

Python

```
import numpy as np  
def train_model(data, epochs):  
    < for e in range(epochs):  
    < predictions = model.predict(data)  
    < loss = compute_loss(predictions, labels)  
    < model.update_weights(loss)  
    < if loss < 0.01:  
        break
```



Curso Completo de Python

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

lucasegaacademy.com

Curso Completo de Python

Autor: Esteban Gutiérrez

lucasegaacademy.com

*Guía completa con explicaciones extensas por unidad,
prácticas y examen final.*

ÍNDICE

PRÓLOGO	4
UNIDAD 1: Tus primeros pasos en Python	6
UNIDAD 2: Variables, tipos de datos y operadores	16
UNIDAD 3: Entrada de datos y manejo de texto	28
UNIDAD 4: Tu programa aprende a decidir	41
UNIDAD 5: Lógica y repetición: el poder de los ciclos	74
UNIDAD 6: Listas y tuplas - tu caja de herramientas de colecciones	116
UNIDAD 7: Diccionarios, Sets y Texto Avanzado - Tu Caja de Herramientas d... ..	161
UNIDAD 8: Funciones y código elegante: deja de repetirte y programa como	204
UNIDAD 9: Que nada te tumbe: errores, archivos y módulos	244
UNIDAD 10: Objetos, tu primer juego y tu graduación como programador ..	284
EXAMEN FINAL	323

PRÓLOGO

Bienvenido, bienvenida. Este libro no es un manual aburrido de los que se quedan juntando polvo en el estante: es tu compañero de ruta para convertirte en programador de Python de verdad. Lo escribí pensando en ti, que tal vez nunca has escrito una sola línea de código, o que ya intentaste aprender y te topaste con explicaciones frías, llenas de tecnicismos y sin un solo ejemplo que funcione en la vida real. Aquí las cosas son distintas: cada concepto viene con explicación clara, analogías de la vida cotidiana, ejemplos paso a paso que puedes correr en tu computadora, errores comunes que te voy a señalar antes de que los cometas, buenas prácticas que usan los profesionales, prácticas con soluciones explicadas y un examen final para que midas tu nivel con honestidad.

Python es el lenguaje de programación más popular del planeta, y no es casualidad. Se usa en inteligencia artificial, análisis de datos, desarrollo web, automatización de tareas, videojuegos, robótica y hasta en la NASA. Pero lo más importante para ti es esto: Python fue diseñado para que los humanos lo lean fácil. Su sintaxis es limpia, casi como escribir instrucciones en español. Eso no significa que sea un lenguaje "de juguete": detrás de esa sencillez hay una potencia brutal, con bibliotecas que hacen en cinco líneas lo que en otros lenguajes toma cincuenta.

Este libro está organizado en 10 unidades. Cada unidad agrupa tres clases del curso de video de Lucas EGA Academy y sigue siempre la misma estructura para que sepas qué esperar: primero una introducción que te pone en contexto y te dice por qué importa lo que vas a aprender; después la explicación detallada, que es el corazón de la unidad, con todo el fondo teórico, analogías, errores típicos y buenas prácticas; luego ejemplos paso a paso que analizamos línea por línea; después una sección de práctica con al menos cinco ejercicios y sus soluciones explicadas paso a paso; y al final un resumen de la unidad con los puntos clave que debes llevarte grabados.

Al final del libro encontrarás el examen final: 50 preguntas que mezclan opción múltiple, verdadero o falso y ejercicios de código abierto. Hazlo con honestidad, sin ver las respuestas antes. Las respuestas y soluciones explicadas están al final, y te recomiendo que, si fallas alguna, regreses a la unidad correspondiente y la repases. Equivocarse es parte del proceso; lo que no se vale es quedarse con la duda.

Cómo estudiar este libro para sacarle el máximo provecho: no lo leas como novela. Programa mientras lees. Ten tu computadora a la mano con Python instalado (en la Unidad 1 te explico cómo), y escribe cada ejemplo tú mismo, no lo copies y pegues. Los dedos también aprenden.

Cuando llegues a la sección de práctica, intenta resolver cada ejercicio antes de mirar la solución; si te atorras más de quince minutos, mira la solución, entiéndela y luego ciérrala y resuélvelo tú solo. Al terminar cada unidad, explícale en voz alta a alguien (o a ti mismo) lo que aprendiste: si puedes explicarlo, lo dominas.

Un consejo de colega: la programación se aprende programando, no leyendo. Este libro te da el mapa, pero el camino lo recorres tú con el teclado. Habrá momentos de frustración, errores que no entiendas y código que no jale a la primera. Es normal. Hasta los programadores con veinte años de experiencia se pelean con errores tontos todos los días. La diferencia entre quien aprende y quien abandona no es la inteligencia: es la constancia. Así que dale, con calma y sin prisa, pero sin pausa.

Nos vemos en la Unidad 1. ¡A programar!

Esteban lucasegaacademy.com

UNIDAD 1: Tus primeros pasos en Python

Introducción

Toda historia tiene un comienzo, y la tuya como programador empieza aquí. En esta unidad no vas a escribir programas complicados ni a memorizar definiciones raras: vas a entender qué es Python y por qué todo el mundo habla de él, vas a instalarlo en tu computadora como un profesional y vas a escribir tu primer programa. Parece poco, pero es muchísimo: la mayoría de la gente que "quiere aprender a programar" nunca pasa de la intención. Tú, al terminar esta unidad, ya tendrás el lenguaje instalado, tu primer código corriendo y la base mental para todo lo que viene.

Piénsalo así: aprender a programar es como aprender a cocinar. Antes de preparar un mole de tres días, necesitas conocer tu cocina, saber dónde están los cuchillos y prender la estufa sin quemar la casa. Esta unidad es eso: conocer tu cocina. Python es tu cocina, la instalación es prender la estufa, y tu primer programa es hervir agua. Suena simple, y lo es, pero sin eso no hay mole que valga.

Además, en esta unidad vas a formar los hábitos correctos desde el día uno: cómo escribir código limpio, cómo probar lo que haces y cómo leer los errores sin entrar en pánico. Esos hábitos te van a acompañar en las 30 clases y en toda tu carrera.

Explicación detallada

Qué es Python y por qué deberías aprenderlo

Python es un lenguaje de programación creado en 1991 por Guido van Rossum, un programador holandés que quería un lenguaje fácil de leer y escribir. El nombre no viene de la serpiente, como muchos creen, sino del grupo de comedia británico Monty Python, del cual Guido era fan. Ese detalle te dice mucho del espíritu del lenguaje: serio en potencia, relajado en forma.

Un lenguaje de programación es, en esencia, una forma de darle instrucciones a la computadora. La computadora solo entiende unos y ceros (código máquina), pero escribir unos y ceros es inhumano. Los lenguajes de programación son el puente: tú escribes instrucciones en un formato legible para humanos, y un programa llamado intérprete las

traduce a algo que la máquina entiende. Python es un lenguaje interpretado, lo que significa que puedes escribir una línea, ejecutarla y ver el resultado al instante, sin pasos intermedios de compilación. Eso lo hace perfecto para aprender y para experimentar.

Hay cientos de lenguajes de programación: Java, C++, JavaScript, Go, Rust y muchos más. Cada uno tiene sus fortalezas. Entonces, ¿por qué Python? Te doy razones de peso:

Primero, la legibilidad. Mira este ejemplo. En Python, para mostrar un mensaje en pantalla escribes:

```
print("Hola mundo")
```

En otros lenguajes, la misma tarea puede tomar cinco o seis líneas con palabras raras. La sintaxis de Python se parece al inglés simple (y con un poco de práctica, lo lees casi como español). Eso significa que pasas menos tiempo peleando con la sintaxis y más tiempo resolviendo problemas de verdad.

Segundo, la versatilidad. Python se usa en prácticamente todo: desarrollo web (con frameworks como Django y Flask se construyen sitios completos), ciencia de datos y análisis (con bibliotecas como pandas y NumPy), inteligencia artificial y machine learning (TensorFlow, PyTorch, scikit-learn), automatización de tareas aburridas (renombrar mil archivos, llenar reportes, mandar correos), videojuegos, robótica, ciberseguridad y más. Aprender Python es como aprender a usar un cuchillo de chef: te sirve para casi todo en la cocina.

Tercero, la comunidad. Python tiene una de las comunidades más grandes y amigables del mundo de la programación. Eso se traduce en algo muy práctico: cuando te atores (y te vas a atorar, todos nos atoramos), hay millones de personas que ya tuvieron tu mismo problema y lo documentaron. Además existen miles de bibliotecas gratuitas: código ya escrito por otros que puedes usar gratis. ¿Necesitas leer un archivo Excel? Hay una biblioteca. ¿Quieres hacer una gráfica? Hay una biblioteca. ¿Quieres entrenar una inteligencia artificial? Hay una biblioteca. Programar en Python muchas veces es como armar con piezas de Lego que otros ya fabricaron.

Cuarto, el mercado laboral. Python aparece consistentemente entre los lenguajes más demandados en las ofertas de trabajo, y suele estar entre los mejor pagados. Las empresas lo usan porque permite desarrollar rápido, y desarrollar rápido significa ahorrar dinero. Si tu meta es trabajar de programador, Python es una de las apuestas más seguras que puedes hacer.

Quinto, es gratis y de código abierto. No pagas licencias, no hay versiones "pro" escondidas. Lo descargas, lo instalas y es tuyo para siempre, para uso personal o comercial.

Analogía: Python como ayudante

Imagina que contratas a un ayudante súper eficiente pero muy literal: hace exactamente lo que le dices, ni más ni menos. Si le dices "tráeme las llaves", va y las trae. Si le dices "tráeme las llaves que están sobre la mesa de la cocina junto al frutero", también lo hace. Pero si le dices "tráeme eso", se queda parado porque "eso" no significa nada para él. Programar es aprender a dar instrucciones tan claras que hasta ese ayudante literal las entienda. Python es el idioma en el que le hablas a ese ayudante, y es un idioma amable: no te exige fórmulas crípticas, solo claridad.

Cómo funciona Python por dentro (sin complicarte)

Cuando escribes código Python y lo ejecutas, pasan tres cosas rapidísimo: primero, el intérprete lee tu código y verifica que la sintaxis esté bien (que no te falte un paréntesis, por ejemplo). Segundo, lo traduce a un formato intermedio llamado bytecode. Tercero, una máquina virtual lo ejecuta instrucción por instrucción. Todo esto ocurre en milisegundos y tú no tienes que hacer nada especial: solo escribes y ejecutas.

Hay dos formas principales de ejecutar Python: en modo interactivo y con scripts. El modo interactivo es como una conversación: abres la terminal, escribes `python`, aparece un prompt con `>>>`, escribes una línea, presionas Enter y ves el resultado al instante. Es ideal para probar ideas rápidas y para aprender. Los scripts son archivos de texto con extensión `.py` donde escribes tu programa completo y luego lo ejecutas de una vez. En este libro usarás ambos: el modo interactivo para experimentar y los scripts para programas de verdad.

Instalar Python como profesional

Instalar Python es gratis y sencillo, pero hay detalles que marcan la diferencia entre una instalación que te da dolores de cabeza y una que simplemente funciona. Vamos paso a paso según tu sistema.

En Windows: entra a python.org, ve a la sección de descargas y baja el instalador para Windows (siempre la versión estable más reciente de Python 3; olvídate de Python 2, que ya murió oficialmente en 2020). Al ejecutar el instalador, verás una casilla que dice "Add python.exe to PATH": MÁRCALA. Esto es crucial. PATH es la lista de lugares donde Windows busca programas cuando escribes un comando en la terminal. Si no marcas esa casilla, cuando escribas `python` en la terminal Windows te dirá que no conoce ese comando, y tendrás que reinstalar o configurar el PATH a mano. Márcala y dale a instalar. Después abre la terminal (busca "cmd" en el menú inicio) y escribe `python --version`. Si te muestra algo como `Python 3.12.1`, quedó perfecto.

En macOS: las Mac traen un Python viejo preinstalado que no debes usar (es Python 2 o una versión del sistema que no debes tocar). Descarga el instalador oficial desde python.org para macOS e instálalo. Después abre la Terminal y verifica con `python3 --version` (en Mac y Linux el comando suele ser `python3` en vez de `python`).

En Linux: casi todas las distribuciones ya traen Python 3 instalado. Abre la terminal y escribe `python3 --version`. Si no está, instálalo con el gestor de paquetes de tu distribución (por ejemplo, `sudo apt install python3` en Ubuntu/Debian).

Error común número uno: instalar Python dos veces o mezclar versiones. Si ya lo intentaste antes y algo falló, desinstala todo y empieza de cero con una sola versión oficial. Error común número dos: no marcar "Add to PATH" en Windows. Si ya te pasó, no llores: reinstala marcando la casilla, es lo más rápido.

Además de Python, te recomiendo instalar un buen editor de código. El Bloc de notas no sirve para programar (aunque técnicamente podrías). Las opciones populares y gratuitas son Visual Studio Code (el favorito de la mayoría, con extensiones para Python), PyCharm Community (potente, específico para Python) o incluso Thonny (muy simple, ideal para principiantes absolutos). Para este libro, cualquiera funciona. Elige uno, instálalo y familiarízate con abrir archivos `.py` en él.

Tu primer programa: el ritual del "Hola mundo"

En programación existe una tradición: el primer programa que escribes en un lenguaje nuevo debe mostrar el texto "Hola mundo" en pantalla. Es un ritual que empezó en los años setenta y todos los programadores del mundo lo han hecho. Hoy te toca a ti.

Abre tu editor, crea un archivo llamado `hola.py` y escribe exactamente esto:

```
print("Hola mundo")
```

Guárdalo, abre la terminal en la carpeta donde lo guardaste y ejecuta:

```
python hola.py
```

Verás aparecer `Hola mundo` en la pantalla. Felicidades: acabas de escribir y ejecutar tu primer programa.

Vamos a diseccionarlo porque, aunque es una sola línea, hay tres conceptos importantes escondidos. `print` es una función: un bloque de código reutilizable que hace algo específico. En este caso, muestra cosas en pantalla. Python trae muchas funciones integradas (las llamadas built-in) y `print` es la más famosa. Los paréntesis `()` después de `print` indican que estás llamando a la función, es decir, pidiéndole que haga su trabajo. Dentro de los paréntesis va el argumento: el dato que le pasas a la función para que trabaje con él. Aquí el argumento es `"Hola mundo"`, un texto encerrado entre comillas. En programación, a los textos se les llama strings (cadenas), y las comillas le dicen a Python "esto es texto, no una instrucción".

Prueba estas variantes en modo interactivo para afianzar:

```
print("Hola mundo")  
print('Hola mundo')  
print("¿Cómo estás?")
```

Las comillas dobles y simples funcionan igual para textos simples. Más adelante verás cuándo conviene usar unas u otras.

Otro experimento: ¿qué pasa si escribes `print(Hola mundo)` sin comillas? Pruébalo. Python te lanzará un error. Léelo con calma: te dirá algo como que `"Hola"` no está definido. Y tiene razón: sin comillas, Python cree que `Hola` es el nombre de una variable o algo definido, y como no existe, se queja. Este es tu primer encuentro con los mensajes de error, y te doy el consejo más valioso de todo el libro: los errores no son tus enemigos, son mensajes de tu ayudante diciéndote exactamente qué no entendió. Léelos siempre, no los ignores.

Buenas prácticas desde el día uno

Primera: guarda tus archivos con nombres claros y sin espacios. `hola.py` está bien; `mi primer programa.py` te dará problemas en la terminal. Usa guiones bajos si necesitas separar palabras: `mi_programa.py`.

Segunda: un archivo, un programa. No mezcles experimentos distintos en el mismo archivo; crea uno nuevo por cada ejercicio. Te lo agradecerás cuando quieras repasar.

Tercera: prueba en pequeño. Antes de escribir un programa grande, prueba cada pieza por separado en modo interactivo. Los profesionales hacen esto todos los días.

Cuarta: lee los errores completos. Cuando Python falla, muestra un traceback: un reporte que indica el archivo, la línea y el tipo de error. La última línea del traceback es la más importante: ahí dice qué pasó. Acostúmbrate a leerla.

Errores comunes de esta unidad

El error más común es el de sintaxis por paréntesis o comillas sin cerrar: `print("Hola mundo)` o `print("Hola mundo"`. Python te dirá `SyntaxError` y señalará dónde se confundió. La solución es contar: cada paréntesis que abres debe cerrarse, cada comilla que abres debe cerrarse.

Otro clásico: ejecutar el archivo desde la carpeta equivocada. Si la terminal dice que no encuentra `hola.py`, es que no estás en la carpeta donde lo guardaste. Usa el comando `cd` para moverte a la carpeta correcta, o en tu editor busca la opción de "ejecutar en terminal".

Y el tercero: en Windows, escribir `python` y que se abra la tienda de Microsoft en vez de Python. Eso pasa cuando Python no está en el PATH. Solución: reinstala marcando "Add to PATH".

Ejemplos paso a paso

Ejemplo 1: Tu primer script con varias líneas

Crea un archivo `presentacion.py`:

```
print("Hola, soy estudiante de Python")
print("Este es mi primer programa")
print("Voy a aprender a programar")
```

Ejecútalo con `python presentacion.py`. Verás las tres líneas en orden. Observa: Python ejecuta las instrucciones de arriba hacia abajo, una por una, en secuencia. Este concepto se llama flujo secuencial y es la base de todo: salvo que le digas lo contrario (con condicionales y ciclos, que verás en unidades siguientes), Python hace las cosas en el orden en que las escribes.

Ejemplo 2: print con números y operaciones

```
print(2 + 3)
print(10 - 4)
print("2 + 3")
```

La primera línea muestra 5, porque dentro de `print` puedes poner operaciones y Python las calcula antes de mostrar el resultado. La segunda muestra 6. La tercera muestra literalmente `2 + 3`, porque entre comillas todo es texto: Python no calcula nada dentro de un string. Esta diferencia entre código y texto es fundamental: las comillas crean una frontera. Afuera de las comillas, Python interpreta; adentro, solo muestra.

Ejemplo 3: Múltiples argumentos en print

```
print("Me llamo", "Ana", "y tengo", 25, "años")
```

`print` acepta varios argumentos separados por comas y los muestra separados por espacios: Me llamo Ana y tengo 25 años. Fíjate que mezclamos textos y un número sin comillas, y funcionó. Más adelante aprenderás formas más elegantes de armar mensajes (los f-strings de la Unidad 3), pero esto ya te sirve.

Ejemplo 4: Verificando tu instalación a fondo

Abre la terminal y ejecuta estos comandos uno por uno:

```
python --version
```

Debe mostrar tu versión de Python 3. Luego entra al modo interactivo:

```
python
```

Verás el prompt `>>>`. Ahí escribe:

```
>>> print("Python sí funciona")
>>> 2 + 2
>>> exit()
```

En modo interactivo, si escribes una operación sin `print`, Python muestra el resultado directamente. Para salir, escribe `exit()` o presiona Ctrl+Z y Enter en Windows, o Ctrl+D en Mac/Linux. Si todo esto funcionó, tu instalación está perfecta y estás listo para todo el curso.

Ejemplo 5: Comentarios, tu primer superpoder de documentación

```
# Este es un comentario: Python lo ignora por completo
print("Hola mundo") # Los comentarios pueden ir al final de una línea
```

Todo lo que sigue a un `#` en una línea es un comentario: Python no lo ejecuta. Los comentarios sirven para explicar qué hace tu código, para ti del futuro y para otros programadores. Úsalos desde ahora: cada programa que escribas en este libro debería tener al menos un comentario explicando qué hace. Los profesionales comentan su código; es una marca de calidad.

Práctica

Ejercicio 1

Escribe un programa que muestre tu nombre, tu edad y tu ciudad, cada dato en una línea distinta.

Solución paso a paso: necesitas tres llamadas a `print`, una por línea, porque cada `print` salta a la línea siguiente automáticamente. Crea el archivo `datos.py`:

```
# Muestra mis datos personales, uno por línea
print("Me llamo Carlos")
print("Tengo 28 años")
print("Vivo en Guadalajara")
```

Al ejecutarlo verás las tres líneas en orden. El concepto clave: `print` siempre termina con un salto de línea, así que cada llamada es una línea nueva.

Ejercicio 2

Muestra en pantalla el resultado de estas operaciones usando `print`: $15 + 27$, $100 - 58$, $7 * 6$.

Solución: mete cada operación dentro de un `print` y deja que Python calcule:

```
print(15 + 27)
print(100 - 58)
print(7 * 6)
```

Resultado: 42, 42 y 42. (Sí, las tres dan 42; no es magia, así las elegí para que verifiques.)

Concepto clave: dentro de los paréntesis de `print` puedes escribir expresiones, no solo datos fijos.

Ejercicio 3

Escribe un programa que muestre el siguiente texto exactamente, con los saltos de línea:

"Python es genial / Yo voy a dominarlo / lucasegaacademy.com".

Solución:

```
print("Python es genial")
print("Yo voy a dominarlo")
print("lucasegaacademy.com")
```

Si intentaste poner todo en un solo `print` separado por diagonales, también vale, pero la forma natural es un `print` por línea. Concepto clave: piensa en líneas, no en párrafos.

Ejercicio 4

Provoca un error a propósito: escribe `print("Hola mundo)` (sin cerrar la comilla) y ejecútalo.

Lee el mensaje de error completo y escríbelo en un comentario dentro del archivo corregido.

