necesitás. Además, las cadenas pueden tener cualquier longitud, desde estar completamente vacías hasta contener kilómetros de texto, ¡bueno, o lo que te permita la memoria de tu computadora!

Por ejemplo, mirá estos nombres:

```
empresa = "Tech Nology S.A."
director = "Martinez, Ana"
```

En este caso, hemos creado dos cadenas y las hemos guardado en las variables empresa y director. Si quisiéramos saber cuántos caracteres tiene cada una, Python tiene una herramienta genial para eso: la función **len()**.

Probá este código:

```
nombre = "Alberto Torres"
print(nombre, "contiene", len(nombre), "caracteres.")
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Este programa te va a mostrar cuántos caracteres tiene el nombre completo "Alberto Torres". Así que si alguna vez querés contar cuántas letras tiene una palabra o un texto, **len()** es tu mejor amiga.

Otra cosa que hace Python es permitirnos usar **comillas simples o dobles** para escribir cadenas. Esto es especialmente útil cuando querés incluir un tipo de comilla dentro del texto sin tener que hacer malabares. Fijate en este ejemplo:

```
cadena1 = "Hola mundo!"
cadena2 = 'Hola mundo!'
cadena3 = "Hola 'mundo'"
cadena4 = 'Hola, "mundo"'

print(cadena1) # Imprime Hola mundo!
print(cadena2) # Imprime Hola mundo!
print(cadena3) # Imprime Hola 'mundo'
print(cadena4) # Imprime Hola, "mundo"
```

Como ves, podés mezclar comillas simples y dobles según lo que necesites. Eso hace que trabajar con texto sea mucho más cómodo y evites errores cuando hay comillas en la misma oración.

Otro dato interesante es que las cadenas no sólo son texto simple. Pueden incluir saltos de línea **(\n)** o tabulaciones **(\t)**, algo que puede ser útil cuando necesitás organizar mejor tu salida de datos. Además, se pueden sumar (concatenar) usando el operador **+** o repetir usando el operador *****.

Imaginemos un ejemplo divertido con una risa:

```
risa = "Ja "
carcajada1 = risa + risa + "Jaaaaaa!"
print(carcajada1) # Imprime Ja Ja Jaaaaaa!

carcajada2 = risa * 4
print(carcajada2) # Imprime Ja Ja Ja Ja
```

Y si querés darle un poco de estilo visual a la salida, podés hacer algo como esto:

```
print("-"*20) # Imprime --------------------
```

Finalmente, algo súper interesante de las cadenas en **Python** es que podés acceder a cada uno de sus caracteres de manera individual. Cada letra o símbolo tiene una posición o índice, que empieza en cero. Así que, si tenés la palabra "Python", el primer carácter es **'P' (índice 0)** y el último es **'n' (índice 5)**.

Con estas herramientas, Python te permite manejar el texto de una forma muy práctica. Ya sea que estés creando un mensaje, una lista de productos o cualquier otra cosa, vas a tener la flexibilidad y el poder para hacerlo fácilmente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Cadenas e índices.

Hasta ahora, venimos viendo que las cadenas de texto en Python son súper útiles para representar nombres, mensajes o cualquier secuencia de caracteres. Pero hay algo muy interesante que todavía no exploramos en profundidad: cada letra, número o símbolo dentro de una cadena tiene una posición asignada. Esa posición se llama índice.

Y acá viene lo importante: en **Python**, los índices comienzan desde cero. Así que, cuando hablamos de la palabra "**Python**", el primer carácter, la letra '**P**', **está en la posición 0**, la '**y**' **en la posición 1**, y así sucesivamente hasta llegar a la '**n**' que **está en la posición 5**. ¡Parece un detalle menor, pero en realidad es fundamental cuando necesitás manipular texto de manera precisa!

Por ejemplo, si querés obtener un carácter específico de una cadena, todo lo que tenés que hacer es indicar el índice de ese carácter usando corchetes. Vamos a verlo en acción:

```
# Definimos una cadena
palabra = "Python"

# Accedemos al primer carácter (índice 0)
primer_caracter = palabra[0]
print("El primer carácter es:", primer_caracter)
# Imprime El primer carácter es: P