Solución: el error será un `SyntaxError` que menciona algo como "unterminated string literal" (literal de cadena sin terminar). El archivo corregido queda:

```
# Error provocado: SyntaxError: unterminated string literal
# (literal de cadena sin terminar). Faltaba la comilla de cierre.
print("Hola mundo")
```


Concepto clave: provocar errores a propósito y leerlos es la forma más rápida de dejar de tenerles miedo.

Ejercicio 5

En modo interactivo, calcula cuántos segundos tiene un día (24 horas × 60 minutos × 60 segundos) y cuántos tiene una semana, mostrando ambos resultados.

Solución: en el prompt `>>>`:

```
>>> 24 * 60 * 60
86400
>>> 24 * 60 * 60 * 7
604800
```

Un día tiene 86,400 segundos y una semana 604,800. Concepto clave: el modo interactivo es tu calculadora y tu laboratorio; úsalo para verificar ideas antes de escribirlas en un script.

Resumen de la unidad

En esta unidad aprendiste qué es Python y por qué es el lenguaje ideal para empezar: legible, versátil, con una comunidad enorme, demandado en el mercado laboral y completamente gratis. Entendiste que Python es un lenguaje interpretado: escribes instrucciones legibles y el intérprete las ejecuta. Instalaste Python en tu sistema (marcando "Add to PATH" en Windows) y verificaste la instalación con `python --version`. Escribiste tu primer programa con la función `print`, aprendiste qué son los argumentos y los strings, descubriste la diferencia entre código y texto (las comillas), probaste el modo interactivo, escribiste tus primeros comentarios con `#` y, lo más importante, aprendiste a leer los errores como mensajes de ayuda y no como fracasos. Con esto tienes tu cocina lista y la estufa prendida. En la Unidad 2 empezarás a guardar información con variables y a hacer cálculos de verdad.

UNIDAD 2: Variables, tipos de datos y operadores

Introducción

Si la Unidad 1 fue conocer tu cocina, esta unidad es llenar la alacena. Un programa que no guarda información no sirve para nada: sería como un chef con amnesia que olvida los ingredientes a cada paso. Las variables son la memoria de tu programa: cajitas con nombre donde guardas datos para usarlos después. Y los tipos de datos son las etiquetas que dicen qué clase de dato guardaste: texto, número entero, número decimal, verdadero o falso.

En esta unidad también vas a convertir a Python en tu calculadora personal con los operadores aritméticos. Parece básico, y lo es, pero aquí se construyen los cimientos de todo: sin variables no hay programas que recuerden nada, sin tipos no hay cálculos correctos, y sin operadores no hay matemáticas. Hasta la inteligencia artificial más avanzada, en el fondo, son variables y operaciones combinadas de formas ingeniosas.

Al terminar esta unidad sabrás guardar cualquier dato en una variable con un buen nombre, identificar y convertir tipos de datos, y hacer cálculos con todos los operadores de Python, incluyendo los que otros lenguajes no tienen.

Explicación detallada

Variables: la memoria de tu programa

Una variable es un nombre que le pones a un pedazo de memoria donde guardas un dato. La analogía clásica es la caja con etiqueta: imagina una caja de cartón donde guardas algo y le pegas una etiqueta con el nombre. Cuando necesitas lo que hay dentro, no tienes que describir la caja: solo dices su nombre. En código se ve así:

```
nombre = "Ana"
edad = 25
altura = 1.68
```

La línea `nombre = "Ana"` hace tres cosas: reserva un espacio en memoria, guarda ahí el texto "Ana" y le pega la etiqueta `nombre`. A partir de ese momento, cada vez que escribas `nombre` en tu programa, Python irá a buscar lo que guardaste.

Fíjate en el símbolo `=`. En matemáticas significa "es igual a", pero en programación significa "guárdale esto a esto". Se lee de derecha a izquierda: toma el valor de la derecha y guárdalo en el nombre de la izquierda. Esta diferencia es crucial y causa confusión al inicio: `edad = 25` no está afirmando que `edad` sea igual a 25 como ecuación, está dando la orden de guardar 25 en la variable `edad`. Más adelante, cuando veas `==` (doble igual), ese sí compara.

Lo hermoso de Python es que no tienes que declarar el tipo de la variable antes de usarla. En otros lenguajes debes escribir algo como "esta variable va a guardar un número entero" antes de usarla. En Python simplemente asignas y él detecta el tipo solo. Esto se llama tipado dinámico, y es una de las razones por las que Python se escribe tan rápido.

Las variables pueden cambiar de valor: por eso se llaman variables, porque varían.

```
edad = 25
print(edad)
edad = 26
print(edad)
```

Primero muestra 25, luego 26. La etiqueta sigue siendo la misma, pero el contenido de la caja cambió. También puedes crear variables a partir de otras:

```
precio = 100
iva = precio * 0.16
total = precio + iva
print(total)
```

Aquí `iva` se calcula usando `precio`, y `total` usa ambos. Python evalúa la expresión de la derecha primero y luego guarda el resultado. El programa muestra 116.0. Fíjate cómo el programa "recuerda" los valores: eso es exactamente para lo que sirven las variables.

Reglas para nombrar variables

Python tiene reglas estrictas y convenciones recomendadas. Las reglas estrictas (si las rompes, error): el nombre puede contener letras, números y guiones bajos, pero no puede empezar con número; no puede contener espacios ni símbolos raros; y no puede ser una palabra reservada del lenguaje (como `print`, `if`, `for`, `class`). Las convenciones (si las

rompes, no hay error, pero los programadores te mirarán feo): usa minúsculas y separa palabras con guiones bajos, lo que se llama `snake_case`. `nombre_usuario` está bien; `NombreUsuario` se usa para otra cosa (clases, Unidad 10); `nombreusuario` es legal pero difícil de leer.

Nombres buenos: `edad`, `precio_total`, `nombre_cliente`, `temperatura`. Nombres malos: `x` (¿x de qué?), `cosa`, `dato2`, `a1b2c3`. La regla de oro: el nombre debe decir qué guarda la variable. Tu código se lee muchas más veces de las que se escribe, así que invierte en buenos nombres. Dentro de seis meses, cuando releas tu programa, `total_con_iva` te dirá todo y `t` no te dirá nada.

Tipos de datos: las etiquetas de tus cajas

Cada dato en Python tiene un tipo, y el tipo determina qué puedes hacer con él. Los cinco tipos básicos que debes dominar ahora:

El string (`str`) es texto: cualquier cosa entre comillas. `"Ana"`, `"Hola mundo"`, `"123"` (ojo: entre comillas, `123` es texto, no número). Los strings sirven para nombres, mensajes, direcciones, todo lo que sea texto.

El entero (`int`) es un número sin decimales: `25`, `-10`, `0`, `1000000`. Sirve para contar cosas: edades, cantidades, años.

El flotante (`float`) es un número con decimales: `1.68`, `3.1416`, `-0.5`, `99.99`. En Python los decimales se escriben con punto, nunca con coma. Sirve para medidas, precios, promedios, todo lo que necesite precisión decimal.

El booleano (`bool`) solo tiene dos valores: `True` y `False` (con mayúscula inicial, así es Python). Representa verdadero o falso, sí o no, prendido o apagado. Es el tipo más simple y uno de los más poderosos: toda la lógica de decisiones de tus programas (Unidad 4) se basa en booleanos.

`None` es un valor especial que significa "nada", "vacío", "sin valor". Se usa para indicar que una variable aún no tiene un dato real. Es como una caja con etiqueta pero vacía a propósito.

Para saber el tipo de cualquier dato, usa la función `type()`:

```
print(type("Ana"))
print(type(25))
print(type(1.68))
print(type(True))
print(type(None))
```

Esto muestra `<class 'str'>`, `<class 'int'>`, `<class 'float'>`, `<class 'bool'>` y `<class 'NoneType'>`. (La palabra `class` aparecerá ahí; ignórala por ahora, la entenderás en la Unidad 10.)

¿Por qué importan los tipos? Porque cada tipo permite operaciones distintas. Puedes sumar dos enteros, pero ¿qué pasa si sumas un string con un entero? Pruébalo:

```
print("5" + 3)
```

Python lanza un `TypeError`: no puedes mezclar texto con números en una suma porque no tiene sentido. ¿Debería dar 8 o "53"? Python se niega a adivinar y te obliga a ser explícito. Esa rigidez es una virtud: evita errores silenciosos que en otros lenguajes te dan resultados equivocados sin avisarte.

En cambio, sumar dos strings sí funciona, pero hace algo distinto: los concatena (los pega).

```
print("Hola " + "mundo")
```

Muestra `Hola mundo`. El operador `+` hace cosas distintas según el tipo: con números suma, con textos pega. Esto se llama sobrecarga de operadores, y es elegante una vez que lo entiendes, pero peligroso si mezclas tipos sin querer.

Conversión de tipos: el arte de transformar datos

Como los tipos son estrictos, necesitas convertir datos de un tipo a otro. Python te da funciones para eso: `int()` convierte a entero, `float()` convierte a flotante, `str()` convierte a texto y `bool()` convierte a booleano.

```
texto = "25"
numero = int(texto)
print(numero + 5)
```

Aquí `int("25")` convierte el texto "25" en el número 25, y entonces `25 + 5` da 30. Sin la conversión, `"25" + 5` habría lanzado un error. Esta conversión es vital porque, como verás en la Unidad 3, todo lo que el usuario escribe en tu programa llega como texto, aunque escriba números.

```
print(float("3.5") + 1)
print(str(25) + " años")
print(int(3.99))
```

La primera línea muestra 4.5. La segunda muestra 25 años: convertimos el número a texto para poder pegarlo. La tercera muestra 3: `int()` trunca los decimales, no redondea. Guarda ese detalle, porque es una trampa clásica: `int(3.99)` es 3, no 4.

No todo se puede convertir: `int("hola")` lanza un `ValueError` porque "hola" no es un número. Python te avisa en vez de inventar un resultado. Cuando conviertas datos que vienen del usuario, siempre considera qué pasa si la conversión falla; en la Unidad 9 aprenderás a manejar esos errores con elegancia.

Operadores aritméticos: tu calculadora

Python tiene siete operadores aritméticos. Los cuatro básicos los conoces: `+` suma, `-` resta, `*` multiplica, `/` divide. Los tres especiales son joyas:

La división entera `//` divide y se queda solo con la parte entera: `7 // 2` es 3. El módulo `%` da el residuo de la división: `7 % 2` es 1. La potencia `**` eleva: `2 ** 3` es 8 (dos al cubo).

El módulo `%` merece atención especial porque es más útil de lo que parece. Sirve para saber si un número es par o impar (`numero % 2 == 0` significa par), para ciclos que se repiten (como los días de la semana), y para limitar valores a un rango. Lo verás en acción en las prácticas.

La precedencia de operadores (el orden en que se hacen las cuentas) sigue las reglas matemáticas que aprendiste en la escuela: primero potencias, luego multiplicaciones y divisiones, al final sumas y restas. Y como en la escuela, los paréntesis mandan:

```
print(2 + 3 * 4)
print((2 + 3) * 4)
print(2 ** 3 ** 2)
```

La primera da 14 (primero $3*4=12$, luego $+2$). La segunda da 20 (el paréntesis primero). La tercera da 512, porque las potencias se evalúan de derecha a izquierda: $3^{2=9}$, luego $2^9=512$. Cuando tengas duda, usa paréntesis: el código queda más claro y evitas sorpresas.

Un detalle importante: en Python 3, la división `/` siempre da flotante, aunque dividas enteros: $4 / 2$ es `2.0`, no `2`. Si quieres el resultado entero, usa `//`. Este fue un cambio respecto a Python 2 y es una mejora: la división se comporta como en matemáticas de verdad.

Operadores de asignación abreviada

Cuando quieres modificar una variable usando su propio valor, hay una forma corta:

```
contador = 0
contador = contador + 1
```

Esto se puede escribir como:

```
contador = 0
contador += 1
```

El `+=` significa "súmale esto a lo que ya tiene". Existen para todos los operadores: `-=`, `*=`, `/=`, `//=`, `%=`, `**=`. Son muy usados en ciclos (Unidades 5) para contadores y acumuladores. `contador += 1` es probablemente la línea más escrita en la historia de la programación.

Analogía: la alacena etiquetada

Vuelve a la analogía de la cocina. Las variables son frascos con etiqueta en tu alacena: el frasco `azúcar` guarda azúcar, el frasco `harina` guarda harina. Los tipos son el contenido: no puedes usar harina donde la receta pide azúcar solo porque ambos son polvos blancos; cada ingrediente tiene su naturaleza. Los operadores son tus utensilios: el cuchillo corta, la cuchara mezcla. Y la conversión de tipos es como moler: el grano de café entero no sirve para la cafetera de filtro, pero molido sí; transformaste el mismo ingrediente a la forma que necesitabas.

Errores comunes de esta unidad

El primero: usar una variable antes de crearla. Si escribes `print(edad)` sin haber asignado `edad` antes, Python lanza `NameError`. Las variables nacen cuando les asignas valor por primera vez; no existen antes.

El segundo: confundir = con ==. `edad = 25` guarda; `edad == 25` pregunta si edad vale 25. Si escribes `if edad = 25:` Python se quejará con un `SyntaxError`. Grábatelo: un igual asigna, dos iguales comparan.

El tercero: mezclar tipos sin convertir. `"Tienes"+ 25` falla; `"Tienes"+ str(25)` funciona. Cada vez que veas un `TypeError` mencionando `str` e `int`, casi seguro es esto.

El cuarto: nombres de variables con errores de dedo. Si creas `nomre` y luego usas `nombre`, Python dirá `NameError`. Para él son dos etiquetas distintas. Los buenos editores resaltan las variables para ayudarte a detectar esto.

El quinto: la división entera accidental. Si calculas un promedio con `//` en vez de `/`, perderás los decimales sin darte cuenta. Revisa qué división necesitas en cada caso.

Buenas prácticas

Nombra tus variables en español o inglés, pero sé consistente en todo el programa. No mezcles `nombre` con `user_age` en el mismo código. Inicializa tus contadores y acumuladores antes de usarlos (`total = 0`). Evita reutilizar la misma variable para cosas distintas: si `x` era la edad y luego la usas para el precio, tu código se vuelve un acertijo. Y cuando conviertas tipos de datos que vienen del usuario, hazlo lo antes posible, cerca de donde los recibes, para que el resto del programa trabaje con tipos correctos.

Ejemplos paso a paso

Ejemplo 1: Calculadora de propina

Un programa que calcula cuánto dejar de propina en un restaurante:

```
# Calculadora de propina
cuenta = 350.0
porcentaje = 15
propina = cuenta * porcentaje / 100
total = cuenta + propina
print("Propina:", propina)
print("Total a pagar:", total)
```

Línea por línea: guardamos la cuenta como flotante (350.0, porque el dinero lleva decimales), el porcentaje como entero, calculamos la propina con la fórmula clásica y sumamos para el

total. El programa muestra Propina: 52.5 y Total a pagar: 402.5. Observa cómo cada variable tiene un nombre que explica qué es, y cómo construimos el resultado paso a paso en vez de una sola fórmula gigante. Así se escribe código legible.

Ejemplo 2: Conversión de temperaturas

Convertir grados Celsius a Fahrenheit con la fórmula $F = C \times 9/5 + 32$:

```
celsius = 25
fahrenheit = celsius * 9 / 5 + 32
print(celsius, "grados Celsius son", fahrenheit, "Fahrenheit")
```

Muestra 25 grados Celsius son 77.0 Fahrenheit. Fíjate en la precedencia: primero `celsius * 9`, luego `/ 5`, luego `+ 32`. Da lo correcto porque la multiplicación y división van antes que la suma. Si hubieras escrito `celsius * (9 / 5 + 32)` daría un disparate: los paréntesis cambian todo.

Ejemplo 3: Par o impar con el módulo

```
numero = 17
residuo = numero % 2
print("Residuo:", residuo)
```

Muestra Residuo: 1. Todo número impar dividido entre 2 deja residuo 1; todo par deja residuo 0. Este truco del módulo es la base para detectar pares, para repartir turnos y para muchos algoritmos. Guárdalo bien.

Ejemplo 4: Tipos en acción y conversión

```
edad_texto = "30"
print(type(edad_texto))
edad = int(edad_texto)
print(type(edad))
print("En 10 años tendrás", edad + 10, "años")
```

Primero confirmamos que `edad_texto` es string, lo convertimos a entero, confirmamos que ahora es int, y hacemos la suma. Muestra `<class 'str'>`, `<class 'int'>` y En 10 años tendrás 40 años. Este patrón (recibir texto, convertir, calcular) lo usarás en casi todos tus programas interactivos.

Ejemplo 5: Acumulador con asignación abreviada

```
# Suma de tres ventas del día
total = 0
total += 150.50
total += 89.99
total += 200.00
print("Venta total del día:", total)
```

Muestra Venta total del día: 440.49. El acumulador `total` empieza en cero y va creciendo con `+=`. Este patrón aparece en todos lados: sumar ventas, contar puntos, acumular calificaciones. Nota el detalle de los flotantes: $150.50 + 89.99 + 200.00$ da 440.49, pero con decimales más complejos a veces verás cosas como 0.30000000000000004. Es un tema de cómo las computadoras guardan decimales (punto flotante binario); para dinero real los profesionales usan el módulo `decimal`, pero para este nivel, no te preocupes.

Ejemplo 6: Potencias y raíces

```
# Calcular el área de un círculo: pi * radio^2
radio = 5
pi = 3.1416
area = pi * radio ** 2
print("El área es:", area)
```

Muestra El área es: 78.54. Usamos `**` para elevar al cuadrado. ¿Y una raíz cuadrada? Es una potencia de 0.5:

```
numero = 16
raiz = numero ** 0.5
print("La raíz cuadrada es:", raiz)
```

Muestra 4.0. Truco elegante: en matemáticas, la raíz cuadrada es lo mismo que elevar a la $1/2$, y Python lo hace directo con `**`.

Práctica

Ejercicio 1

Crea variables para tu nombre (string), tu edad (int), tu altura en metros (float) y si eres estudiante (bool). Muestra cada una con `print` y también muestra su tipo con `type()`.

Solución paso a paso: declara las cuatro variables con tipos distintos y luego imprímelas junto con su tipo.

```
nombre = "Lucía"
edad = 22
altura = 1.65
es_estudiante = True
print(nombre, type(nombre))
print(edad, type(edad))
print(altura, type(altura))
print(es_estudiante, type(es_estudiante))
```

Verás cada valor seguido de su tipo. Concepto clave: una variable puede guardar cualquier tipo, y `type()` te dice cuál es en cada momento.

Ejercicio 2

Escribe un programa que convierta dólares a pesos. Guarda el tipo de cambio en una variable (por ejemplo 17.50 pesos por dólar) y una cantidad en dólares, y muestra cuántos pesos son.

Solución:

```
tipo_cambio = 17.50
dolares = 100
pesos = dolares * tipo_cambio
print(dolares, "dólares son", pesos, "pesos")
```

Muestra 100 dólares son 1750.0 pesos. Concepto clave: guarda los datos que pueden cambiar (como el tipo de cambio) en variables, no escritos directo en la fórmula; así solo cambias un lugar si el dato cambia.

Ejercicio 3

Pide mentalmente estos cálculos y verifícalos con Python: ¿cuánto es 2 elevado a la 10? ¿Cuál es el residuo de 100 entre 7? ¿Cuánto es 100 // 7?

Solución en modo interactivo:

```
>>> 2 ** 10
1024
>>> 100 % 7
2
>>> 100 // 7
14
```

$2^{10} = 1024$ (dato curioso: es la base del kilobyte). 100 entre 7 es 14 con residuo 2, por eso `//` da 14 y `%` da 2. Concepto clave: `//` y `%` son las dos caras de la división entera: cociente y residuo.

Ejercicio 4

Corrige este programa que tiene tres errores. Escríbelo bien y explica cada corrección:

```
nomre = "Pedro"
print(Nombre)
edad = "25"
print(edad + 5)
```

Solución: error 1, la variable se creó como `nomre` pero se usa como `Nombre`: Python distingue mayúsculas y es estricto con los nombres, así que es `NameError`. Error 2, relacionado: debe usarse el mismo nombre exacto. Error 3, `edad` es texto `"25"` y no se puede sumar con el número 5: hay que convertir con `int()`. Versión corregida:

```
nombre = "Pedro"
print(nombre)
edad = "25"
print(int(edad) + 5)
```

Muestra `Pedro` y `30`. Concepto clave: los tres errores más comunes del principiante (nombre mal escrito, mayúsculas, tipos mezclados) en un solo ejercicio.

Ejercicio 5

Un terreno rectangular mide 25 metros de largo por 18 de ancho. Calcula su área, su perímetro y cuántos metros de malla necesitas si quieres cercarlo dando dos vueltas completas.

Solución paso a paso: $\text{área} = \text{largo} \times \text{ancho}$; $\text{perímetro} = 2 \times (\text{largo} + \text{ancho})$; $\text{dos vueltas} = \text{perímetro} \times 2$.

```
largo = 25
ancho = 18
area = largo * ancho
perimetro = 2 * (largo + ancho)
malla = perimetro * 2
print("Área:", area, "m2")
print("Perímetro:", perimetro, "m")
print("Malla para dos vueltas:", malla, "m")
```

Resultados: área 450 m², perímetro 86 m, malla 172 m. Concepto clave: descomponer un problema en pasos con variables intermedias hace el código claro y fácil de verificar.

Ejercicio 6

Experimenta: ¿qué muestra `print("3" * 4)`? ¿Y `print("-" * 20)`? Explica por qué.

Solución: el primero muestra 3333 y el segundo muestra veinte guiones seguidos. El operador `*` con un string y un entero repite el texto esa cantidad de veces. No es un error: es una característica diseñada a propósito, útil para dibujar líneas separadoras o repetir patrones. Concepto clave: los operadores se adaptan al tipo de dato; con strings, `*` significa repetición.

Resumen de la unidad

En esta unidad llenaste tu alacena: aprendiste que las variables son nombres que guardan datos en memoria, que `=` asigna (no compara), y que Python detecta los tipos solo. Conociste los tipos básicos (`str`, `int`, `float`, `bool`, `None`), la función `type()`, y las conversiones con `int()`, `float()` y `str()`, incluyendo la trampa de que `int()` trunca en vez de redondear. Dominaste los siete operadores aritméticos (`+`, `-`, `*`, `/`, `//`, `%`, `**`) con su precedencia, descubriste los usos ingeniosos del módulo `math` para detectar pares, aprendiste la asignación abreviada (`+=` y familia) para contadores y acumuladores, y viste que los operadores cambian de comportamiento según el tipo (concatenar textos, repetir strings). También incorporaste buenas prácticas: nombres descriptivos en `snake_case`, no mezclar tipos sin convertir, e inicializar acumuladores. En la Unidad 3 vas a interactuar con el usuario: pedirle datos con `input`, convertirlos correctamente y armar mensajes dinámicos con f-strings.

UNIDAD 3: Entrada de datos y manejo de texto

Introducción

Hasta ahora tus programas son monólogos: tú escribes los datos y el programa los procesa. En esta unidad se vuelven diálogos: el programa le va a preguntar cosas al usuario, va a escuchar sus respuestas y va a responder con mensajes personalizados. Esa es la diferencia entre una calculadora con los números ya escritos y una calculadora de verdad donde tú tecleas.

Vas a dominar la función `input()` para pedir datos, entenderás por qué todo lo que entra por `input` llega como texto (y cómo convertirlo sin que tu programa se rompa), y aprenderás los f-strings, la forma moderna y elegante de mezclar variables dentro de textos. También conocerás los métodos de strings: herramientas listas para convertir mayúsculas, quitar espacios, dividir textos y reemplazar palabras.

Al terminar esta unidad podrás escribir programas interactivos de verdad: que pregunten el nombre, la edad, que hagan cálculos con lo que el usuario escriba y que respondan con mensajes bien presentados. Es el salto de "programas que hacen cosas" a "programas que hablan con personas".

Explicación detallada

`input()`: tu programa aprende a preguntar

La función `input()` detiene el programa, muestra un mensaje en pantalla y espera a que el usuario escriba algo y presione Enter. Lo que el usuario escriba se guarda donde tú digas:

```
nombre = input("¿Cómo te llamas? ")  
print("Hola", nombre)
```

Cuando ejecutas esto, ves ¿Cómo te llamas? y el programa se queda esperando. Escribes Ana, presionas Enter, y muestra Hola Ana. El texto dentro de los paréntesis de `input` es el prompt: el mensaje que invita al usuario a escribir. Fíjate que dejé un espacio al final del prompt ("¿Cómo te llamas? "): es un detalle de presentación para que lo que el usuario escriba no quede pegado al signo de interrogación. Detalles así separan los programas amateurs de los profesionales.

Puedes usar `input()` sin mensaje, pero no lo recomiendo: el usuario vería solo un cursor parpadeando sin saber qué se espera de él. Siempre pon un prompt claro que diga qué dato quieres: "¿Cuántos años tienes? ", "Escribe tu ciudad: ", "Dame un número: ".

La trampa de oro: input siempre devuelve texto

Aquí está el detalle más importante de toda la unidad, el que rompe más programas de principiantes: `input()` SIEMPRE devuelve un string, aunque el usuario escriba un número. Si el usuario escribe 25, tu programa no recibe el número 25: recibe el texto "25". Son cosas distintas, como vimos en la Unidad 2.

Mira lo que pasa si no conviertes:

```
edad = input("¿Cuántos años tienes? ")
print(edad + 10)
```

Si el usuario escribe 25, esto lanza `TypeError`, porque intenta sumar el texto "25" con el número 10. La solución es convertir inmediatamente después de recibir el dato:

```
edad = int(input("¿Cuántos años tienes? "))
print("En 10 años tendrás", edad + 10)
```

El patrón `int(input(...))` es uno de los más usados en Python: pide el dato y lo convierte a entero en la misma línea. Para decimales usa `float(input(...))`. Para texto que se queda como texto, solo `input(...)`.

¿Qué pasa si el usuario escribe letras cuando esperabas un número? `int("hola")` lanza `ValueError` y el programa se rompe. En programas reales eso se maneja con `try/except`, que aprenderás en la Unidad 9. Por ahora, asume que el usuario colabora, pero ten presente que los usuarios reales escriben cosas inesperadas: es tu trabajo futuro anticiparlo.

Buena práctica: convierte los datos justo donde los recibes, no tres líneas después. Así todo el resto del programa trabaja con tipos correctos y no tienes que andar recordando qué era texto y qué era número.

f-strings: textos dinámicos elegantes

Armar mensajes mezclando texto y variables es algo que harás todos los días. La forma antigua era concatenar con `+` y convertir todo a string a mano, que es tedioso y propenso a errores. La forma moderna son los f-strings: pones una `f` antes de las comillas y metes las variables entre llaves directamente en el texto:

```
nombre = "Ana"
edad = 25
print(f"Hola {nombre}, tienes {edad} años")
```

Muestra `Hola Ana, tienes 25 años`. Python sustituye cada `{variable}` por su valor automáticamente, sin que tengas que convertir nada. Dentro de las llaves puedes poner expresiones, no solo variables:

```
precio = 100
print(f"Total con IVA: {precio * 1.16}")
```

Muestra `Total con IVA: 116.0`. Y puedes formatear números: `f"{precio:.2f}"` muestra el número con dos decimales, ideal para dinero:

```
pi = 3.14159265
print(f"Pi con 2 decimales: {pi:.2f}")
print(f"Pi con 4 decimales: {pi:.4f}")
```

Muestra `3.14` y `3.1416`. El formato: `.2f` significa "punto flotante con 2 decimales". Para porcentajes existe: `.0%`:

```
descuento = 0.15
print(f"Descuento: {descuento:.0%}")
```

Muestra `Descuento: 15%`. Estos formatos hacen que tus programas se vean profesionales con mínimo esfuerzo.

Los f-strings llegaron en Python 3.6 y hoy son el estándar. Si ves código viejo con `.format()` o con `%`, son las formas anteriores; funcionan, pero los f-strings son más legibles. Úsalos siempre.

Métodos de strings: el texto se domina con herramientas

En Python, los strings son objetos llenos de métodos: funciones pegadas al dato que hacen cosas útiles. Se usan con la sintaxis `texto.metodo()`. Los esenciales:

`upper()` convierte a mayúsculas y `lower()` a minúsculas:

```
nombre = "aNa MaRÍA"
print(nombre.lower())
print(nombre.upper())
```

Muestra `ana maría` y `ANA MARÍA`. Útil para comparar respuestas del usuario sin importar cómo las escriba: convierte todo a minúsculas antes de comparar.

`strip()` quita espacios al inicio y al final:

```
respuesta = "  sí  "
print(respuesta.strip())
```

Muestra `sí` sin espacios. Los usuarios dejan espacios sin querer; `strip()` limpia la entrada. Existen `lstrip()` y `rstrip()` para quitar solo de un lado.

`replace()` reemplaza partes del texto:

```
frase = "Me gusta Java"
print(frase.replace("Java", "Python"))
```

Muestra `Me gusta Python`. Reemplaza todas las apariciones, no solo la primera.

`split()` divide un string en una lista de partes:

```
datos = "Ana,25,Guadalajara"
print(datos.split(","))
```

Muestra `['Ana', '25', 'Guadalajara']`. Es la forma estándar de procesar datos separados por comas, como archivos CSV simples. El argumento es el separador; sin argumento, divide por espacios.

`join()` es lo contrario: une una lista en un string:

```
palabras = ["Me", "gusta", "Python"]
print(" ".join(palabras))
```

Muestra Me gusta Python. El string antes del punto es el pegamento que une las partes.

`startswith()` y `endswith()` preguntan cómo empieza o termina un texto, devolviendo `True` o `False`:

```
archivo = "foto.jpg"
print(archivo.endswith(".jpg"))
print(archivo.startswith("foto"))
```

Ambos `True`. Útiles para validar nombres de archivos o respuestas.

`find()` busca una parte dentro del texto y devuelve su posición (o -1 si no está), y `count()` cuenta cuántas veces aparece algo:

```
texto = "Python es genial, Python es fácil"
print(texto.find("genial"))
print(texto.count("Python"))
```

`find` devuelve 10 (la posición donde empieza "genial"; las posiciones empiezan en cero) y `count` devuelve 2.

`in` no es un método sino un operador, pero es el más usado para buscar dentro de textos:

```
print("Python" in "Me gusta Python")
```

Muestra `True`. Se lee casi como español: "¿'Python' está en 'Me gusta Python'?".

Detalle importante: los métodos de strings NO cambian el string original, devuelven uno nuevo. Los strings son inmutables:

```
texto = "hola"
texto.upper()
print(texto)
```

Sigue mostrando `hola` en minúsculas, porque `upper()` devolvió un string nuevo que no guardamos en ningún lado. Para que funcione debes asignar: `texto = texto.upper()`. Este es un error clásico: llamar al método y olvidar guardar el resultado.

Caracteres especiales y strings multilínea

A veces necesitas caracteres que son difíciles de escribir directo, como saltos de línea dentro de un string o comillas dentro de comillas. Para eso existen las secuencias de escape, que empiezan con backslash `\`: `\n` es salto de línea, `\t` es tabulación, `\\` es un backslash literal, `\"` son comillas dobles dentro de un string con comillas dobles.

```
print("Línea uno\nLínea dos")
print("Ella dijo \"hola\"")
```

Para textos largos de varias líneas, usa triple comilla:

```
poema = """Rosa es roja,
el cielo es azul,
Python es genial,
y tú también."""
print(poema)
```

Respetar los saltos de línea tal como los escribes. Es ideal para mensajes largos, instrucciones o textos preformateados.

Analogía: el mostrador de atención al cliente

Piensa en tu programa como un empleado de mostrador. `input()` es cuando el empleado pregunta "¿en qué le puedo ayudar?" y escucha. El prompt es la pregunta amable y clara; sin ella, el cliente se queda confundido frente al mostrador. La conversión de tipos es cuando el empleado interpreta: si el cliente dice "veinticinco", el empleado lo anota como número 25 en el sistema, no como la palabra. Los f-strings son la respuesta personalizada: "Gracias, Ana, su total es \$116.00", no un "gracias, CLIENTE, su total es NUMERO". Y los métodos de strings son las herramientas del mostrador: el corrector que pone todo en mayúsculas para el archivo, el borrador que quita espacios de más en el formulario.

Errores comunes de esta unidad

El error rey: olvidar convertir lo que devuelve `input` y luego intentar operar con eso. Cada `TypeError` después de un `input` es sospechoso de esto. Solución: `int(input())` o `float(input())` según necesites.

El segundo: convertir sin verificar. Si el usuario escribe "veinte" en vez de "20", `int()` falla. En esta etapa acepta que pasará; en la Unidad 9 lo blindarás.

El tercero: olvidar guardar el resultado de un método de string. `nombre.upper()` solo no sirve; necesitas `nombre = nombre.upper()`.

El cuarto: llaves literales en f-strings. Si quieres mostrar llaves de verdad en un f-string, dúplicalas: `f"{{ho la}}"` muestra `{ho la}`. Es raro necesitarlo, pero cuando lo necesites, ya lo sabes.

El quinto: prompts sin espacio final o sin claridad. "Dame tu edad:" pega lo que el usuario escriba a los dos puntos; "Dame tu edad: "se ve profesional. Y "Dame un dato: "no dice qué dato; sé específico.

Buenas prácticas

Pide un dato por vez con prompts claros. Convierte inmediatamente al recibir. Usa f-strings para todos tus mensajes: son más legibles que concatenar. Limpia la entrada del usuario con `strip()` cuando compares respuestas. Muestra los números con formato adecuado (`:.2f` para dinero). Y valida lo obvio: si pides una edad, un número negativo es sospechoso (la validación completa viene con condicionales en la Unidad 4, pero ve pensando en ello).

Ejemplos paso a paso

Ejemplo 1: Presentación interactiva completa

```
# Programa que entrevista al usuario y lo presenta
print("=== ENTREVISTA RÁPIDA ===")
nombre = input("¿Cómo te llamas? ").strip()
edad = int(input("¿Cuántos años tienes? "))
ciudad = input("¿En qué ciudad vives? ").strip()
print(f"\nMucho gusto, {nombre}.")
print(f"Tienes {edad} años y vives en {ciudad}.")
print(f"En 5 años tendrás {edad + 5} años. ¡A seguir aprendiendo!")
```

Observa los detalles profesionales: título con separadores, `strip()` para limpiar espacios accidentales, conversión inmediata de la edad, y un f-string final que calcula dentro de las llaves. La `\n` al inicio del mensaje crea una línea en blanco para separar visualmente la entrevista del resultado.

Ejemplo 2: Ticket de compra formateado

```
# Genera un ticket de compra con formato de dinero
producto = input("Producto: ").strip()
precio = float(input("Precio: "))
cantidad = int(input("Cantidad: "))
subtotal = precio * cantidad
iva = subtotal * 0.16
total = subtotal + iva
print("\n--- TICKET ---")
print(f"Producto: {producto}")
print(f"Cantidad: {cantidad}")
print(f"Subtotal: ${subtotal:.2f}")
print(f"IVA (16%): ${iva:.2f}")
print(f"TOTAL: ${total:.2f}")
```

Con: `.2f` todos los montos se ven como dinero de verdad: \$402.50 en vez de \$402.5. Este programa ya parece software de tienda real. Fíjate cómo cada variable tiene un solo trabajo y los cálculos van en orden lógico.

Ejemplo 3: Normalizando respuestas del usuario

```
# Acepta "sí" en cualquier forma que lo escriban
respuesta = input("¿Te gusta Python? (sí/no): ").strip().lower()
if respuesta == "sí" or respuesta == "si":
    print("¡Excelente elección!")
else:
    print("Dale una oportunidad, te va a gustar.")
```

Encadenamos `strip().lower()`: primero quita espacios, luego pasa a minúsculas. Así "Sí", "SÍ" y "sí" quedan igual. (El `if` lo verás a fondo en la Unidad 4; aquí solo observa la limpieza de datos.) Aceptamos "si" sin acento porque muchos no lo escriben. Pensar en cómo escribe la gente real es parte del oficio.

Ejemplo 4: Procesando datos separados por comas

```
# El usuario escribe sus 3 materias favoritas separadas por comas
entrada = input("Escribe 3 materias separadas por comas: ")
materias = entrada.split(",")
print(f"Materia 1: {materias[0].strip().title()}")
print(f"Materia 2: {materias[1].strip().title()}")
print(f"Materia 3: {materias[2].strip().title()}")
```

`split(",")` rompe el texto en partes; `strip()` limpia espacios; `title()` pone mayúscula inicial a cada palabra ("matemáticas" -> "Matemáticas"). Los corchetes `[0]`, `[1]`, `[2]` acceden a cada parte (las listas se estudian en la Unidad 6, pero aquí ves su uso natural). Si el usuario escribe "matemáticas, historia, python", el programa lo presenta bonito.

Ejemplo 5: Contador de palabras y caracteres

```
# Analiza un texto que escriba el usuario
texto = input("Escribe una frase: ")
print(f"Caracteres: {len(texto)}")
print(f"Palabras: {len(texto.split())}")
print(f"En mayúsculas: {texto.upper()}")
print(f"¿Menciona 'Python'? {'python' in texto.lower()}")
```

`len()` cuenta caracteres del string; `len(texto.split())` cuenta palabras (`split()` sin argumento divide por espacios). La última línea combina `in` con `lower()` para buscar sin importar mayúsculas. Es un mini analizador de texto en cinco líneas.

Ejemplo 6: Escape y multilínea en un recibo

```
nombre = input("Nombre del cliente: ").strip()
print(f"\nRecibo para:\n\t{nombre}\nGracias por su compra.")
```

`\n` crea saltos de línea y `\t` una tabulación que indenta el nombre. El recibo se ve así:

```
Recibo para:
    Ana
Gracias por su compra.
```

Pequeños caracteres de formato, gran diferencia en presentación.

Práctica

Ejercicio 1

Escribe un programa que pida el nombre y el año de nacimiento del usuario, calcule su edad aproximada (año actual menos año de nacimiento) y muestre un mensaje con f-string.

Solución paso a paso: pide el nombre (texto), pide el año (convértelo a int de inmediato), calcula restando el año actual (2026), y presenta con f-string.

```
nombre = input("¿Cómo te llamas? ").strip()
nacimiento = int(input("¿En qué año naciste? "))
edad = 2026 - nacimiento
print(f"{nombre}, tienes aproximadamente {edad} años.")
```

Concepto clave: el patrón completo de programa interactivo (pedir -> convertir -> calcular -> mostrar) en cuatro líneas.

Ejercicio 2

Pide al usuario dos números (pueden tener decimales) y muestra su suma, resta, multiplicación y división, cada resultado con 2 decimales.

Solución: usa `float(input())` para aceptar decimales, calcula las cuatro operaciones y formatea.

```
a = float(input("Primer número: "))
b = float(input("Segundo número: "))
print(f"Suma: {a + b:.2f}")
print(f"Resta: {a - b:.2f}")
print(f"Multiplicación: {a * b:.2f}")
print(f"División: {a / b:.2f}")
```

Concepto clave: `float` en vez de `int` cuando los datos pueden tener decimales; el formato `:.2f` se aplica a expresiones, no solo a variables.

Ejercicio 3

Pide al usuario una frase y muestra: la frase en mayúsculas, en minúsculas, sin espacios al inicio/final, con la palabra "malo" reemplazada por "bueno", y cuántas veces aparece la letra "a".

Solución: aplica cada método y muestra el resultado.

```
frase = input("Escribe una frase: ")
print(f"Mayúsculas: {frase.upper()}")
print(f"Minúsculas: {frase.lower()}")
print(f"Sin espacios extra: '{frase.strip()}'")
print(f"Reemplazo: {frase.replace('malo', 'bueno')}")
print(f"Letra 'a': {frase.lower().count('a')} veces")
```

Nota que para contar la "a" convertimos a minúsculas primero, para contar también las mayúsculas. Concepto clave: los métodos se pueden encadenar y combinar para procesar texto en una sola línea.

Ejercicio 4

El siguiente programa falla. Encuentra el error, explícalo y corrígelo:

```
precio = input("Precio: ")
descuento = input("Descuento %: ")
final = precio - precio * descuento / 100
print(f"Precio final: {final}")
```

Solución: input devuelve texto, así que precio y descuento son strings y no se puede restar ni multiplicar con ellos: TypeError. Hay que convertir a float al recibir. Versión corregida:

```
precio = float(input("Precio: "))
descuento = float(input("Descuento %: "))
final = precio - precio * descuento / 100
print(f"Precio final: ${final:.2f}")
```

Concepto clave: este es EL error clásico del principiante; si lo reconoces al instante, ya vas adelante de la mayoría.

Ejercicio 5

Pide al usuario su nombre completo y muestra: solo el nombre de pila en formato título, sus iniciales en mayúsculas, y la longitud total del nombre sin contar espacios.

Solución paso a paso: divide por espacios, toma la primera parte, construye iniciales con la primera letra de cada parte, y cuenta caracteres quitando espacios con `replace`.

```
completo = input("Nombre completo: ").strip()
partes = completo.split()
pila = partes[0].title()
iniciales = "".join([p[0].upper() for p in partes])
sin_espacios = len(completo.replace(" ", ""))
print(f"Nombre de pila: {pila}")
print(f"Iniciales: {iniciales}")
print(f"Letras (sin espacios): {sin_espacios}")
```

Si escribes "ana maría lópez", obtienes pila "Ana", iniciales "AML" y 13 letras. (La línea de iniciales usa una comprensión de listas, tema de la Unidad 8; por ahora tómallo como receta.)

Concepto clave: combinar `split`, indexado y métodos resuelve problemas reales de procesamiento de nombres.

Ejercicio 6

Crea un programa "adivinator de edad en días": pide la edad en años y muestra cuántos días, horas y minutos aproximadamente ha vivido el usuario (usa 365 días por año).

Solución:

```
edad = int(input("¿Cuántos años tienes? "))
dias = edad * 365
horas = dias * 24
minutos = horas * 60
print(f"Has vivido aproximadamente:")
print(f"- {dias:,} días")
print(f"- {horas:,} horas")
print(f"- {minutos:,} minutos")
```

El formato `, .` agrega separadores de miles: 10,950 en vez de 10950. Mucho más legible para números grandes. Concepto clave: un cálculo simple con buena presentación se siente como un programa profesional.