# Accedemos al cuarto carácter (índice 3)
cuarto_caracter = palabra[3]
print("El cuarto carácter es:", cuarto_caracter)
# Imprime El cuarto carácter es: h
```

¿Viste? Es como tener un pequeño control remoto que te permite ir directamente al lugar exacto de la cadena que necesitás. Si alguna vez tuviste que buscar una letra en un mensaje o reemplazar un símbolo en una contraseña, este truco te va a ahorrar mucho tiempo.

Este sistema de indexación no sólo es útil, es clave cuando empiezas a trabajar con textos más complejos. Te permite **seleccionar** y **manipular** cualquier parte de una cadena sin esfuerzo. Ahora podés elegir letras, generar subcadenas o incluso recorrer palabra por palabra dentro de un texto más largo. Es como si cada cadena fuera una lista de elementos a los que podés acceder con total libertad.

Slicing (rebanado) en cadenas.

A medida que vamos avanzando en el mundo de Python, nos encontramos con una herramienta súper poderosa llamada **slicing**, que en castellano podemos traducir como "rebanado". **El slicing te permite cortar porciones de una cadena de caracteres**, lo que sería algo así como seleccionar fragmentos específicos de texto. Lo interesante es que no sólo podés hacerlo con cadenas, más adelante vas a ver que esta técnica también se puede aplicar a listas y otros tipos de datos en Python.

Para hacer slicing, usamos corchetes **[ ]** y adentro indicamos desde qué índice queremos empezar a cortar, y hasta qué índice queremos llegar. Importante: el índice final que pongas no se incluye, Python va a cortar antes de llegar ahí. Es como decir: "cortame desde acá hasta acá, pero sin incluir el último punto".

Veamos un ejemplo concreto para que esto quede claro:

```
frase = "Python es asombroso"
subcadena = frase[0:6]
print(subcadena)   # Imprime Python
```

En este caso, hemos recortado la palabra "Python" de la frase original, empezando en el índice 0 y terminando justo antes del índice 6. La subcadena que obtuvimos es "Python", la guardamos en una variable, y luego la mostramos.

Pero el slicing no se queda sólo ahí. **Python te permite trabajar con índices positivos y negativos**, lo que le da aún más versatilidad. Los índices **positivos arrancan desde 0**, de izquierda a derecha, como lo hicimos recientemente. En cambio, los **índices negativos arrancan desde -1**, pero van de derecha a izquierda, empezando desde el final de la cadena.

Mirá cómo se ve esto en la práctica:

```
texto = "Python"

# Acceso utilizando índices positivos
print("Índice 0:", texto[0])  # P
print("Índice 1:", texto[1])  # y

# Acceso utilizando índices negativos
print("Índice -1:", texto[-1])  # n
print("Índice -2:", texto[-2])  # o
```

Así, con índices positivos o negativos, podés navegar por cualquier cadena como si tuvieras un control remoto para elegir los caracteres que necesites.

Una de las cosas más interesantes del slicing es que podés omitir el índice de inicio o el índice de fin, y Python automáticamente va a asumir que querés cortar desde el principio o hasta el final de la cadena, respectivamente. Esto te da mucha flexibilidad cuando trabajás con textos largos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Por ejemplo:

```
frase = "Aprender Python es divertido"

# Subcadena desde el inicio hasta el índice 8 (sin incluirlo)
subcadena = frase[:8]
print("Subcadena desde el inicio hasta el índice 8:",
subcadena)
# Imprime Aprender

# Subcadena desde el índice 9 hasta el final
subcadena = frase[9:]
print("Subcadena desde el índice 9 hasta el final:",
subcadena)
# Imprime Python es divertido
```

Pero, ¿qué pasa si te pasás con los índices? Tranquilo, Python es lo suficientemente inteligente como para no darte un error. Si pedís más de lo que hay, simplemente te va a devolver lo que pueda:

```
texto = "Hola"
subcadena = texto[1:10]
print("Subcadena obtenida:", subcadena)
# Imprime ola
```

Y todavía hay más. También podés hacer slicing con saltos entre caracteres, usando un tercer parámetro. Esto te permite, por ejemplo, tomar sólo cada segundo carácter de una cadena.