Resumen de la unidad

En esta unidad tus programas aprendieron a conversar: dominas `input()` con prompts claros, entiendes que siempre devuelve texto y aplicas el patrón `int(input()) / float(input())` para convertir de inmediato. Manejas los f-strings para armar mensajes dinámicos, incluyendo expresiones dentro de las llaves y formatos como `:.2f` para dinero, `:.0%` para porcentajes y `:, .` para miles. Conoces los métodos esenciales de strings (`upper`, `lower`, `strip`, `replace`, `split`, `join`, `startswith`, `endswith`, `find`, `count`), el operador `in`, y el detalle crucial de que los métodos devuelven strings nuevos sin modificar el original. También viste secuencias de escape (`\n`, `\t`) y strings multilínea con triple comilla. Con esto ya haces programas interactivos que piden datos, calculan y presentan resultados bonitos. En la Unidad 4 les darás cerebro: tus programas empezarán a tomar decisiones con condicionales.

UNIDAD 4: Tu programa aprende a decidir

Introducción

Hasta este momento, tus programas han sido como un tren sobre rieles: siempre recorren exactamente el mismo camino, hacen exactamente lo mismo cada vez que los ejecutas y nunca se detienen a pensar. Les das un dato, hacen una operación, imprimen un resultado y se acabó. Eso está bien para empezar, pero te voy a decir algo: ningún programa útil del mundo real funciona así. Ni uno solo. Los programas que usas todos los días -el cajero automático, la aplicación de tu banco, el videojuego que te desvela, el sistema de calificaciones de la escuela- todos ellos tienen una habilidad que tus programas todavía no tienen: saben tomar decisiones.

Piensa en el cajero automático. No te da dinero a ciegas: primero revisa si tienes fondos. Si tienes suficiente, te entrega los billetes; si no, te muestra un mensaje de "fondos insuficientes" y te manda a tu casa con las manos vacías. Piensa en la aplicación de tu banco: si intentas transferir más de lo que tienes, no lo permite. Piensa en un videojuego: si tu personaje choca con una pared, no la atraviesa; si recoge una moneda, suma puntos; si le quedan cero vidas, aparece la pantalla de "game over". En todos estos casos el programa está haciendo lo mismo que tú haces cientos de veces al día: evaluar una situación y elegir qué hacer según el resultado.

Tú decides todo el tiempo sin darte cuenta. Si llueve, agarras paraguas; si no llueve, sales sin él. Si el semáforo está en rojo, te detienes; si está en verde, avanzas. Si tienes hambre y hay tacos al pastor cerca, pues ya sabes lo que pasa. Cada una de esas decisiones sigue un patrón muy simple: "SI pasa esto, ENTONCES hago esto otro; SI NO, hago otra cosa". Y aquí viene la buena noticia: ese patrón es exactamente lo que vas a aprender a programar en esta unidad. Los condicionales son la forma en que le enseñas a tu programa a comportarse como tú: a mirar una condición, decidir si se cumple o no, y actuar en consecuencia.

En esta unidad vamos a cubrir tres clases del curso que se conectan como piezas de un rompecabezas. En la Clase 9 vas a conocer la sentencia `if`, la más importante de todas: cómo se escribe, por qué lleva dos puntos al final y por qué la indentación (esos espacios al inicio de la línea) no es decoración sino ley en Python. En la Clase 10 vas a aprender `elif` y `else`, que te permiten cubrir todos los caminos posibles: ya no solo "si pasa esto", sino "si pasa esto, haz A; si pasa esto otro, haz B; y si no pasa nada de eso, haz C". Y en la Clase 11 vas a dominar los operadores de comparación - `==`, `!=`, `>`, `<`, `>=`, `<=` - que son las preguntas

que el `if` necesita para decidir, incluyendo la trampa más famosa de todos los lenguajes de programación: confundir el `=` (que asigna) con el `==` (que compara).

Cuando termines esta unidad vas a poder escribir programas que se comportan de forma inteligente: que saludan diferente según la hora del día, que calculan descuentos según la edad, que califican exámenes, que validan contraseñas, que reaccionan de una forma si un número es positivo y de otra si es negativo. En resumen: vas a pasar de programas que repiten a programas que piensan. Y te lo digo desde ahorita: este es el tema que más errores provoca en los principiantes -los dos puntos olvidados, la indentación mal puesta, el `=` en lugar del `==` - así que voy a dedicarle espacio generoso a cada error típico, con su explicación, su ejemplo de código roto y su arreglo. Porque aquí no se trata de memorizar sintaxis: se trata de entender la lógica de decidir, que es la misma lógica que usas todos los días desde que te levantas.

Una advertencia honesta antes de empezar: los condicionales parecen fáciles cuando los lees y se vuelven tramposos cuando los escribes tú. Vas a cometer el error de los dos puntos. Vas a mezclar tabs con espacios. Vas a comparar un texto con un número y Python te va a mirar feo. Todo eso es normal y es parte del aprendizaje. Lo importante es que cuando te pase, sepas exactamente qué buscar. Al final de esta unidad, cada uno de esos errores te va a parecer obvio, y eso es señal de que realmente aprendiste.

Explicación detallada

El `if`: tu programa hace su primera pregunta

Imagina que eres el guardia de un antro. Tu trabajo es sencillísimo: llega una persona, le pides su identificación, revisas su edad y tomas una decisión. Si tiene 18 años o más, la dejas pasar. Si tiene menos de 18, le dices "a tu casa, morro" y no entra. Tú no decides al azar: aplicas una regla a cada persona que llega, y dependiendo de si la regla se cumple o no, haces una cosa u otra.

El `if` de Python es exactamente ese guardia. Es una sentencia que le dice al programa: "evalúa esta condición; si es verdadera, ejecuta este bloque de código; si es falsa, sáltalo y sigue con tu vida". La palabra `if` significa "si" en inglés, y la lógica es idéntica a la del español: SI se cumple la condición, ENTONCES haz esto.

Mira el ejemplo más simple posible:

```
edad = 20

if edad >= 18:
    print("Puedes entrar al antro")
```

Analicemos cada parte con calma, porque cada detalle aquí importa. La primera línea crea una variable `edad` con el valor 20; eso ya lo dominas de unidades anteriores. La segunda línea es la novedad: empieza con la palabra reservada `if`, seguida de una condición (`edad >= 18`), y termina con dos puntos: `:`. La tercera línea es el bloque de código que se ejecuta solo si la condición es verdadera, y nota algo crucial: está indentada, es decir, tiene cuatro espacios al inicio.

Cuando Python ejecuta este programa, hace lo siguiente: llega a la línea del `if`, evalúa la condición `edad >= 18`. Como `edad` vale 20, la pregunta "¿20 es mayor o igual que 18?" tiene como respuesta sí, es verdadera. Entonces Python entra al bloque indentado y ejecuta el `print`. En pantalla aparece "Puedes entrar al antro". Si cambias `edad` a 15, la condición resulta falsa, Python se salta el bloque completo y el programa termina sin imprimir nada. Sin errores, sin quejas: simplemente no había nada que hacer.

Y aquí hay un concepto fundamental que tienes que grabarte a fuego: en Python, las condiciones se evalúan a valores booleanos, es decir, a `True` (verdadero) o `False` (falso). Todo `if` es, en el fondo, una pregunta que solo admite dos respuestas: sí o no. No hay "más o menos", no hay "tal vez". `edad >= 18` no es una frase decorativa: es una expresión que Python calcula y cuyo resultado es `True` o `False`. El `if` simplemente pregunta: "¿el resultado fue `True`?" Si sí, ejecuta el bloque. Si no, lo ignora.

Puedes comprobarlo tú mismo en la consola:

```
>>> edad = 20
>>> edad >= 18
True
>>> edad = 15
>>> edad >= 18
False
```

¿Ves? La comparación es una expresión como cualquier otra: recibe valores y devuelve un booleano. El `if` no hace magia; solo reacciona a ese `True` o `False`. Esta idea -que las condiciones son expresiones que devuelven booleanos- es la base de todo lo que viene en esta unidad, así que asegúrate de tenerla clara antes de seguir.

Otro ejemplo de la vida real: el semáforo. Imagina que estás programando el cerebro de un coche autónomo (o más modestamente, un juego de carreras). La lógica es: si el semáforo está en rojo, frena; en cualquier otro caso, sigue. En código:

```
semaforo = "rojo"

if semaforo == "rojo":
    print("Frena")
```

Si `semaforo` vale "rojo", la comparación `semaforo == "rojo"` devuelve `True` y se imprime "Frena". Si vale "verde", la comparación devuelve `False` y no pasa nada. Simple, ¿no? Pues agárrate, porque esta simpleza es engañosa: detrás de estas tres líneas hay decisiones de diseño del lenguaje que vamos a desmenuzar en las siguientes secciones, y cada una tiene su razón de ser.

Los dos puntos y la indentación: la ley sagrada de Python

Vamos a hablar del detalle que más dolores de cabeza causa a los principiantes y que, paradójicamente, es una de las mejores decisiones de diseño de Python: la indentación obligatoria.

En la mayoría de los lenguajes de programación, los bloques de código se marcan con llaves `{}` o con palabras como `begin` y `end`. Python decidió hacer algo distinto: usa la indentación -los espacios al inicio de la línea- para indicar qué instrucciones pertenecen al `if` y cuáles no. No es una sugerencia de estilo. No es para que el código "se vea bonito". Es sintaxis: si la indentación está mal, el programa no corre.

Mira este ejemplo y fíjate en los espacios:

```
edad = 20

if edad >= 18:
    print("Línea 1: estoy dentro del if")
    print("Línea 2: yo también estoy dentro del if")
print("Línea 3: yo estoy fuera del if")
```

Las líneas 1 y 2 tienen cuatro espacios al inicio: pertenecen al bloque del `if` y solo se ejecutan si la condición es verdadera. La línea 3 no tiene espacios: está fuera del `if` y se ejecuta siempre, haya entrado o no al bloque. Con `edad = 20`, la salida es:

```
Línea 1: estoy dentro del if
Línea 2: yo también estoy dentro del if
Línea 3: yo estoy fuera del if
```

Con `edad = 15`, la salida es:

```
Línea 3: yo estoy fuera del if
```

¿Captas la diferencia? La indentación es lo que le dice a Python dónde empieza y dónde termina el bloque. Es como los sangrados de un documento bien escrito: todo lo que está "adentro" del `if` va con sangría; cuando se acaba la sangría, se acabó el bloque.

Ahora, ¿por qué los dos puntos al final de la línea del `if`? Los dos puntos son la señal que le dice a Python (y a ti, que lees el código): "atención, aquí empieza un bloque indentado". Es como cuando en español escribes "los ingredientes son:" y después viene una lista. Los dos puntos anuncian que lo que sigue pertenece a lo anterior. Sin los dos puntos, Python no sabe que viene un bloque y te suelta un `SyntaxError` en la cara. Es el error más común de esta unidad y también el más fácil de arreglar: si Python te marca error en la línea del `if`, lo primero que revisas son los dos puntos. El noventa por ciento de las veces, ahí está el problema.

La regla de oro de la indentación: usa siempre 4 espacios por nivel. No 2, no 3, no 8, no un tabulador: 4 espacios. Es la convención oficial de Python (está escrita en el documento PEP 8, que es como la constitución del estilo en Python) y es lo que todo programador de Python espera ver. Si usas un número consistente de espacios, Python acepta técnicamente otros valores, pero no lo hagas: el día que trabajes con otras personas o que alguien lea tu código, los 4 espacios son el idioma común.

Y aquí viene la trampa mortal: mezclar tabuladores (tabs) con espacios. Se ven igual en muchos editores, pero para Python son caracteres distintos. Si indentas una línea con tab y la siguiente con espacios, Python se confunde y te lanza un `IndentationError` o, peor, un `TabError`. Configura tu editor para que la tecla Tab inserte 4 espacios automáticamente; todos los editores decentes (VS Code, PyCharm, hasta el IDLE que viene con Python) tienen esa opción. Créeme: cinco minutos configurando eso te ahorran horas de frustración después.

El error clásico se ve así:

```
if edad >= 18:
print("Puedes pasar")    # ¡ERROR! Falta la indentación
```

Python responde con `IndentationError: expected an indented block`. Traducción: "oye, me prometiste un bloque después de los dos puntos y no me diste nada indentado". El arreglo es obvio una vez que lo sabes: indenta la línea. Pero cuando estás empezando y ves ese mensaje en inglés, se siente como si la computadora te estuviera regañando en otro idioma. Ahora ya sabes qué significa y cómo arreglarlo en dos segundos.

Otro error típico, más sutil:

```
if edad >= 18:
    print("Mayor de edad")
    print("Esta línea tiene 5 espacios")    # ¡ERROR! Indentación inconsistente
```

La segunda línea tiene un espacio de más. Python exige que todas las líneas del mismo bloque tengan exactamente la misma indentación. Ni un espacio más, ni uno menos. Es estricto porque tiene que serlo: si fuera permisivo, no podría saber con certeza qué pertenece al bloque y qué no.

Piénsalo así: la indentación en Python es como las paredes de una casa. Los dos puntos son la puerta que anuncia "aquí empieza una habitación", y la indentación son las paredes que delimitan dónde termina. Sin paredes claras, no sabes si el sillón está en la sala o en la cocina. Python necesita saberlo con precisión milimétrica, y por eso es tan exigente.

Un consejo práctico que te va a salvar muchas veces: cuando escribas un `if`, escribe primero la línea con los dos puntos, presiona Enter y deja que el editor indente solo; escribe el bloque; y cuando termines el bloque, presiona Backspace o Shift+Tab para "salir" del bloque antes de seguir escribiendo código normal. Ese hábito -entrar y salir de los bloques de forma consciente- elimina la mayoría de los errores de indentación antes de que nazcan.

`if`, `elif`, `else`: todos los caminos

El `if` solo tiene un problema: únicamente sabe qué hacer cuando la condición se cumple. Cuando no se cumple, se queda callado y sigue adelante. Pero en la vida real casi siempre necesitas cubrir el otro camino también. Volvamos al guardia del antro: no basta con dejar pasar a los mayores de edad; también tienes que decirles algo a los menores, aunque sea "no puedes entrar". Para eso existe el `else`.

`else` significa "si no" o "de lo contrario". Se escribe después del bloque del `if`, al mismo nivel de indentación que el `if` (sin espacios extra, pegado al margen igual que el `if`), seguido de dos puntos, y debajo va su propio bloque indentado:


```
edad = 15

if edad >= 18:
    print("Puedes entrar al antro")
else:
    print("No puedes entrar, vuelve cuando cumplas 18")
```

Con `edad = 15`, la condición es falsa, así que Python se salta el bloque del `if` y ejecuta el bloque del `else`: imprime "No puedes entrar, vuelve cuando cumplas 18". Con `edad = 20`, ejecuta el bloque del `if` e ignora el `else`. Fíjate en algo importante: el `else` no lleva condición. No la necesita, porque su significado es "en cualquier otro caso". Si no entraste al `if`, entras al `else`. Punto. Son caminos mutuamente excluyentes: o uno o el otro, nunca los dos, nunca ninguno.

Pero la vida rara vez tiene solo dos caminos. Piensa en las calificaciones de la escuela: no es "aprobado o reprobado" nada más; hay matices. Un 10 es excelente, un 8 es bueno, un 6 es "de panzazo" y un 5 es reprobado. ¿Cómo programas eso con solo `if` y `else`? Podrías anidar `if` dentro del `else`, y funcionaría, pero el código se vuelve una pirámide inclinada hacia la derecha, difícil de leer:

```
# Así NO: pirámide de la perdición
if calificacion == 10:
    print("Excelente")
else:
    if calificacion >= 8:
        print("Bien")
    else:
        if calificacion >= 6:
            print("De panzazo")
        else:
            print("Reprobado")
```

Funciona, pero es horrible de leer. Para eso existe `elif`, que es la contracción de "else if" ("si no, si..."). Te permite encadenar tantas condiciones como quieras, de forma plana y legible:

```

calificacion = 8

if calificacion == 10:
    print("Excelente, ¡felicidades!")
elif calificacion >= 8:
    print("Muy bien")
elif calificacion >= 6:
    print("Pasaste de panzazo")
else:
    print("Reprobado, a estudiar más")

```

Con `calificacion = 8`, Python evalúa de arriba hacia abajo: ¿es igual a 10? No. ¿Es mayor o igual a 8? Sí. Ejecuta ese bloque, imprime "Muy bien", y -esto es clave- se salta todo lo demás, incluyendo el `else`. La cadena `if / elif / else` ejecuta como máximo UN bloque: el primero cuya condición sea verdadera. En cuanto encuentra uno, los demás ni los voltea a ver.

Esta es una diferencia fundamental con escribir varios `if` seguidos e independientes. Mira la diferencia:

```

# Varios if independientes: se evalúan TODOS
temperatura = 30

if temperatura > 25:
    print("Hace calor")

if temperatura > 20:
    print("Está agradable")

```

Aquí se imprimen las dos líneas, porque son dos `if` independientes y cada uno evalúa su propia condición. En cambio:

```

# Cadena if/elif: se ejecuta SOLO UN bloque
temperatura = 30

if temperatura > 25:
    print("Hace calor")
elif temperatura > 20:
    print("Está agradable")

```

Aquí solo se imprime "Hace calor", porque en cuanto la primera condición fue verdadera, la cadena terminó. ¿Cuándo usar cada uno? Usa `if / elif / else` cuando las opciones son

excluyentes (o una o la otra, como las calificaciones). Usa `if` independientes cuando varias cosas pueden ser verdaderas al mismo tiempo (por ejemplo, verificar por separado si un usuario es mayor de edad Y si tiene membresía premium: ambas pueden cumplirse).

El orden de las condiciones en una cadena `elif` importa muchísimo, y es una fuente clásica de errores lógicos (errores donde el programa corre sin quejarse pero hace lo incorrecto). Mira este ejemplo tramposo:

```
edad = 70

if edad >= 18:
    print("Adulto")
elif edad >= 60:
    print("Adulto mayor")
```

Con `edad = 70`, ¿qué se imprime? "Adulto". Porque la primera condición (`edad >= 18`) ya es verdadera para un 70, así que el `elif` nunca se evalúa. El bloque de "Adulto mayor" es código muerto: inalcanzable. La versión correcta pone primero la condición más específica (o más restrictiva):

```
edad = 70

if edad >= 60:
    print("Adulto mayor")
elif edad >= 18:
    print("Adulto")
else:
    print("Menor de edad")
```

Regla de oro: en una cadena `if / elif`, ordena las condiciones de la más específica a la más general, o de la más restrictiva a la menos restrictiva. Si lo haces al revés, las condiciones generales "se tragan" los casos especiales y esos bloques jamás se ejecutan. Python no te va a avisar; el programa corre perfecto y el resultado está mal. Estos son los errores más difíciles de cazar, así que desarrolla el hábito de revisar el orden de tus `elif` cada vez que escribas una cadena.

Un ejemplo completo de la vida real: descuentos por edad en el cine. Los niños menores de 12 pagan mitad de precio, los adultos mayores de 60 también tienen descuento, y el resto paga completo:

```

edad = 8
precio_base = 100

if edad < 12:
    precio = precio_base * 0.5
    print("Descuento infantil")
elif edad >= 60:
    precio = precio_base * 0.5
    print("Descuento adulto mayor")
else:
    precio = precio_base
    print("Precio completo")

print("Total a pagar:", precio)

```

Con `edad = 8` se imprime "Descuento infantil" y el total es 50.0. Nota que dentro de cada bloque puedes poner varias líneas, y todas deben tener la misma indentación. También nota algo elegante: la variable `precio` se define en cada camino, así que al final del `if / elif / else`, `precio` siempre tiene un valor sin importar qué camino se tomó. Ese es un patrón muy común: usar la cadena de condicionales para decidir un valor y luego usar ese valor después.

¿Y el `else` final es obligatorio? No. Puedes tener un `if` solo, un `if / else`, o un `if` con varios `elif` sin `else`. El `else` es el "por si acaso": captura todo lo que no entró en ninguna condición anterior. Mi recomendación: cuando las opciones son un conjunto cerrado y conocido (como las calificaciones), pon el `else` como red de seguridad. Si algún día llega un valor inesperado (una calificación de 15, por error), el `else` lo atrapa en lugar de dejar que el programa siga como si nada hubiera pasado.

Los operadores de comparación: las preguntas que el `if` necesita

Ya vimos que el `if` necesita una condición que se evalúe a `True` o `False`. Pero ¿de dónde salen esas condiciones? La mayoría de las veces, de comparar dos valores con los operadores de comparación. Son seis, y te los tienes que aprender como las tablas de multiplicar, porque los vas a usar en cada programa que escribas por el resto de tu vida como programador.

Aquí está la tabla completa, con su nombre, su símbolo y un ejemplo de cada uno:

Operador	Nombre	Significado	Ejemplo (<code>x = 10</code>)	Resultado
<code>==</code>	Igual a	¿son iguales?	<code>x == 10</code>	<code>True</code>
<code>!=</code>	Diferente de	¿son distintos?	<code>x != 10</code>	<code>False</code>
<code>></code>	Mayor que	¿el de la		

izquierda es mayor? | `x > 5` | `True` || `<` | Menor que | ¿el de la izquierda es menor? | `x < 5` | `False` || `>=` | Mayor o igual que | ¿es mayor o igual? | `x >= 10` | `True` || `<=` | Menor o igual que | ¿es menor o igual? | `x <= 9` | `False` |

Vamos uno por uno con ejemplos de la vida real.

`==` (igual a): pregunta si dos valores son iguales. Ojo: son DOS signos de igual, no uno. De la diferencia entre `=` y `==` hablamos en la siguiente sección porque merece su propio espacio. Por ahora:

```
password = "tacos123"

if password == "tacos123":
    print("Acceso permitido")
else:
    print("Contraseña incorrecta")
```

`!=` (diferente de): es lo contrario de `==`. Se lee "distinto de" y se escribe con signo de exclamación seguido de igual. Es útil cuando quieres reaccionar a que algo NO sea un valor específico:

```
semaforo = "verde"

if semaforo != "rojo":
    print("Puedes avanzar")
```

Si el semáforo es cualquier cosa menos rojo (verde, amarillo, o hasta un valor raro), avanzas. Solo te detienes con el rojo.

`>` y `<` (mayor que y menor que): comparan magnitudes. Funcionan con números, y también con textos (los comparan alfabéticamente, lo cual tiene sus trucos y lo veremos en su momento):

```
temperatura = 38

if temperatura > 37.5:
    print("Tienes fiebre, ve al doctor")
```

`>=` y `<=` (mayor o igual, menor o igual): incluyen el valor límite. La diferencia entre `>` y `>=` parece mínima, pero en la práctica decide casos frontera importantísimos. El guardia del antro usa `>= 18` porque alguien que justo hoy cumple 18 años SÍ puede entrar. Si usaras `> 18`, el

recién cumplido se quedaría fuera injustamente. Cada vez que escribas una comparación, pregúntate: "¿el valor exacto del límite debe entrar o no?" Esa pregunta te dice si necesitas el = extra.

Un detalle importante: los operadores de comparación no solo devuelven booleanos para el if; puedes guardarlos en variables y usarlos después:

```
edad = 20
es_mayor = edad >= 18    # es_mayor ahora vale True

if es_mayor:
    print("Adelante")
```

Esto es perfectamente válido y, de hecho, es una buena práctica cuando la condición es compleja o la vas a usar varias veces: le pones un nombre descriptivo y tu código se lee casi como español. `if es_mayor:` se entiende al instante; `if edad >= 18 and tiene_identificacion and not esta_en_lista_negra:` requiere más esfuerzo mental. Guardar condiciones con nombre es una forma barata de hacer tu código más legible.

También puedes combinar comparaciones con los operadores lógicos `and`, `or` y `not`, que verás a fondo en la siguiente unidad, pero te adelanto lo básico porque los vas a necesitar ya: `and` exige que ambas condiciones sean verdaderas, `or` se conforma con que al menos una lo sea, y `not` invierte el resultado:

```
edad = 20
tiene_ine = True

if edad >= 18 and tiene_ine:
    print("Puedes entrar al antro")
```

Aquí necesitas las dos cosas: ser mayor Y traer identificación. Si falta una, no entras. Es la lógica del guardia estricto.

Comparaciones encadenadas: el truco elegante de Python

En matemáticas escribes cosas como `"18 <= edad < 65"` para decir que la edad está entre 18 y 65. La mayoría de los lenguajes de programación no te dejan escribir eso directamente; tienes que partirlo en dos: `edad >= 18 and edad < 65`. Python, en cambio, sí te deja encadenar comparaciones exactamente como en matemáticas:

```
edad = 30

if 18 <= edad < 65:
    print("Edad laboral")
```

Esto se lee naturalito: "si 18 es menor o igual que edad Y edad es menor que 65". Python lo evalúa como `18 <= edad and edad < 65`, pero la versión encadenada es más corta y más legible. Úsala cada vez que verifiques que un valor está dentro de un rango: es uno de esos detallitos que hacen que el código Python se sienta bien escrito.

Otro ejemplo, con calificaciones:

```
calificacion = 8.5

if 6 <= calificacion <= 10:
    print("Calificación válida y aprobatoria")
```

El = no es el ==: la trampa más famosa de la programación

Vamos con el error que ha hecho llorar a más programadores principiantes que ningún otro en la historia. En Python (y en casi todos los lenguajes), el signo `=` y el doble `==` son dos cosas completamente distintas:

- `=` es el operador de ASIGNACIÓN: le da un valor a una variable. `edad = 20` significa "guarda el 20 en la variable edad".
- `==` es el operador de COMPARACIÓN: pregunta si dos valores son iguales. `edad == 20` significa "¿edad vale 20?" y responde `True` o `False`.

El error clásico es escribir `=` dentro de un `if` cuando querías comparar:

```
# ¡MAL! Esto es asignación, no comparación
if edad = 20:
    print("Tienes 20 años")
```

En Python, esto ni siquiera corre: te lanza `SyntaxError` porque la asignación no es una expresión válida como condición del `if`. En ese sentido, Python te protege: el error es ruidoso y obvio. Pero en otros lenguajes (como C o JavaScript en ciertos casos) este error sí corre y causa comportamientos extrañísimos. De todos modos, en Python hay una variante de este error que SÍ es silenciosa y peligrosa: cuando comparas con `==` algo que no es lo que crees, o cuando asignas antes del `if` sin darte cuenta:

```
# El error silencioso: querías comparar, pero asignaste antes
edad = 20
if edad == 20:      # bien
    print("Tienes 20")
```

Mi truco para no confundirlos nunca: lee el código en voz alta. `edad = 20` se lee "edad RECIBE 20" o "edad ES IGUAL A 20" en el sentido de asignación. `edad == 20` se lee "¿edad ES IGUAL A 20?" con tono de pregunta. Si tu línea del `if` no suena como pregunta, algo está mal. Otro truco: en el `if` casi siempre quieres `==` (pregunta); el `=` solito (afirmación) va en las líneas normales de código, nunca en la condición.

Y un caso que confunde incluso a intermedios: comparar textos con `==` distingue mayúsculas de minúsculas:

```
nombre = "Juan"

if nombre == "juan":    # False: "Juan" no es igual a "juan"
    print("Hola tocayo")
```

"Juan" y "juan" son textos distintos para Python, porque las mayúsculas y minúsculas son caracteres diferentes. Si quieres comparar sin importar mayúsculas, convierte ambos lados a minúsculas con `.lower()`:

```
if nombre.lower() == "juan":
    print("Hola tocayo")
```

Condicionales anidados: un `if` dentro de otro `if`

A veces una decisión depende de otra decisión previa. Por ejemplo, en el antro: primero verificas si es mayor de edad; solo si lo es, verificas si trae identificación. Eso es un `if` dentro de otro `if`, y se llama anidamiento:


```

edad = 20
tiene_ine = True

if edad >= 18:
    print("Eres mayor de edad")
    if tiene_ine:
        print("Puedes entrar al antro")
    else:
        print("Ve por tu INE y regresas")
else:
    print("No puedes entrar, eres menor")

```

Fíjate en la indentación: el `if` interior tiene 4 espacios (está dentro del bloque del `if` exterior), y su bloque tiene 8 espacios (dos niveles). Cada nivel de anidamiento agrega 4 espacios más. Python sabe exactamente qué bloque pertenece a qué `if` gracias a esos niveles.

Los anidados son poderosos, pero tienen un límite práctico: si anidas más de 2 o 3 niveles, el código se vuelve difícil de seguir (cada nivel lo empuja más a la derecha y tu cerebro tiene que recordar todas las condiciones anteriores). Cuando sientas que estás anidando demasiado, casi siempre puedes simplificar combinando condiciones con `and`:

```

# Anidado (funciona, pero se puede simplificar)
if edad >= 18:
    if tiene_ine:
        print("Puedes entrar")

# Equivalente con and (más plano y legible)
if edad >= 18 and tiene_ine:
    print("Puedes entrar")

```

No siempre son equivalentes (a veces necesitas mensajes distintos en cada nivel, como en el ejemplo del antro), pero cuando el resultado final es el mismo, prefiere la versión plana.

Valores truthy y falsy: el `if` es más flexible de lo que crees

Hasta ahora todas nuestras condiciones eran comparaciones que devolvían `True` o `False` explícitamente. Pero Python te deja poner casi cualquier cosa como condición del `if`, y él decide si la considera verdadera o falsa con unas reglas fijas. A los valores que se consideran verdaderos se les dice "truthy", y a los que se consideran falsos, "falsy".

La lista de valores falsy es corta y te la tienes que aprender: False, None, el número 0, el 0.0, los textos vacíos "", las listas vacías [], los diccionarios vacíos {} y los conjuntos vacíos. Todo lo demás es truthy. En la práctica, esto sirve muchísimo para verificar si algo "tiene contenido":

```
nombre = input("¿Cómo te llamas? ")

if nombre:
    print("Hola,", nombre)
else:
    print("No escribiste tu nombre")
```

Si el usuario solo presiona Enter, nombre es "" (texto vacío = falsy) y entra al else. Si escribió algo, entra al if. Sin esta flexibilidad tendrías que escribir `if nombre != ""`, que funciona igual pero es más verboso. La versión corta es idiomática en Python: los programadores experimentados la usan y la esperan.

Otro uso común: verificar que una lista tenga elementos antes de procesarla:

```
tareas = []

if tareas:
    print("Tienes", len(tareas), "tareas pendientes")
else:
    print("No tienes tareas, ¡a descansar!")
```

Pero ojo con un matiz importante: el número 0 es falsy. Si estás verificando una cantidad donde el cero es un valor válido y significativo, no uses la forma corta:

```
# ¡CUIDADO! Si saldo es 0, entra al else aunque 0 sea un saldo válido
saldo = 0
if saldo:
    print("Tu saldo es:", saldo)
else:
    print("No tienes saldo registrado") # ¡Mensaje incorrecto!
```

Aquí el 0 es un saldo perfectamente válido, pero como es falsy, el programa dice "no tienes saldo registrado", lo cual es mentira. En casos así, sé explícito: `if saldo is not None`: o maneja el cero como lo que es. La moraleja: la forma corta es elegante para textos y listas, pero con números piénsalo dos veces.

Errores comunes: el museo de los horrores del condicional

Vamos a hacer un recorrido por los errores que TODOS cometemos con los condicionales. Los pongo aquí juntos, con su código roto y su arreglo, para que cuando te topes con ellos (porque te vas a topar) los reconozcas al instante.

Error 1: olvidar los dos puntos. El más común de todos, sin competencia.

```
if edad >= 18
    print("Puedes pasar")
```

Python responde: `SyntaxError: expected ':'`. El arreglo: agrega los dos puntos al final de la línea del `if`. Punto. Desarrolla el reflejo: cada vez que escribas `if`, `elif` o `else`, los dos puntos van de cajón, como el punto final de una oración.

Error 2: indentación inconsistente o ausente.

```
if edad >= 18:
print("Puedes pasar")
```

Python responde: `IndentationError: expected an indented block`. El arreglo: indenta con 4 espacios. Y recuerda: todas las líneas del mismo bloque llevan exactamente la misma indentación.

Error 3: mezclar tabs y espacios. Se ve bien en tu editor, pero Python ve dos caracteres distintos y te lanza `TabError: inconsistent use of tabs and spaces in indentation`. El arreglo: configura tu editor para que el Tab inserte 4 espacios, y si heredaste código con tabs, usa la opción "convertir tabs a espacios" de tu editor.

****Error 4: usar `=` en lugar de `==` en la condición.****

```
if password = "tacos123":
```

Python responde con `SyntaxError`, así que al menos te avisa. El arreglo: doble igual para comparar. Lee la condición en voz alta: si no suena como pregunta, está mal.

Error 5: comparar textos con números. Este es silencioso en el sentido de que a veces corre pero nunca hace lo que quieres:

```
edad_texto = input("¿Cuántos años tienes? ") # input siempre devuelve texto
if edad_texto >= 18: # ¡TypeError! No se puede comparar texto con número
```

`input()` siempre devuelve un string, aunque el usuario escriba un número. Comparar `"20" >= 18` lanza `TypeError: '>=' not supported between instances of 'str' and 'int'`. El arreglo: convierte a número antes de comparar: `edad = int(input("¿Cuántos años tienes? "))`. Este error aparece en casi todos los programas interactivos de principiantes; ahora ya sabes que existe y cómo evitarlo.

****Error 6: el `else` huérfano o mal indentado.**** El `else` debe estar al mismo nivel que su `if`. Si lo indentas como si fuera parte del bloque, Python se confunde:

```
if edad >= 18:
    print("Mayor")
else:           # ¡ERROR! El else está dentro del bloque del if
    print("Menor")
```

El `else` es pareja del `if`: van alineados, los dos pegados al mismo margen. Piensa en ellos como una pareja de baile: siempre al mismo nivel.

****Error 7: condiciones en el orden equivocado en cadenas `elif`.**** Ya lo vimos con el ejemplo de "Adulto" vs "Adulto mayor": si la condición general va primero, se traga los casos especiales. Python no te avisa; el programa corre y da resultados incorrectos. El arreglo: de lo más específico a lo más general.

****Error 8: comparar flotantes con `==`.**** Los números decimales (`float`) tienen imprecisiones de representación interna. Esto:

```
resultado = 0.1 + 0.2
if resultado == 0.3:
    print("Son iguales")
```

NO imprime nada, porque `0.1 + 0.2` en la computadora es `0.30000000000000004`, no `0.3` exacto. Es un tema de cómo las computadoras guardan decimales en binario. El arreglo: en lugar de igualdad exacta, verifica que la diferencia sea muy pequeña: `if abs(resultado - 0.3) < 0.0001:`. No te preocupes si esto suena avanzado; solo recuerda la regla: nunca compares decimales con `==` esperando exactitud perfecta.

Buenas prácticas: escribe condicionales como profesional

Ya sabes cómo funcionan los condicionales; ahora veamos cómo escribirlos bien. La diferencia entre un principiante y un programador experimentado no está en conocer el `if`, sino en escribir condiciones que cualquier persona pueda leer y entender a la primera.

1. Condiciones legibles: que se lean como español. Si tu condición necesita un párrafo para explicarse, guárdala en una variable con un buen nombre:

```
# Difícil de leer
if edad >= 18 and tiene_ine and not esta_bloqueado and saldo > 0:
    print("Acceso permitido")

# Fácil de leer
puede_entrar = edad >= 18 and tiene_ine and not esta_bloqueado and saldo > 0
if puede_entrar:
    print("Acceso permitido")
```

2. Evita las negaciones dobles. `if not es_menor:` obliga a tu cerebro a hacer dos saltos mentales (no + menor = mayor). Escribe directamente lo que quieres decir: `if es_mayor:`. Y en general, prefiere condiciones positivas: `if tiene_permiso:` se entiende más rápido que `if not sin_permiso:`.

3. Early return (concepto): salte temprano de lo inválido. Cuando aprendas funciones, conocerás el `return`, pero el concepto aplica desde ya en tu forma de pensar: en lugar de anidar toda tu lógica dentro de un `if` gigante, primero descarta los casos inválidos y luego trabaja con lo bueno. Compara:

```
# Anidado: toda la lógica vive dentro del if
if edad >= 18:
    if tiene_ine:
        print("Bienvenido al antro")
        # ... aquí irían 50 líneas más, cada vez más a la derecha
```

```
# Temprano: descarta lo malo primero, lo bueno queda plano
if edad < 18:
    print("No puedes entrar")
else:
    if not tiene_ine:
        print("Ve por tu INE")
    else:
        print("Bienvenido al antro")
    # ... la lógica principal queda al nivel más externo posible
```

La idea es: maneja los casos especiales y los errores primero, para que el "camino feliz" (el flujo normal de tu programa) quede lo más plano y legible posible.

****4. No compares booleanos con `== True`.**** Si ya tienes un booleano, úsalo directo:

```
# Innecesario
if es_mayor == True:
    print("Pasa")

# Idiomático
if es_mayor:
    print("Pasa")

# Para negar, usa not
if not es_mayor:
    print("No pasas")
```

`es_mayor == True` es redundante: es como preguntar "¿es verdad que es verdad?". Los programadores de Python lo notan al instante y delata a un principiante.

5. Cubre todos los casos o documenta por qué no. Cuando escribas una cadena `if / elif` sin `else`, pregúntate: "¿qué pasa si ninguna condición se cumple?" Si la respuesta es "nada, y está bien que no pase nada", perfecto. Pero si hay valores posibles que no estás manejando, agrega el `else` como red de seguridad, aunque solo imprima un mensaje de advertencia. Los programas que fallan en silencio son los más difíciles de depurar.

6. Una condición, una idea. Si tu `if` tiene cuatro condiciones unidas con `and / or`, probablemente está haciendo demasiado. Divídelo en pasos con variables intermedias que tengan nombre, o en varios `if` secuenciales. Tu yo del futuro (y tus compañeros de trabajo) te lo van a agradecer.

Con esto cerramos la teoría. Tienes en tus manos todo el arsenal de las decisiones en Python: el `if` con sus dos puntos y su indentación sagrada, el `elif` y el `else` para cubrir todos los caminos, los seis operadores de comparación, la diferencia entre `=` y `==`, las comparaciones encadenadas, los anidados, los valores `truthy/falsy`, el catálogo de errores y las buenas prácticas. Ahora vamos a ver todo esto en acción, línea por línea.

Depurando condicionales: cuando el programa decide mal

Hay dos tipos de errores con los condicionales. Los primeros son ruidosos: Python te lanza un `SyntaxError` o un `IndentationError` y te dice en qué línea está el problema. Esos son molestos pero fáciles. Los segundos son silenciosos: el programa corre sin quejarse, pero toma el camino equivocado. Entra al `else` cuando debía entrar al `if`, o no entra a ningún bloque cuando debía entrar a uno. Estos son los peligrosos, porque la computadora cree que todo está bien.

La técnica más efectiva para cazarlos es vieja, simple y la usan hasta los programadores senior: imprime los valores antes del `if`. Si tu programa decide mal, lo primero que haces es verificar qué está viendo realmente:

```
edad = int(input("¿Cuántos años tienes? "))
print("DEBUG: edad =", edad, "tipo:", type(edad))

if edad >= 18:
    print("Mayor de edad")
else:
    print("Menor de edad")
```

Ese `print` de depuración te muestra el valor real y su tipo. La mitad de las veces, el misterio se resuelve aquí: descubres que `edad` es un texto en lugar de un número, o que trae un espacio extra, o que vale algo que no esperabas. Cuando ya encontraste el problema, borras la línea de `DEBUG` y listo.

La segunda técnica es imprimir qué condición se cumplió dentro de cada bloque, sobre todo en cadenas `elif` largas:

```
if calificacion == 10:
    print("DEBUG: entró al bloque del 10")
    print("¡Perfecto!")
elif calificacion >= 9:
    print("DEBUG: entró al bloque del 9")
    print("Excelente")
```

Así ves exactamente qué camino tomó el programa, sin adivinar. Es como ponerle cámaras de seguridad a tus decisiones.

Y la tercera técnica es probar con valores frontera: los valores justo en el límite de tus condiciones. Si tu condición es `edad >= 18`, prueba con 17, 18 y 19. El 17 debe ir por un camino, el 18 y el 19 por el otro. Si tu condición es `monto >= 500`, prueba con 499.99, 500 y 500.01. Los errores de `>` vs `>=` solo se revelan en la frontera: con valores holgados como 10 o

1000, ambas versiones se comportan igual y el error pasa desapercibido. Los programadores profesionales prueban las fronteras por instinto; tú puedes empezar a hacerlo desde hoy con cada `if` que escribas.

Un último consejo de depuración: cuando una condición compleja con `and` / `or` no se comporta como esperas, divídela e imprime cada parte por separado:

```
condicion_a = edad >= 18
condicion_b = tiene_ine
print("DEBUG: A =", condicion_a, "| B =", condicion_b)

if condicion_a and condicion_b:
    print("Puedes entrar")
```

Ver cada sub-condición por separado te dice inmediatamente cuál de las dos está fallando. Es mucho más rápido que mirar fijamente la línea completa tratando de razonarla en tu cabeza. Depurar no es adivinar: es recolectar evidencia.

Ejemplos paso a paso

Vamos a analizar seis programas completos, línea por línea. Te recomiendo que escribas cada uno en tu computadora y lo ejecutes con distintos valores para ver cómo cambian los caminos. Leer código sirve; correr código enseña.

Ejemplo 1: Mayoría de edad (el `if` / `else` básico)

```
edad = int(input("¿Cuántos años tienes? "))

if edad >= 18:
    print("Eres mayor de edad.")
    print("Puedes votar y sacar tu INE.")
else:
    print("Eres menor de edad.")
    print("Te faltan", 18 - edad, "años para la mayoría de edad.")

print("Fin del programa.")
```

Línea por línea: la primera línea pide la edad al usuario y la convierte a entero con `int()`, porque `input()` siempre devuelve texto y no podemos comparar texto con números (¿recuerdas el Error 5?). La línea del `if` evalúa `edad >= 18`: si el usuario tiene 18 o más, se

ejecutan las dos líneas indentadas del bloque `if`. Si tiene menos, se ejecutan las dos líneas del `else`; nota el detalle elegante de `18 - edad`, que calcula cuántos años le faltan, usando una operación aritmética dentro del `print`. La última línea está sin indentar, fuera de ambos bloques, así que se ejecuta siempre: es el cierre del programa.

Pruébalo con 20: imprime las dos líneas de mayor de edad más el "Fin del programa".
Pruébalo con 15: imprime "Eres menor de edad", "Te faltan 3 años para la mayoría de edad" y el fin. Dos caminos, un programa.