```
texto = "Talento Tech"
subcadena = texto[::2]
print("Subcadena obtenida:", subcadena)
# Imprime TlnoTc
```

Acá, con **::2**, le dijimos a Python que queremos tomar cada segundo carácter, ¡y lo hizo sin problemas!

El slicing es una herramienta muy poderosa para cuando necesitás manipular texto en Python. Con una simple y clara sintaxis, te permite obtener exactamente lo que querés de una cadena, sin errores innecesarios. Es una de las tantas razones por las que Python es tan popular entre los programadores de todo el mundo.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

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. 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.

Ahora, vayamos directo al punto: ¿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.

Estos operadores son como herramientas que te permiten tomar decisiones dentro de tu código.

Por ejemplo, imagina que tenés que hacer que tu programa reaccione de manera distinta dependiendo del valor que le dé el usuario. Estos operadores son los que van a hacer esa comparación entre lo que vos esperás y lo que te llega.

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 encuentre 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!

Buenos Aires aprende
Agencia de Habilidades para el Futuro

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 son fundamentales cuando estás tomando decisiones dentro de tu código, ya que 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.
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

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**.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

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. ¡Y lo mejor es que ya sabés cómo usarlos!

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. Estas estructuras son lo que le da vida a los programas, haciéndolos más dinámicos e interactivos.

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.

Las estructuras de control son la base de cualquier programa que tenga un poco más de complejidad. Permiten que el software sea capaz de manejar distintas entradas, que se adapte a diferentes escenarios y que pueda realizar tareas complejas de manera efectiva. En otras palabras, son como los cimientos de un edificio: sin ellas, no podrías construir nada sólido.

Primero, vamos a trabajar con estructuras condicionales. Estas son las que te permiten ejecutar ciertos bloques de código sólo si se cumplen determinadas condiciones. Acá es donde aparecen nuestras amigas **if**, **elif**, y **else**. Básicamente, lo que hacen es evaluar una condición, y según si es verdadera (**True**) o falsa (**False**), deciden qué parte del código

ejecutar. Es como si estuvieras evaluando constantemente: "¿Si pasa tal cosa, hago esto; si no, hago lo otro?".

Después de las condicionales, vamos a meternos con algo muy potente: los **bucles** o **loops**. Los bucles son ideales para cuando necesitás repetir una acción varias veces. Podés usar un bucle for cuando querés que algo se repita un número específico de veces, o un bucle while cuando querés que algo siga ocurriendo mientras se cumpla una condición. Son súper útiles cuando necesitás automatizar tareas que se repiten, recorrer colecciones de datos como listas o diccionarios, o hacer que una acción se repita hasta que una condición cambie.

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 inteligentes 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.

¿Por qué son tan importantes las estructuras condicionales?

Las estructuras condicionales son la clave para que un programa pueda tomar decisiones de manera dinámica. No importa si estás escribiendo una calculadora, un juego, o una aplicación web, siempre vas a necesitar que tu código reaccione de diferentes maneras según lo que esté ocurriendo. Las estructuras condicionales como **if** te permiten escribir programas interactivos y que se adapten a diferentes situaciones.

Por ejemplo, imagina un juego donde un jugador tiene que evitar obstáculos. El programa tiene que evaluar constantemente si el jugador está por chocar con algo, y si eso ocurre, ejecutar una acción, como restar puntos o terminar el juego. Esa lógica está construida con if.

Y esto es solo el principio. Muy pronto vas a ver cómo podés agregar más condiciones con elif o manejar situaciones que no cumplen ninguna condición con **else**. Pero, por ahora, quédate con esta idea: el if es el que le da "poder de decisión" a tu código.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

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!

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Estructuras condicionales anidadas

Hasta ahora, vimos cómo if y else te permiten tomar decisiones en tu programa, pero ¿qué pasa cuando necesitás evaluar más de una condición a la vez? Acá es donde entran en juego las estructuras condicionales anidadas. Este concepto puede sonar un poco complicado al principio, pero en realidad es bastante intuitivo: se trata de poner una condición dentro de otra. Esto te permite manejar situaciones donde la decisión que vas a tomar depende de una serie de condiciones que deben evaluarse en secuencia.

Cuando hablamos de anidar condiciones, simplemente estamos diciendo que vamos a incluir un if dentro de otro if, o dentro de un else. Esto te da la capacidad de realizar nuevas evaluaciones basadas en el resultado de una condición anterior. Es como si tu programa tuviera que pasar por varias puertas, una tras otra, donde cada puerta decide si debe seguir o detenerse.

Para que Python entienda qué bloque de código pertenece a qué condición, es fundamental la indentación. La indentación es esa sangría que ponés al principio de las líneas, y es lo que le dice a Python cuál es la jerarquía de las instrucciones. Cuanto más adentro esté el bloque de código, más profundo será el nivel de anidación.

Miremos un par de ejemplos para que quede más claro.

Ejemplo 1: Decisión basada en múltiples condiciones
Imaginá que querés escribir un programa que verifique si una persona puede conducir. Primero, chequeás si tiene la edad suficiente. Si es así, el programa debería verificar si la persona tiene licencia. Este tipo de lógica es esencial cuando necesitás evaluar múltiples criterios uno después del otro.