Ejemplo 2: Descuento por edad en el cine (cadena `if / elif / else`)

```
edad = int(input("¿Cuántos años tienes? "))
precio_base = 120

if edad < 5:
    descuento = 1.0
    mensaje = "¡Entrada gratis para peques!"
elif edad < 12:
    descuento = 0.5
    mensaje = "Descuento infantil del 50%"
elif edad >= 60:
    descuento = 0.5
    mensaje = "Descuento de adulto mayor del 50%"
else:
    descuento = 0.0
    mensaje = "Precio completo"

precio_final = precio_base * (1 - descuento)
print(mensaje)
print("Total a pagar: $", precio_final)
```

Análisis: primero pedimos la edad y definimos el precio base. La cadena evalúa de arriba hacia abajo. Con `edad = 4`: ¿es menor que 5? Sí: descuento total ($1.0 = 100\%$), mensaje de entrada gratis, y se salta todo lo demás. Con `edad = 8`: ¿menor que 5? No. ¿Menor que 12? Sí: 50% de descuento. Con `edad = 65`: las dos primeras fallan, ¿mayor o igual a 60? Sí: descuento de adulto mayor. Con `edad = 30`: ninguna se cumple, cae al `else`: precio completo.

Fíjate en el orden: las condiciones van de la más específica a la más general. Si pusiéramos `edad < 12` antes que `edad < 5`, un niño de 4 años caería en el descuento infantil del 50% en lugar de entrar gratis, porque la primera condición verdadera gana. El cálculo final

`precio_base * (1 - descuento)` es elegante: con descuento 1.0 da 0, con 0.5 da la mitad, con 0.0 da el precio completo. Una sola fórmula para todos los caminos.

Ejemplo 3: Calificaciones con `elif` (todos los caminos)

```
calificacion = float(input("¿Cuál fue tu calificación (0-10)? "))

if calificacion < 0 or calificacion > 10:
    print("Calificación inválida. Debe estar entre 0 y 10.")
elif calificacion == 10:
    print("¡Perfecto! Excelencia total.")
elif calificacion >= 9:
    print("Excelente trabajo.")
elif calificacion >= 8:
    print("Muy bien.")
elif calificacion >= 7:
    print("Bien, pero puedes mejorar.")
elif calificacion >= 6:
    print("Pasaste de panzazo. ¡A echarle ganas!")
else:
    print("Reprobado. A estudiar más para la próxima.")
```

Análisis: usamos `float()` porque las calificaciones pueden tener decimales (8.5). La primera condición es la red de seguridad: si la calificación está fuera del rango 0-10, avisamos que es inválida ANTES de clasificarla. Esto es el principio de "descarta lo malo primero" que vimos en buenas prácticas. Después, la cadena va de arriba hacia abajo: el 10 perfecto se revisa primero con `==` (porque `>= 9` también atraparía al 10 y le daría el mensaje genérico), luego 9, 8, 7, 6, y el `else` atrapa todo lo menor a 6.

Pruébalo con 10, 9.2, 7.5, 6.0, 5.9 y 15. Observa cómo el 15 cae en la primera condición (inválida) y nunca llega a la clasificación. Este patrón -validar primero, procesar después- lo vas a usar en casi todos los programas que reciban datos del usuario.

Ejemplo 4: Contraseña simple (comparación de textos)

```
password_correcta = "tacos_al_pastor_123"
intento = input("Escribe la contraseña: ")

if intento == password_correcta:
    print("Acceso permitido. Bienvenido.")
else:
    print("Contraseña incorrecta. Acceso denegado.")
```

Análisis: guardamos la contraseña correcta en una variable (en un programa real no harías esto así, pero para aprender está bien). Pedimos el intento al usuario. La comparación `intento == password_correcta` compara dos textos carácter por carácter, distinguiendo mayúsculas y minúsculas. Si el usuario escribe exactamente "tacos_al_pastor_123", entra; cualquier diferencia -una mayúscula, un espacio de más, un carácter distinto- lo manda al `else`.

Pruébalo escribiendo la contraseña con mayúscula al inicio ("Tacos_al_pastor_123") y verás que falla: así de estricta es la comparación de textos. Si quisieras hacerla insensible a mayúsculas, compararías `intento.lower() == password_correcta.lower()`. Este ejemplo también muestra algo importante: el `if` no solo compara números; compara cualquier tipo de dato.

Ejemplo 5: Número positivo, negativo o cero (tres caminos)

```
numero = float(input("Escribe un número: "))

if numero > 0:
    print("El número es positivo.")
elif numero < 0:
    print("El número es negativo.")
else:
    print("El número es cero.")
```

Análisis: el ejemplo más limpio de tres caminos mutuamente excluyentes. Todo número real cae en exactamente una de estas tres categorías: mayor que cero, menor que cero, o cero. Por eso el `else` final es seguro: si no es positivo ni negativo, por lógica matemática tiene que ser cero. No necesitamos escribir `elif numero == 0:` (aunque podríamos), porque el `else` ya lo cubre.

Este ejemplo es perfecto para entender por qué la cadena `if / elif / else` ejecuta un solo bloque: con `numero = 5`, la primera condición es verdadera, se imprime "positivo" y las demás

ni se evalúan. Con `numero = -3`, la primera falla, la segunda acierta. Con `numero = 0`, ambas fallan y cae al `else`. Tres entradas, tres salidas, cero ambigüedad.

Ejemplo 6: Comparador de dos números (comparaciones + f-strings)

```
a = float(input("Escribe el primer número: "))
b = float(input("Escribe el segundo número: "))

if a > b:
    print(f"{a} es mayor que {b}")
elif a < b:
    print(f"{a} es menor que {b}")
else:
    print(f"{a} es igual a {b}")

if a != b:
    diferencia = abs(a - b)
    print(f"Se diferencian por {diferencia}")
```

Análisis: la primera parte es la cadena clásica de tres caminos: mayor, menor o igual. Usamos f-strings (las verás a fondo en otra unidad, pero ya las conoces de ejemplos anteriores) para meter los valores dentro del mensaje. La segunda parte es un `if` independiente, separado de la cadena anterior: se evalúa por su cuenta. Si los números son distintos, calcula la diferencia absoluta con `abs()` (que convierte negativos a positivos) y la muestra.

Con `a = 10`, `b = 4`: imprime "10.0 es mayor que 4.0" y "Se diferencian por 6.0". Con `a = 7`, `b = 7`: imprime "7.0 es igual a 7.0" y el segundo `if` no hace nada porque `a != b` es falso. Este ejemplo muestra cómo combinar una cadena excluyente (la comparación principal) con un `if` independiente (el dato extra): dos estructuras de decisión conviviendo en el mismo programa, cada una con su propósito.

Práctica

Intenta resolver cada ejercicio por tu cuenta antes de mirar la solución. Si te atorras más de quince minutos, lee la solución, entiéndela, ciérrala y vuelve a escribirla tú solo. Esa es la forma en que el conocimiento se queda.

Ejercicio 1: Par o impar

Escribe un programa que pida un número entero al usuario e indique si es par o impar. Pista: un número es par si al dividirlo entre 2 el residuo es 0; en Python, el residuo se obtiene con el operador%.

Solución explicada paso a paso:

```
numero = int(input("Escribe un número entero: "))

if numero % 2 == 0:
    print("El número", numero, "es par.")
else:
    print("El número", numero, "es impar.")
```

Paso 1: pedimos el número y lo convertimos a entero con `int()`, porque necesitamos un número, no texto. Paso 2: la condición `numero % 2 == 0` calcula el residuo de dividir entre 2 y pregunta si es cero. Si el usuario escribió 8, `8 % 2` es 0, la condición es `True` y entra al `if`. Paso 3: si el residuo no es cero (por ejemplo, `7 % 2 = 1`), la condición es `False` y entra al `else`. Un detalle curioso: esto también funciona con negativos, porque `-4 % 2` es 0 en Python. Y con el cero: `0 % 2` es 0, así que el cero se reporta como par, lo cual es matemáticamente correcto.

Ejercicio 2: Descuento en la tiendita

Una tiendita ofrece descuentos según el monto de la compra: si la compra es de \$500 o más, 20% de descuento; si es de \$200 o más (pero menos de \$500), 10% de descuento; si es menor a \$200, sin descuento. Escribe el programa que pida el monto y muestre el descuento aplicado y el total a pagar.

Solución explicada paso a paso:

```
monto = float(input("Monto de la compra: $"))

if monto >= 500:
    descuento = 0.20
elif monto >= 200:
    descuento = 0.10
else:
    descuento = 0.0

total = monto * (1 - descuento)

print("Descuento aplicado:", int(descuento * 100), "%")
print("Total a pagar: $", round(total, 2))
```

Paso 1: leemos el monto como `float` porque puede tener centavos. Paso 2: la cadena de condiciones, ordenada de la más restrictiva a la menos restrictiva. Con \$600: la primera condición (`>= 500`) es verdadera, descuento del 20%, y se salta el resto. Con \$300: la primera falla, la segunda (`>= 200`) acierta, 10%. Con \$100: ambas fallan, `else`, sin descuento. Paso 3: calculamos el total con la fórmula `monto * (1 - descuento)`, que funciona para los tres casos. Paso 4: mostramos el descuento como porcentaje legible (`0.20 * 100 = 20`) y el total redondeado a 2 decimales con `round()`, porque el dinero siempre se muestra con centavos.

Nota el orden: si hubiéramos puesto `monto >= 200` primero, una compra de \$600 habría caído en el 10% en lugar del 20%. El orden de los `elif` no es decorativo: define el resultado.

Ejercicio 3: Semáforo inteligente

Escribe un programa que pida el color del semáforo ("rojo", "amarillo" o "verde", en minúsculas) y muestre la acción correspondiente: rojo = "Detente", amarillo = "Precaución, reduce la velocidad", verde = "Avanza". Si el usuario escribe cualquier otra cosa, muestra "Color no válido".

Solución explicada paso a paso:

```

color = input("Color del semáforo (rojo/amarillo/verde): ").lower()

if color == "rojo":
    print("Detente.")
elif color == "amarillo":
    print("Precaución, reduce la velocidad.")
elif color == "verde":
    print("Avanza.")
else:
    print("Color no válido. Escribe rojo, amarillo o verde.")

```

Paso 1: leemos el color y le aplicamos `.lower()` de inmediato, para que "ROJO", "Rojo" o "rOjO" también funcionen. Es un detalle de robustez: los usuarios escriben como quieren y tu programa debe aguantarlo. Paso 2: comparamos con `==` contra cada color válido. Como son textos exactos, el `==` es la herramienta correcta. Paso 3: el `else` final atrapa cualquier otra cosa ("azul", "morado", un typo como "rogo") y le dice al usuario qué se esperaba. Sin ese `else`, un color inválido haría que el programa terminara en silencio, y el usuario no sabría qué pasó.

Ejercicio 4: Aprobado con condiciones

En una materia se aprueba con 6 o más, pero además hay una regla especial: si la calificación es exactamente 10, se muestra un mensaje de felicitación; si es entre 6 y 9.99, "Aprobado"; si es menor a 6 pero mayor o igual a 0, "Reprobado"; y si está fuera del rango 0-10, "Calificación inválida". Escribe el programa.

Solución explicada paso a paso:

```

calificacion = float(input("Calificación (0-10): "))

if calificacion < 0 or calificacion > 10:
    print("Calificación inválida.")
elif calificacion == 10:
    print("¡Felicidades! Calificación perfecta. Aprobado.")
elif calificacion >= 6:
    print("Aprobado.")
else:
    print("Reprobado.")

```

Paso 1: validamos el rango primero, antes de clasificar. Si alguien escribe 15 o -3, lo atrapamos aquí y no contaminamos la lógica de aprobación. Paso 2: el caso especial del 10 va

antes que el `>= 6`, porque 10 también cumple `>= 6` y la primera condición verdadera gana. Si invirtieras el orden, el 10 perfecto recibiría el mensaje genérico de "Aprobado" y nunca verías la felicitación. Paso 3: `>= 6` atrapa todo lo aprobado restante (6 a 9.99). Paso 4: el `else` solo puede recibir valores entre 0 y 5.99 (todo lo demás ya fue atrapado arriba), así que es seguro decir "Reprobado". Este ejercicio combina las dos lecciones clave de la unidad: validar primero y ordenar de lo específico a lo general.

Ejercicio 5: El mayor de tres números

Pide tres números al usuario e indica cuál es el mayor. Si hay empate en el mayor (dos o tres iguales), indícalo también.

Solución explicada paso a paso:

```
a = float(input("Primer número: "))
b = float(input("Segundo número: "))
c = float(input("Tercer número: "))

if a >= b and a >= c:
    mayor = a
elif b >= a and b >= c:
    mayor = b
else:
    mayor = c

print("El mayor es:", mayor)

if (a == b == mayor) or (a == c == mayor) or (b == c == mayor):
    print("Hay un empate en el valor mayor.")
```

Paso 1: leemos los tres números. Paso 2: para saber si `a` es el mayor, no basta con que sea mayor que uno: tiene que ser mayor o igual que LOS DOS, por eso usamos `and`. Usamos `>=` en lugar de `>` para que los empates también entren por aquí (si `a` y `b` valen 9 y `c` vale 5, `a >= b and a >= c` es verdadero y `mayor` queda en 9, que es correcto). Paso 3: si `a` no fue el mayor, revisamos `b` con la misma lógica; si tampoco, el mayor tiene que ser `c` por eliminación, y el `else` lo asigna sin necesidad de más comparaciones. Paso 4: para detectar el empate, verificamos si al menos dos de los tres números son iguales al mayor. La expresión `a == b == mayor` es una comparación encadenada: significa `a == b and b == mayor`. Si cualquiera de las tres parejas empata en el valor mayor, lo reportamos.

Pruébalo con 5, 9, 3 (mayor: 9), con 7, 7, 2 (mayor: 7, hay empate) y con 4, 4, 4 (mayor: 4, hay empate). Observa cómo el `>=` maneja los empates sin código extra.

Ejercicio 6: Cajero automático simplificado

Escribe un programa que simule un retiro en un cajero: el usuario tiene un saldo inicial de \$5,000. El programa pide el monto a retirar y debe: rechazar montos menores o iguales a cero ("Monto inválido"), rechazar montos mayores al saldo ("Fondos insuficientes"), y en caso contrario restar el monto del saldo y mostrar el nuevo saldo. Además, si el retiro deja el saldo en cero exacto, muestra una advertencia: "Te quedaste sin fondos".

Solución explicada paso a paso:

```
saldo = 5000.0
retiro = float(input("Monto a retirar: $"))

if retiro <= 0:
    print("Monto inválido. Debe ser mayor a cero.")
elif retiro > saldo:
    print("Fondos insuficientes. Tu saldo es $", saldo)
else:
    saldo = saldo - retiro
    print("Retiro exitoso. Nuevo saldo: $", saldo)
    if saldo == 0:
        print("Advertencia: te quedaste sin fondos.")
```

Paso 1: definimos el saldo inicial y pedimos el monto. Paso 2: primera validación: montos inválidos (cero o negativos). Un cajero real no te deja "retirar" -\$100, y tu programa tampoco debería. Paso 3: segunda validación: si el retiro supera el saldo, lo rechazamos e informamos el saldo actual para que el usuario sepa cuánto sí puede retirar. El orden importa: primero lo inválido, luego lo imposible, al final lo válido. Paso 4: el `else` maneja el caso feliz: restamos y mostramos el nuevo saldo. Paso 5: el `if` anidado dentro del `else` revisa un caso especial posterior al retiro: si el saldo quedó en cero exacto, advertimos. Nota que este `if` interior solo se ejecuta si el retiro fue exitoso, porque vive dentro del bloque del `else`. Es un ejemplo perfecto de anidamiento con propósito: la advertencia solo tiene sentido después de un retiro válido.

Pruébalo con -50 (inválido), con 6000 (insuficiente), con 2000 (nuevo saldo: 3000) y con 5000 (nuevo saldo: 0 más la advertencia). Cuatro entradas, cuatro comportamientos distintos, un solo programa que decide.

Resumen de la unidad

Llegaste al final de la Unidad 4. Tus programas ya no son trenes sobre rieles: ahora saben mirar una condición y elegir su camino. Estos son los puntos clave que debes llevarte grabados:

- El `if` es la sentencia de decisión de Python: evalúa una condición y, si es `True`, ejecuta su bloque de código; si es `False`, lo salta.
- Toda condición es una expresión que devuelve un booleano (`True` o `False`). El `if` no hace magia: solo reacciona a ese valor.
- Los dos puntos al final de la línea del `if` anuncian que viene un bloque; olvidarlos es el error más común y el más fácil de arreglar.
- La indentación de 4 espacios no es decoración: es sintaxis. Define dónde empieza y termina cada bloque. Sé consistente y configura tu editor para que el Tab inserte espacios, nunca mezcles tabs con espacios.
- `else` significa "en cualquier otro caso" y no lleva condición; va al mismo nivel que su `if`.
- `elif` ("else if") encadena múltiples condiciones excluyentes; solo se ejecuta el bloque de la PRIMERA condición verdadera.
- En una cadena `if / elif`, ordena las condiciones de la más específica a la más general, o las generales se tragarán los casos especiales sin que Python te avise.
- Los seis operadores de comparación: `==` (igual), `!=` (distinto), `>` (mayor), `<` (menor), `>=` (mayor o igual), `<=` (menor o igual). Apréndetelos como las tablas.
- `=` asigna, `==` compara. En el `if` casi siempre quieres `==`. Lee la condición en voz alta: si no suena como pregunta, está mal.
- Python permite comparaciones encadenadas como `18 <= edad < 65`: úsalas para verificar rangos, se leen como matemáticas.
- Los `if` anidados (uno dentro de otro) sirven cuando una decisión depende de otra; no anides más de 2 o 3 niveles, o combina con `and`.
- Valores falsy: `False`, `None`, `0`, `""`, `[]`, `{}`, entre otros. Todo lo demás es truthy. Útil para verificar si algo tiene contenido, pero cuidado con el `0` cuando sea un valor válido.
- Convierte los datos de `input()` a número con `int()` o `float()` antes de compararlos: comparar texto con número lanza `TypeError`.

- Buenas prácticas: condiciones con nombres legibles, evita negaciones dobles, descarta los casos inválidos primero, no compares booleanos con `== True`, y usa el `else` como red de seguridad cuando el conjunto de casos es cerrado.

En la siguiente unidad vas a conocer los operadores lógicos `and`, `or` y `not` a fondo, y con ellos tus condiciones se van a volver mucho más expresivas: podrás combinar preguntas, invertirlas y construir lógica de decisión tan fina como la de un guardia de antro en sábado por la noche. Sigue programando mientras lees: los dedos también aprenden.

UNIDAD 5: Lógica y repetición: el poder de los ciclos

Introducción

Bienvenido a la unidad donde tu código empieza a pensar y a no rendirse. Hasta ahora, tus programas han sido como un guion de teatro que se lee de principio a fin: una línea tras otra, sin pausa, sin decisiones, sin segundas oportunidades. Eso está bien para empezar, pero los programas de verdad -los que usas todos los días en tu celular, en el banco, en los videojuegos- hacen dos cosas que tus programas todavía no saben hacer: **decidir** y **repetir**.

Piensa en un semáforo. Un semáforo no es una lista de instrucciones que se ejecuta una sola vez y se apaga. Es una máquina que decide ("¿está en verde? entonces deja pasar los carros; ¿está en rojo? entonces deténlos") y que repite ("una vez que pase por el rojo, vuelve al verde y empieza otra vez"). Día y noche, llueva o truene, el semáforo decide y repite, decide y repite. Tu misión en esta unidad es convertir tus programas en semáforos: que tomen decisiones combinadas con lógica y que repitan tareas sin cansarse.

La primera mitad de la unidad es la **lógica booleana**: el arte de combinar condiciones. Ya conoces los operadores de comparación de la unidad anterior ($>$, $<$, $==$, etc.), que te dan respuestas de sí o no. Ahora vas a aprender a combinar esas respuestas con las palabras **and** (y), **or** (o) y **not** (no). Esto es lo que permite que un banco decida: "te presto dinero SI tienes buen historial Y tienes ingresos suficientes". O que Netflix te muestre una película: "recomiéndala SI es de acción O es de comedia, pero NO si ya la vio el usuario". La lógica booleana está en todas partes: en los filtros de búsqueda, en los videojuegos ("¿tiene vida el jugador Y tiene llaves? entonces abre la puerta"), en los sistemas de seguridad. Es el lenguaje con el que las computadoras razonan.

La segunda mitad de la unidad son los **ciclos** (también llamados bucles o **loops**): **while** y **for**. Un ciclo es la herramienta más poderosa de la programación porque convierte a la computadora en tu empleada incansable. ¿Necesitas sumar un millón de números? Tú tardarías semanas; un ciclo lo hace en menos de un segundo. ¿Necesitas revisar 10,000 nombres en una lista buscando uno en particular? El ciclo no se aburre, no se distrae, no pide café.

Aquí va mi analogía favorita para esta unidad: imagina que eres el dueño de una tortillería. Todas las mañanas tienes que hacer 500 tortillas. Sin ciclos, tu "programa" sería una lista

infinita: "haz la tortilla 1, haz la tortilla 2, haz la tortilla 3..." hasta la 500. Un recetario de 500 pasos idénticos. Ridículo, ¿verdad? Con un ciclo, tu recetario dice: "**mientras** queden menos de 500 tortillas, haz otra". Dos líneas que hacen el trabajo de quinientas. Y con lógica booleana decides cosas como: "si la tortilla está dorada Y no está quemada, sácala del comal; si está cruda O quemada, déjala otro rato (o tírala)". Así de simple, así de poderoso.

Otra analogía: piensa en un DJ en una fiesta. El DJ tiene una **playlist** y la pone en **repeat**. Pero no es un repeat tonto: el DJ decide ("¿la gente está bailando? sigue con este género; ¿se sentaron todos? cambia de ritmo") y repite (cuando se acaba la lista, vuelve a empezar). Tus programas van a ser ese DJ: deciden con `and`, `or`, `not`, y repiten con `while` y `for`.

Al terminar esta unidad vas a poder escribir programas que validen contraseñas, que adivinen números jugando contigo, que impriman tablas de multiplicar completas, que cuenten, que sumen, que busquen, que filtren. Vas a entender por qué un ciclo infinito es el error más famoso (y más chistoso) de la programación, y cómo evitarlo. Y vas a dominar `range()`, la función que genera secuencias de números y que es la mejor amiga del ciclo `for`.

Un aviso honesto antes de empezar: esta unidad tiene más "trampas mentales" que las anteriores. La lógica booleana parece fácil ("¿qué tiene de difícil un **y** y un **o**?") hasta que te topas con la precedencia de operadores o con el cortocircuito. Y los ciclos parecen fáciles ("¿repetir? pues repito") hasta que tu programa se queda colgado en un ciclo infinito y no sabes por qué. No te preocupes: vamos a ver cada trampa con lupa, con ejemplos cotidianos y con los errores más comunes que cometen todos los programadores (sí, todos, incluidos los profesionales). Al final de la unidad, estas trampas te van a parecer obvias. Empecemos.

Explicación detallada

Los valores booleanos: el sí y el no de Python

Antes de combinar condiciones, recordemos la materia prima: los valores booleanos. En Python solo existen dos: `True` (verdadero) y `False` (falso). Escríbelos siempre con mayúscula inicial, porque `true` en minúsculas no existe para Python y te va a marcar error.

```
encendido = True
apagado = False
print(encendido) # True
print(apagado)   # False
```

Todo lo que evalúas con operadores de comparación produce un booleano:

```
edad = 20
print(edad >= 18)    # True
print(edad == 15)    # False
```

Y aquí viene la primera idea importante de la unidad: **los booleanos se pueden combinar como piezas de Lego**. Con `and`, `or` y `not` puedes armar condiciones complejas a partir de condiciones simples. Es como en la vida real: "voy a la fiesta" es una decisión simple, pero "voy a la fiesta SI termina mi tarea Y SI me dan permiso" es una decisión combinada. Python piensa igual que tú.

El operador `and`: las dos cosas o nada

El operador `and` significa "y". El resultado es `True` **solo si AMBAS condiciones son verdaderas**. Si al menos una es falsa, el resultado es falso. Piensa en entrar a un antro: necesitas tener INE **y** ser mayor de edad. Si te falta una de las dos, no entras.

Tabla de verdad de `and`:

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Memotécnica mexicana: `and` es como el taquero exigente. Le pides "tacos al pastor **y** con todo". Si se acabó el pastor, no hay tacos. Si hay pastor pero no hay cebolla ni cilantro, tampoco te los da "con todo". Solo cuando hay pastor **y** hay de todo, te sirve tus tacos. Exigente el don.

```
tiene_ine = True
es_mayor = True
puede_entrar = tiene_ine and es_mayor
print(puede_entrar)    # True

tiene_ine = True
es_mayor = False
puede_entrar = tiene_ine and es_mayor
print(puede_entrar)    # False (le falta una)
```

Ejemplo cotidiano: un banco que decide si te da un préstamo.

```
buen_historial = True
ingresos_suficientes = False

aprueba_prestamo = buen_historial and ingresos_suficientes
print(aprueba_prestamo) # False: le faltó una condición
```

Puedes encadenar más de dos condiciones con `and`. Todas tienen que ser verdaderas:

```
# Para graduarte necesitas: promedio >= 8 Y no deber materias Y pagar la
colegiatura
promedio = 9.0
sin_adeudos_materias = True
colegiatura_pagada = True

te_graduas = (promedio >= 8) and sin_adeudos_materias and colegiatura_pagada
print(te_graduas) # True: ¡felicidades, ya eres licenciado!
```

El operador `or`: con una basta

El operador `or` significa "o". El resultado es `True` si **AL MENOS UNA** de las condiciones es verdadera. Solo es `False` cuando todas son falsas. Es el amigo buena onda, el flexible.

Tabla de verdad de `or`:

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Memotécnica mexicana: `or` es como tu abuelita ofreciéndote de comer. "¿Quieres tamales **o** quieres pozole?" Con que aceptes uno, ella es feliz. Solo se pone triste si rechazas todo ("¿entonces no quieres nada, mi'jo?"). Con una condición verdadera basta para que `or` sea feliz.

```
es_fin_de_semana = False
es_dia_festivo = True

descansas = es_fin_de_semana or es_dia_festivo
print(descansas) # True: con el festivo basta
```

Ejemplo cotidiano: una tienda que da descuento.

```
# Descuento si eres estudiante 0 eres adulto mayor
es_estudiante = False
es_adulto_mayor = True

tiene_descuento = es_estudiante or es_adulto_mayor
print(tiene_descuento) # True
```

Otro ejemplo muy real: ¿llevo paraguas?

```
# Llevo paraguas si está nublado 0 si el pronóstico dice lluvia
esta_nublado = True
pronostico_lluvia = False

llevo_paraguas = esta_nublado or pronostico_lluvia
print(llevo_paraguas) # True
```

El operador not: el que voltea todo

El operador not significa "no". Invierte el valor: convierte True en False y False en True. Es el rebelde de la familia.

Tabla de verdad de not:

A	not A
True	False
False	True

```
esta_lloviendo = True
puedo_salir_sin_paraguas = not esta_lloviendo
print(puedo_salir_sin_paraguas) # False

tengo_hambre = False
estoy_lleno = not tengo_hambre
print(estoy_lleno) # True
```

not es muy útil para expresar "mientras NO pase algo", que es justo como piensan los ciclos while, como verás más adelante. También sirve para verificar que algo esté vacío o apagado:

```
bateria_baja = False
puedo_jugar = not bateria_baja
print(puedo_jugar) # True: a jugar se ha dicho
```


Un detalle de sintaxis: `not` va antes de la condición, solito, sin paréntesis obligatorios (aunque puedes usarlos para claridad):

```
activo = True
print(not activo)      # False
print(not(activo))     # False (igual, con paréntesis)
```

Combinando todo: expresiones booleanas complejas

La magia de verdad aparece cuando mezclas `and`, `or` y `not` en una sola expresión. Ejemplo de la vida real: ¿puedo entrar al concierto VIP?

```
tiene_boleto_vip = True
esta_en_lista = False
es_mayor = True
vetado = False

# Entra si (tiene boleto VIP O está en la lista) Y es mayor Y NO está vetado
puede_entrar = (tiene_boleto_vip or esta_en_lista) and es_mayor and not vetado
print(puede_entrar) # True
```

Vamos a desmenuzar esa expresión paso a paso, porque aquí es donde muchos se pierden:

1. `(tiene_boleto_vip or esta_en_lista) -> (True or False) -> True`
2. `True and es_mayor -> True and True -> True`
3. `True and not vetado -> True and not False -> True and True -> True`

Resultado: `True`. ¡A disfrutar el concierto!

Precedencia: `not` > `and` > `or` (el orden sí importa)

Cuando mezclas operadores sin paréntesis, Python sigue un orden de evaluación, igual que en matemáticas primero se multiplica y luego se suma. El orden es:

1. `** not **` se evalúa primero
2. `** and **` se evalúa después
3. `** or **` se evalúa al final

Memotécnica: "**NOA**" - **N**ot, **A**nd, **O**r... bueno, casi. Piensa en "Nado": **N**ot, **A**nd, **O**r. El que nada primero es `not`, luego `and`, al final `or`.

Mira por qué importa:

```
a = True
b = False
c = True

# Sin paréntesis: Python evalúa "b and c" primero, luego "a or (resultado)"
print(a or b and c)      # True  (porque es a or (b and c) = True or False = True)

# Con paréntesis cambias el significado:
print((a or b) and c)    # True  (porque es (True or False) and True = True and True)
```

Esos dos dieron lo mismo por casualidad. Mira un caso donde Sí cambia:

```
a = False
b = True
c = False

print(a or b and c)      # False (a or (b and c) = False or False = False)
print((a or b) and c)    # False ((False or True) and False = True and False = False)
```

Otro intento con valores que sí muestren la diferencia:

```
a = True
b = True
c = False

print(a or b and c)      # True  (a or (b and c) = True or False = True)
print((a or b) and c)    # False ((True or True) and False = True and False = False)
```

¡Ahí está! La misma expresión con y sin paréntesis da resultados opuestos. ****Moraleja:** cuando combines `and` y `or`, usa paréntesis aunque no sean estrictamente necesarios. ****** Los paréntesis son gratis, no cuestan nada, y hacen tu código legible para ti mismo dentro de tres meses. Los programadores profesionales los usan todo el tiempo por claridad.

```
# Sin paréntesis: tienes que recordar la precedencia
acceso = admin or tiene_permiso and not bloqueado

# Con paréntesis: se entiende a la primera
acceso = admin or (tiene_permiso and not bloqueado)
```

Cortocircuito (short-circuit): Python es flojo (y eso es bueno)

Aquí va un concepto elegante: **Python no evalúa lo que no necesita**. Si ya sabe el resultado, se detiene. Esto se llama **cortocircuito** (en inglés, **short-circuit evaluation**).

Con `and`: si la primera condición es `False`, el resultado ya es `False` seguro, así que Python **ni siquiera mira** la segunda condición.

```
def revisar_segunda():
    print("¡Me evaluaron!")
    return True

resultado = False and revisar_segunda()
print(resultado) # False... y "¡Me evaluaron!" NUNCA se imprimió
```

Con `or`: si la primera condición es `True`, el resultado ya es `True` seguro, así que Python **ni siquiera mira** la segunda.

```
resultado = True or revisar_segunda()
print(resultado) # True... y "¡Me evaluaron!" NUNCA se imprimió
```

¿Por qué te importa esto? Por dos razones:

Razón 1: puedes escribir condiciones más seguras. El truco clásico es verificar que algo exista ANTES de usarlo:

```
# Si la lista está vacía, "len(nombres) > 0" es False y Python
# ya no evalúa "nombres[0] == 'Ana'" (que daría error con lista vacía)
nombres = []
if len(nombres) > 0 and nombres[0] == "Ana":
    print("La primera es Ana")
else:
    print("Lista vacía o no empieza con Ana") # <- se imprime esto, sin error
```

Si Python evaluara las dos condiciones siempre, `nombres[0]` con una lista vacía lanzaría un error. El cortocircuito te protege. Por eso el orden importa: **pon primero la condición "barata y segura"** (la que verifica que algo existe) y después la "cara o riesgosa".

Razón 2: puedes ahorrar trabajo. Si la primera condición de un `or` ya es verdadera, Python se ahorra evaluar el resto. En programas grandes, esto ahorra tiempo de verdad.

Analogía mexicana: el cortocircuito es como cuando vas a pedir fiado a la tienda. Don Chuy te dice: "¿traes tu INE?" -"No." -"Entonces ni me digas lo demás, no hay fiado." No necesita escuchar el resto de tus argumentos porque con el primer "no" ya decidió. Eso es `and` en cortocircuito. Y con `or`: "¿vienes con tu mamá o traes el dinero?" -"Traigo el dinero." -"Ah bueno, pásale, no necesito saber más." Eso es `or` en cortocircuito.

El ciclo `while`: repite mientras sea verdad

Llegamos a los ciclos. El `while` significa "mientras". Su lógica es: **mientras la condición sea verdadera, repite el bloque de código.** Cuando la condición se vuelve falsa, el ciclo termina y el programa sigue con lo que esté después.

Sintaxis:

```
while condicion:
    # código que se repite
    # ¡algo aquí adentro debe acercar la condición a volverse falsa!
```

Ejemplo: contar del 1 al 5.

```
contador = 1
while contador <= 5:
    print(contador)
    contador = contador + 1    # <- SIN ESTA LÍNEA, EL CICLO NUNCA TERMINA

print("¡Terminó el ciclo!")
```

Salida:

```
1
2
3
4
5
¡Terminó el ciclo!
```

Vamos línea por línea:

1. `contador = 1` - Empezamos con el contador en 1. Esto se llama **inicializar**.
2. `while contador <= 5:` - Python pregunta: "`¿1 <= 5?`" Sí (True), entonces entra al ciclo.
3. `print(contador)` - Imprime 1.
4. `contador = contador + 1` - El contador ahora vale 2. Esto se llama **incrementar** (o actualizar).
5. Python regresa a la condición: "`¿2 <= 5?`" Sí. Imprime 2, contador pasa a 3...
6. ...así hasta que contador vale 6. "`¿6 <= 5?`" No (False). El ciclo termina.
7. Se ejecuta `print("¡Terminó el ciclo!")`.

Fíjate en algo crucial: la línea `contador = contador + 1` es la que **garantiza que el ciclo termine**. Cada vuelta acerca el contador al 6, que es donde la condición se vuelve falsa. Si olvidas esa línea, el contador se queda en 1 para siempre, `1 <= 5` siempre es verdadero, y el ciclo nunca termina. Eso nos lleva al tema más famoso de la unidad...

El peligro del ciclo infinito (y cómo escapar)

Un **ciclo infinito** es un `while` cuya condición nunca se vuelve falsa. El programa se queda dando vueltas para siempre, como un perro persiguiéndose la cola.

```
# ¡NO EJECUTES ESTO A MENOS QUE QUIERAS VERLO EN ACCIÓN!
contador = 1
while contador <= 5:
    print(contador)
    # Se nos olvidó incrementar el contador... ¡ciclo infinito!
```

¿Qué pasa si lo ejecutas? Tu programa imprime 1, 1, 1, 1... eternamente. No se traba tu computadora (el programa sigue funcionando, solo que repite lo mismo), pero no avanza ni termina. Es como poner el GPS en "dar vuelta en U" una y otra vez: técnicamente sigues manejando, pero nunca llegas.

¿Cómo se detiene un ciclo infinito? Con **Ctrl + C** en la terminal. Mantén presionada la tecla Ctrl y presiona C. Python interrumpe el programa y te devuelve el control. Pruébalo a propósito una vez (con un ciclo infinito de juguete) para que pierdas el miedo: es el "botón de pánico" del programador y lo vas a usar muchas veces en tu carrera. Todos los programadores han creado ciclos infinitos por accidente; es un rito de iniciación, como quemar las primeras tortillas.

Otro ciclo infinito clásico: la condición que empieza siendo falsa al revés...

```
# La condición YA es falsa desde el inicio: el ciclo nunca se ejecuta
contador = 10
while contador < 5:
    print(contador) # esto jamás se imprime
print("El ciclo ni siquiera empezó")
```

Esto no es un error, es un comportamiento válido: si la condición es falsa desde el principio, el while simplemente se salta. Pero ojo con el error contrario: escribir la condición al revés sin querer y crear un infinito sin darte cuenta.

****Regla de oro del while:**** siempre pregúntate antes de ejecutarlo: "¿qué va a hacer que esta condición se vuelva falsa?" Si no tienes una respuesta clara, tienes un ciclo infinito en potencia.

Contadores y acumuladores con while

Dos patrones (recetas) que vas a usar miles de veces:

El contador cuenta cuántas veces pasó algo:

```
# Contar cuántos números pares hay del 1 al 10
numero = 1
conteo_pares = 0

while numero <= 10:
    if numero % 2 == 0:
        conteo_pares = conteo_pares + 1
    numero = numero + 1

print("Pares encontrados:", conteo_pares) # 5
```

El acumulador va sumando (o juntando) valores:

```
# Sumar los números del 1 al 100 (el famoso ejercicio de Gauss de niño)
numero = 1
suma = 0

while numero <= 100:
    suma = suma + numero
    numero = numero + 1

print("La suma es:", suma) # 5050
```

La diferencia: el contador suma **1** cada vez que algo pasa (conteo = conteo + 1); el acumulador suma **el valor que corresponda** (suma = suma + numero). El acumulador también puede concatenar texto:

```
# Juntar letras para formar una palabra
letra_num = 1
palabra = ""

while letra_num <= 3:
    pedazo = input("Dame una letra: ")
    palabra = palabra + pedazo
    letra_num = letra_num + 1

print("Formaste:", palabra)
```

Si el usuario escribe "P", "a", "n", el acumulador palabra va creciendo: "" -> "P" -> "Pa" -> "Pan". ¡Pan! Ya tienes hambre, ¿verdad?

Validación de entrada con while: "no te dejo pasar hasta que..."

Uno de los usos más prácticos del while es **obligar al usuario a dar una respuesta válida**. Es el cadenero del antro: "mientras no me des una respuesta válida, te sigo preguntando".

```
edad = int(input("¿Cuántos años tienes? "))

while edad < 0 or edad > 120:
    print("Esa edad no tiene sentido. Intenta de nuevo.")
    edad = int(input("¿Cuántos años tienes? "))

print("Edad registrada:", edad)
```

Fíjate en la condición: `edad < 0 or edad > 120`. El ciclo se repite **mientras** la edad sea inválida (negativa o mayor a 120). En cuanto el usuario da una edad entre 0 y 120, la condición se vuelve falsa y el programa continúa. Lógica booleana + `while` trabajando en equipo: eso es programación de verdad.

Otro ejemplo clásico: pedir una contraseña hasta que sea correcta.

```
password_correcto = "tacos123"

intento = input("Escribe la contraseña: ")
while intento != password_correcto:
    print("Contraseña incorrecta. Intenta de nuevo.")
    intento = input("Escribe la contraseña: ")

print("¡Bienvenido! Contraseña correcta.")
```

El ciclo sigue preguntando **mientras** el intento sea diferente al correcto. En cuanto coincide, sales del ciclo. Este patrón lo vas a ver en cajeros automáticos, celulares bloqueados y videojuegos.

El ciclo `for` con `range()`: el ciclo más usado

El `for` es el ciclo consentido de Python. Mientras `while` repite "mientras algo sea verdad", `for` repite "para cada elemento de una secuencia". Y su mejor amiga es la función `range()`, que genera secuencias de números.

La forma más simple:

```
for i in range(5):
    print(i)
```

Salida:

```
0
1
2
3
4
```

¡Ojo! `range(5)` genera los números del **0 al 4**, no del 1 al 5. Python empieza a contar desde 0 (como los pisos de un edificio en muchos países: planta baja, primer piso...). Este es el famoso

off-by-one (error de uno): esperar 1-5 y obtener 0-4. No es un error de Python, es su diseño; solo tienes que acostumbrarte.

`range()` tiene tres formas:

****1.** `range(fin)` ** - del 0 hasta `fin - 1`:

```
for i in range(5):
    print(i) # 0, 1, 2, 3, 4
```

****2.** `range(inicio, fin)` ** - desde `inicio` hasta `fin - 1`:

```
for i in range(1, 6):
    print(i) # 1, 2, 3, 4, 5
```

****3.** `range(inicio, fin, paso)` ** - desde `inicio` hasta `fin - 1`, avanzando de `paso` en `paso`:

```
for i in range(0, 10, 2):
    print(i) # 0, 2, 4, 6, 8 (números pares)

for i in range(1, 10, 2):
    print(i) # 1, 3, 5, 7, 9 (números impares)
```

El `paso` también puede ser **negativo** para contar hacia atrás:

```
for i in range(10, 0, -1):
    print(i) # 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
print("¡Despegue!")
```

Y con `paso` negativo más grande:

```
for i in range(100, 0, -10):
    print(i) # 100, 90, 80, 70, 60, 50, 40, 30, 20, 10
```

Analogía: `range()` es como decirle a alguien "cuenta del 1 al 10". `range(10)` es "cuenta hasta el 10 empezando de cero" (raro en español, normal en Python). `range(1, 11)` es "del 1 al 10" dicho a la mexicana. `range(0, 11, 2)` es "del 0 al 10 de dos en dos". Y `range(10, 0, -1)` es la cuenta regresiva de Año Nuevo.

Detalles importantes de `range()`:

- El valor **fin nunca se incluye**. `range(1, 5)` da 1, 2, 3, 4. El 5 no aparece. Piensa: "range llega HASTA pero no INCLUYE el fin".
- `range()` no crea una lista gigante en memoria; genera los números uno por uno según se necesitan. Por eso `range(1000000)` no truená tu computadora.
- Si quieres ver los números como lista (para depurar), usa `list(range(5))` -> `[0, 1, 2, 3, 4]`.

Comparando `while` y `for` para la misma tarea:

```
# Con while (tú controlas el contador manualmente)
contador = 1
while contador <= 5:
    print(contador)
    contador = contador + 1

# Con for (Python controla el contador por ti)
for i in range(1, 6):
    print(i)
```

Ambos imprimen 1, 2, 3, 4, 5. Pero el `for` es más corto, más legible y **no puede volverse infinito por olvidar el incremento**, porque `range()` ya sabe cuándo terminar. Por eso el `for` es el ciclo más usado: cuando sabes cuántas veces vas a repetir, `for` es tu amigo.

for recorriendo strings: letra por letra

El `for` no solo trabaja con `range()`; puede recorrer **cualquier secuencia**, incluyendo strings. Cada vuelta, la variable toma un carácter del texto:

```
palabra = "Python"
for letra in palabra:
    print(letra)
```

Salida:

```
P
y
t
h
o
n
```

Esto es increíblemente útil. Ejemplo: contar cuántas vocales tiene una palabra.

```
texto = "murciélago"
vocales = "aeiouáéíóú"
conteo = 0

for letra in texto:
    if letra in vocales:
        conteo = conteo + 1

print("Vocales:", conteo) # 5
```

(Ojo: `in` aquí pregunta "¿esta letra está dentro del string vocales?". Funciona con strings y es muy práctico.)

Otro ejemplo: censurar una palabra.

```
mensaje = "eres un tonto"
censurado = ""

for letra in mensaje:
    if letra == "o":
        censurado = censurado + "*"
    else:
        censurado = censurado + letra

print(censurado) # eres un t*nt*
```

(Bueno, quedó medio censurado, pero se entiende la idea.)

break: la salida de emergencia

A veces necesitas salir de un ciclo **antes** de que termine naturalmente. Para eso existe `break`: rompe el ciclo en seco y salta a lo que sigue después.

```
for i in range(1, 100):
    print(i)
    if i == 5:
        break
print("Salí del ciclo")
```

Salida:

```
1
2
3
4
5
Salí del ciclo
```

Aunque el `range` llegaba hasta 99, el `break` cortó todo en el 5. Es como bajarte del camión antes de la terminal porque ya llegaste a tu parada: el camión (el ciclo) seguiría, pero tú ya te bajas.

El uso más clásico de `break`: **buscar algo y detenerse al encontrarlo**.

```
# Buscar un nombre en una lista
nombres = ["Ana", "Luis", "María", "Pedro", "Sofía"]
buscado = "María"

for nombre in nombres:
    if nombre == buscado:
        print("¡Encontrado!", nombre)
        break
    print("Revisando:", nombre)
```

Salida:

```
Revisando: Ana
Revisando: Luis
¡Encontrado! María
```

Sin el `break`, el ciclo seguiría revisando a Pedro y Sofía aunque ya encontró a María. ¿Para qué seguir buscando las llaves si ya las tienes en la mano?

`break` también funciona en `while`, y ahí es donde brilla con el juego de adivinanzas que verás en los ejemplos.

continue: sáltate esta vuelta

Mientras `break` abandona el ciclo por completo, `continue` solo se **salta la vuelta actual** y pasa a la siguiente. Es como estar pasando lista y decir "a este me lo salto, sigo con el que sigue".

```
for i in range(1, 11):
    if i % 2 == 0:
        continue # si es par, sáltate esta vuelta
    print(i)
```

Salida:

```
1
3
5
7
9
```

Cuando `i` es par, el `continue` hace que Python ignore el `print` y salte directo a la siguiente vuelta con el siguiente número. Solo se imprimen los impares.

Ejemplo práctico: procesar calificaciones ignorando las reprobadas.

```
calificaciones = [10, 6, 9, 5, 8, 4, 10]

for cal in calificaciones:
    if cal < 6:
        print(cal, "-> reprobado, lo salto")
        continue
    print(cal, "-> aprobado, lo cuento")
```

Analogía mexicana: `break` es salirte de la fila de las tortillas porque ya te acordaste de que tienes tortillas en casa (ya no necesitas nada más de la fila). `continue` es cuando la tortillera te pregunta "¿quieres de harina?" y dices "esas no, sigue con las de maíz": te saltas esa opción pero sigues en la fila.

El `else` en los ciclos: el final feliz

Python tiene un truco poco conocido: los ciclos `for` y `while` pueden llevar un `else` al final. El bloque `else` se ejecuta **cuando el ciclo termina de forma natural** (recorrió todo), pero ****NO** se ejecuta si saliste con `break` ******.

```
# Buscar un número: ¿el else se ejecuta?
for i in range(1, 6):
    if i == 10:
        print("¡Lo encontré!")
        break
else:
    print("Terminé de buscar y no lo encontré")
```

Como el 10 nunca aparece en `range(1, 6)`, nunca hay `break`, el ciclo termina natural y se ejecuta el `else`: "Terminé de buscar y no lo encontré".

Ahora con un caso donde Sí hay `break`:

```
for i in range(1, 6):
    if i == 3:
        print("¡Lo encontré!")
        break
else:
    print("Terminé de buscar y no lo encontré")
```

Aquí el `break` se activa en el 3, así que el `else` **no** se ejecuta. Salida: solo "¡Lo encontré!".

Memotécnica: el `else` del ciclo significa "**si no hubo break**". Es perfecto para búsquedas: "si recorrí todo y no hice `break`, entonces no lo encontré".

```
# ¿Hay un número mayor a 100 en la lista?
numeros = [10, 20, 30, 40]

for n in numeros:
    if n > 100:
        print("Encontré uno mayor a 100:", n)
        break
else:
    print("Ningún número es mayor a 100")
```

También funciona con `while`:

```

contador = 1
while contador <= 3:
    print(contador)
    contador = contador + 1
else:
    print("El while terminó sin break")

```

Advertencia honesta: el `else` en ciclos confunde a muchos programadores (incluso experimentados) porque esperan que funcione como el `else` del `if`. Úsalo principalmente para el patrón de búsqueda de arriba, donde su significado ("no se encontró") es claro. Si te confunde, no es obligatorio usarlo: puedes lograr lo mismo con una variable bandera.

Ciclos anidados: un ciclo dentro de otro

Puedes poner un ciclo dentro de otro ciclo. El de adentro se ejecuta **completo** por cada vuelta del de afuera. Es como las manecillas del reloj: por cada vuelta completa del horario (ciclo externo), el minutero da 12 vueltas (ciclo interno)... bueno, casi, pero se entiende la idea.

El ejemplo clásico: la tabla de multiplicar.

```

for tabla in range(1, 4):      # ciclo externo: tablas del 1 al 3
    print("Tabla del", tabla)
    for numero in range(1, 6): # ciclo interno: del 1 al 5
        print(tabla, "x", numero, "=", tabla * numero)
    print("---")

```

Salida:

```

Tabla del 1
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
---
Tabla del 2
2 x 1 = 2
...

```

Por cada valor de `tabla` (1, 2, 3), el ciclo interno recorre `numero` del 1 al 5 completo. Total de vueltas: $3 \times 5 = 15$.

Analogía de la tortillería: el ciclo externo es "para cada cliente en la fila", y el ciclo interno es "para cada tortilla que pidió ese cliente". Si hay 10 clientes y cada uno pide 5 tortillas, haces 50 tortillas. El de adentro siempre termina su chamba antes de que el de afuera avance.

Otro ejemplo visual: dibujar un rectángulo con asteriscos.

```
filas = 3
columnas = 5

for f in range(filas):
    linea = ""
    for c in range(columnas):
        linea = linea + "* "
    print(linea)
```

Salida:

```
* * * * *
* * * * *
* * * * *
```

El ciclo externo controla las filas (3 vueltas), el interno arma cada fila con 5 asteriscos. Los ciclos anidados son la base de los videojuegos de cuadritos (Tetris, buscaminas), de las tablas y de cualquier cosa con filas y columnas.

Cuidado: no anides más de 2-3 niveles si puedes evitarlo. Cada nivel multiplica el trabajo: $10 \times 10 = 100$ vueltas, pero $10 \times 10 \times 10 = 1,000$. Con 4 niveles de 100 ya son 100 millones. Los ciclos anidados profundos son lentos; úsalos con medida.

¿Cuándo usar `while` y cuándo `for`?

La pregunta del millón. Regla práctica:

- ****Usa `for` cuando sabes (o puedes calcular) cuántas veces**** vas a repetir: recorrer una lista, un string, un rango de números, una tabla. El `for` es más seguro porque no se vuelve infinito.
- ****Usa `while` cuando repites hasta que algo pase****, sin saber cuántas vueltas serán: esperar una contraseña correcta, un menú que se repite hasta que el usuario elige "salir", leer datos hasta que se acaben, simular un juego por turnos.


```
# FOR: sé exactamente cuántas veces (10 calificaciones)
calificaciones = [8, 9, 7, 10, 6, 8, 9, 7, 10, 8]
suma = 0
for cal in calificaciones:
    suma = suma + cal
print("Promedio:", suma / len(calificaciones))

# WHILE: no sé cuántas vueltas serán (hasta que adivine)
secreto = 7
intento = int(input("Adivina el número (1-10): "))
while intento != secreto:
    print("No es ese, intenta de nuevo.")
    intento = int(input("Adivina el número (1-10): "))
print("¡Correcto!")
```

¿Se puede usar `while` para todo lo del `for` y viceversa? Técnicamente sí, pero sería como usar un cuchillo para untar mantequilla pudiendo usar un cuchillo de untar: funciona, pero no es lo ideal. Cada ciclo tiene su terreno.

Analogías para no olvidar

El DJ con playlist en repeat: el `for` con `range()` es el DJ que tiene una playlist de 10 canciones: sabe exactamente cuántas va a poner y en qué orden. El `while` es el DJ al que le dijeron "sigue tocando mientras la gente siga bailando": no sabe cuántas canciones pondrá, solo sabe cuándo parar. El `break` es cuando se va la luz y se acaba la fiesta a la mitad. El `continue` es cuando se salta una canción que a nadie le gusta.

La tortillería: `while` quedan clientes en la fila: atiende al siguiente. No sabes cuántos clientes llegarán (`while`). `for cliente in fila_actual`: sí sabes cuántos hay formados ahorita (`for`). Si se acaba la masa: `break` (cierras). Si un cliente solo viene a preguntar precios: `continue` (lo saltas, sigues con el siguiente).

El semáforo: `while True`: (ciclo infinito a propósito) con decisiones booleanas adentro: `if verde and hay_carros`: avanzar. Los semáforos **SÍ** usan ciclos infinitos intencionales: están diseñados para no terminar nunca. La diferencia con un ciclo infinito accidental es que el semáforo fue diseñado así y tiene forma de apagarse.

Errores comunes (la sección que te va a ahorrar horas)

****Error 1: olvidar incrementar el contador en while.**** El clásico. El ciclo infinito más común del mundo.

```
# MAL: ciclo infinito
i = 1
while i <= 5:
    print(i)
    # ¡falta i = i + 1!

# BIEN
i = 1
while i <= 5:
    print(i)
    i = i + 1
```

****Error 2: off-by-one con range().**** Esperar que `range(5)` incluya el 5.

```
# Si quieres del 1 al 10, necesitas range(1, 11), no range(1, 10)
for i in range(1, 10):
    print(i) # imprime 1-9, ¡falta el 10!
```

Truco: cuando dudes, prueba con `list(range(...))` en la consola para ver qué números genera.

****Error 3: modificar la variable del for esperando que cambie el ciclo.**** No funciona: el `for` ignora tus cambios a la variable de control.

```
for i in range(5):
    print(i)
    i = i + 10 # ¡esto NO afecta la siguiente vuelta! i seguirá siendo 0,1,2,3,4
```

El `for` toma el siguiente valor de la secuencia sin importar lo que le hiciste a `i`. Si necesitas controlar el avance manualmente, usa `while`.

****Error 4: condición del while al revés.**** Escribir `while i >= 5` cuando `i` empieza en 1: la condición ya es falsa y el ciclo nunca entra. O peor: una condición que nunca se vuelve falsa.

****Error 5: confundir and con or en validaciones. "Mientras la edad sea menor a 0 y**** mayor a 120" - ¡ningún número es las dos cosas a la vez! Debe ser `or`:

```
# MAL: esto nunca es True, el ciclo nunca valida nada
while edad < 0 and edad > 120:

# BIEN: inválida si es negativa 0 si es mayor a 120
while edad < 0 or edad > 120:
```

Este error es tan común que tiene nombre: es la confusión entre "lo que quiero permitir" y "lo que quiero rechazar". Para rechazar, casi siempre necesitas `or` (cualquiera de las razones basta para rechazar).

****Error 6: usar `=` en lugar de `==` dentro de condiciones.**** En el `while` / `if` necesitas comparar (`==`), no asignar (`=`).

```
# MAL: SyntaxError
while intento = secreto:

# BIEN
while intento != secreto:
```

Error 7: indentación inconsistente en ciclos anidados. Cada nivel de ciclo lleva su propia indentación. Si mezclas espacios y tabs, o indentas mal, Python se confunde o hace algo distinto a lo que querías.

Buenas prácticas

1. ****Prefiere `for` sobre `while` cuando sepas el número de repeticiones.**** Es más legible y a prueba de infinitos.
2. **Usa paréntesis en expresiones booleanas mixtas.** `a or (b and c)` se lee mejor que `a or b and c`.
3. **Pon la condición más barata primero** aprovechando el cortocircuito: `if lista and lista[0] == "x"` es seguro; al revés daría error con lista vacía.
4. **Nombra bien tus variables de ciclo.** `for i in range(10)` está bien para algo rápido, pero `for estudiante in estudiantes` se entiende solo. Evita nombres de una letra en código que otros van a leer.
5. ****No uses `while True` sin un `break` claro.**** Un `while True` es válido (menús, servidores), pero asegúrate de que haya una salida visible.

6. **Extrae condiciones complejas a variables con nombre.** En lugar de un `if` gigante, crea banderas legibles:

```
es_mayor = edad >= 18
tiene_ine = ine_vigente
puede_votar = es_mayor and tiene_ine and not suspendido
```

7. **Prueba tus ciclos con valores pequeños primero.** Antes de procesar un millón de datos, prueba con 5. Si algo está mal, lo verás rápido y sin esperar.
8. ****Cuidado con modificar una lista mientras la recorres con `for`.** Es una trampa clásica que da resultados raros. Si necesitas filtrar, crea una lista nueva.

El menú interactivo: `while True` bien usado

Hay un patrón con `while True` que es totalmente legítimo y lo verás en cajeros automáticos, videojuegos y apps de consola: el **menú que se repite hasta que el usuario elige salir**. La clave es que la salida (`break`) sea obvia y esté siempre visible.

```
while True:
    print("\n--- TAQUERÍA EL BUCLE ---")
    print("1. Ver menú")
    print("2. Pedir tacos")
    print("3. Salir")

    opcion = input("Elige una opción: ")

    if opcion == "1":
        print("Tacos al pastor, de asada, de tripa... ¡todos a $15!")
    elif opcion == "2":
        cuantos = int(input("¿Cuántos tacos quieres? "))
        total = 0
        for i in range(cuantos):
            total = total + 15
        print("Tu total es: $", total)
    elif opcion == "3":
        print("¡Gracias por tu visita! Vuelve pronto. ")
        break # <- la salida: clara y al final
    else:
        print("Opción no válida, intenta de nuevo.")
```

Fíjate en el diseño: el ciclo no tiene condición de salida en el `while` (es `True` eterno), pero la opción 3 siempre ofrece el `break`. El `else` final atrapa opciones inválidas sin romper nada. Este esqueleto te sirve para cualquier programa con menú: una calculadora, una agenda, un inventario. Cámbialo por tu negocio y ya tienes una app.

Un detalle fino: pusimos `print()` con `\n` al inicio del menú para separar visualmente cada vuelta. Pequeños detalles así hacen que tus programas se sientan profesionales.

Negando condiciones compuestas: el truco intuitivo de De Morgan

Hay un par de equivalencias lógicas que te van a salvar cuando necesites negar una condición completa. No necesitas memorizar nombres raros, solo la idea:

- Negar un `and` lo convierte en `or` (negando cada parte): `not (A and B)` es lo mismo que `(not A) or (not B)`.
- Negar un `or` lo convierte en `and` (negando cada parte): `not (A or B)` es lo mismo que `(not A) and (not B)`.

En lenguaje de todos los días: "NO es cierto que (tengo hambre Y tengo dinero)" significa lo mismo que "(NO tengo hambre) O (NO tengo dinero)". Piénsalo: ¿cuándo es falso que tienes hambre y dinero a la vez? Cuando te falta al menos una de las dos cosas. ¡Exacto!

```
tengo_hambre = True
tengo_dinero = False

# Forma 1: negar todo junto
puedo_comprar = not (tengo_hambre and tengo_dinero)
print(puedo_comprar) # True (no puedo comprar: me falta dinero)

# Forma 2: negar por partes (equivalente)
puedo_comprar2 = (not tengo_hambre) or (not tengo_dinero)
print(puedo_comprar2) # True (igualito)
```

¿Para qué sirve en la práctica? Para escribir condiciones de salida de `while` más legibles. Compara:

```
# Versión 1: condición positiva con not afuera
while not (usuario == "admin" and password == "1234"):
    ...

# Versión 2: equivalente, a veces más clara
while usuario != "admin" or password != "1234":
    ...
```

Las dos hacen exactamente lo mismo: seguir pidiendo datos mientras NO se cumplan ambas credenciales. La versión 2 evita el `not` con paréntesis gigantes y muchos la encuentran más natural de leer. Con la práctica, elegirás la que se lea mejor en cada caso. Lo importante es que sepas que son intercambiables y por qué.

Depurando ciclos: el `print` es tu mejor amigo

Cuando un ciclo no hace lo que esperas (se repite de más, de menos, o nunca entra), la técnica de depuración más efectiva y más usada en el mundo real es ridículamente simple: ****mete `print()` dentro del ciclo para ver qué está pasando**** en cada vuelta.

```
# Algo anda mal... ¿por qué imprime 6 números si pedí 5?
n = 5
i = 1
while i <= n:
    print("vuelta:", i, "| condición i <= n:", i <= n) # <- el detective
    print("Número:", i)
    i = i + 1
```

Ese `print` detective te muestra el valor de `i` y de la condición en cada vuelta. En dos segundos ves si el contador avanza, si la condición se evalúa como creías, o si una variable vale algo inesperado. Los programadores profesionales hacen esto todos los días; no es trampa, es oficio.

Tres preguntas de diagnóstico cuando un ciclo se porta mal:

1. **¿El ciclo nunca entra?** Imprime la condición antes del `while`. Probablemente ya es `False` desde el inicio (condición al revés o variable mal inicializada).
2. **¿El ciclo no termina?** Imprime la variable de control dentro del ciclo. Probablemente no se está actualizando (olvidaste el incremento) o la condición nunca puede volverse falsa.

3. **¿Da una vuelta de más o de menos?** Revisa los límites: `<=` vs `<`, y recuerda que `range()` no incluye el fin. El off-by-one vive aquí.

Y un consejo de oro: cuando depures, trabaja con **números pequeños**. Si tu ciclo debe procesar 1,000 elementos y algo falla, pruébalo con 5. Ver 5 líneas de salida es fácil; ver 1,000 es imposible.

Ejemplos paso a paso

Ejemplo 1: Acceso con usuario Y contraseña (operador `and`)

Un sistema de login básico: para entrar necesitas que el usuario **y** la contraseña sean correctos. Las dos cosas, como el taquero exigente.

```
# Datos registrados en el sistema
usuario_registrado = "esteban"
password_registrado = "tacos123"

# Datos que escribe la persona
usuario = input("Usuario: ")
password = input("Contraseña: ")

# Las DOS condiciones deben cumplirse
if usuario == usuario_registrado and password == password_registrado:
    print("¡Bienvenido al sistema, Esteban!")
else:
    print("Usuario o contraseña incorrectos.")
```

Explicación línea por línea:

1. `usuario_registrado = "esteban"` - Guardamos el usuario correcto (en un sistema real estaría en una base de datos).
2. `password_registrado = "tacos123"` - Guardamos la contraseña correcta.
3. `usuario = input("Usuario: ")` - Pedimos el usuario por teclado.
4. `password = input("Contraseña: ")` - Pedimos la contraseña.
5. `if usuario == usuario_registrado and password == password_registrado:` - La condición completa: el usuario escrito debe ser igual al registrado **Y** la contraseña escrita debe ser igual a la registrada. Si alguna falla, todo es `False`.

6. `print("¡Bienvenido...")` - Solo se ejecuta si AMBAS coincidieron.
7. `else: / print("Usuario o contraseña incorrectos.")` - Si algo falló, mensaje genérico (por seguridad no decimos cuál de los dos falló).

Prueba mental: usuario "esteban" + password "tacos123" -> True and True -> True -> bienvenido. Usuario "esteban" + password "otra" -> True and False -> False -> acceso denegado. Exacto como un login real.

Ejemplo 2: Descuento con `or` (con una condición basta)

Una tienda en línea da 15% de descuento si eres estudiante **o** adulto mayor **o** es tu cumpleaños. Con una basta, como la abuelita.

```
# Datos del cliente
es_estudiante = input("¿Eres estudiante? (s/n): ") == "s"
es_adulto_mayor = input("¿Eres adulto mayor? (s/n): ") == "s"
es_cumpleanos = input("¿Hoy es tu cumpleaños? (s/n): ") == "s"

# Con UNA condición verdadera basta para el descuento
if es_estudiante or es_adulto_mayor or es_cumpleanos:
    print("¡Felicidades! Tienes 15% de descuento.")
else:
    print("Precio normal. ¡Gracias por tu compra!")
```

Explicación línea por línea:

1-3. Cada `input(...) == "s"` convierte la respuesta en booleano: si el usuario escribe "s", la variable vale `True`; si escribe otra cosa, `False`. Es un truco compacto muy usado.

4. `if es_estudiante or es_adulto_mayor or es_cumpleanos:` - Si AL MENOS UNA es verdadera, hay descuento. El `or` encadenado funciona: `True or False or False -> True`.

5-6. Mensajes según el resultado.

Prueba mental: estudiante "n", adulto mayor "n", cumpleaños "s" -> `False or False or True -> True` -> ¡descuento de cumpleaños! Las tres "n" -> `False -> False` -> precio normal.

Ejemplo 3: `while` contando hasta 10 (contador básico)

El "hola mundo" de los ciclos: contar del 1 al 10 con `while`.


```
# Inicializamos el contador
numero = 1

# Mientras no pasemos de 10, seguimos contando
while numero <= 10:
    print("Número:", numero)
    numero = numero + 1 # ¡incremento! sin esto sería infinito

print("¡Terminamos de contar!")
```

Explicación línea por línea:

1. `numero = 1` - **Inicialización**: el contador empieza en 1.
2. `while numero <= 10