```
edad = 20
tiene_licencia = True

if edad >= 18:
    if tiene_licencia:
```

```
        print("Podés conducir.")
    else:
        print("No podés conducir porque no tenés licencia.")
else:
    print("No tenés la edad suficiente para conducir.")
```

Acá lo que pasa es que primero chequeamos si la persona tiene 18 años o más. Si eso es verdadero, entramos en el bloque de código del primer if. Una vez dentro, evaluamos una segunda condición: si tiene licencia. Si las dos condiciones son True, el programa va a imprimir "Podés conducir". Si tiene la edad pero no tiene licencia, el programa va a imprimir que no puede conducir por esa razón. Y si no tiene ni la edad suficiente, directamente va a saltar al else del primer bloque y no va a evaluar nada más.

Ejemplo 2: Evaluación de calificaciones
Supongamos que estamos creando un programa que evalúa una calificación y decide qué mensaje imprimir en base al resultado. Primero, verificamos si la persona aprobó. Si es así, después evaluamos en qué rango de calificaciones se encuentra para dar un feedback más preciso.

```
nota = 85

if nota >= 60:
    print("¡Aprobaste!")
    if nota >= 90:
        print("¡Excelente calificación!")
    elif nota >= 75:
        print("Muy buen trabajo.")
    else:
        print("Buen esfuerzo, pero hay margen de mejora.")
else:
```

```
    print("No alcanzaste la calificación mínima para aprobar.")
```

En este ejemplo, primero chequeamos si la calificación es suficiente para aprobar. Si lo es, se imprime el mensaje de aprobación, pero ahí no termina. Dependiendo de qué tan alta sea la nota, el programa va a imprimir un mensaje adicional. Si la nota es 90 o más, el programa va a decir que es excelente. Si está entre 75 y 89, va a felicitar al estudiante por el buen trabajo. Si es menor, va a sugerir que aún hay margen para mejorar.

¿Por qué son útiles las estructuras condicionales anidadas?

Este tipo de estructuras te permite manejar situaciones donde las decisiones no son tan simples. A veces no basta con evaluar una sola condición, y necesitás chequear más de una cosa antes de llegar a una conclusión. Las estructuras condicionales anidadas son justamente la herramienta que te permite hacer esto, y lo mejor es que Python, con su indentación, te facilita mucho la legibilidad del código.

Cuando usás estas estructuras, podés crear programas que manejen situaciones complejas de manera eficiente. Solo tenés que asegurarte de que cada bloque de código esté correctamente indentado, así Python sabrá exactamente qué condiciones deben evaluarse en cada paso.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicio práctico N° 1:

Control de inventario de una tienda de videojuegos-

Imaginá que estás ayudando a una tienda de videojuegos a organizar su inventario. El dueño te pide que escribas un programa que verifique si hay stock suficiente de un videojuego y, si no hay, que avise que hay que reponerlo.

El programa debería pedirle al usuario que ingrese la cantidad actual en stock y, en base a esa cantidad, mostrar si se necesita hacer un nuevo pedido o no.

Ejercicio práctico N° 2:

Compra con descuentos

Escribe un programa en Python que solicite al usuario el monto total de la compra y la cantidad de artículos que está comprando. El programa debe determinar el descuento aplicable según las siguientes reglas:

- Si la cantidad de artículos comprados es mayor o igual a 5 y el monto total es mayor a $10000, aplica un descuento del 15%.
- Si la cantidad de artículos comprados es menor a 5 pero mayor o igual a 3, aplica un descuento del 10%.
- Si la cantidad de artículos comprados es menor a 3, no se aplica descuento.

Al final, el programa debe imprimir el monto total de la compra después de aplicar cualquier descuento o simplemente el monto original si no hay descuento.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad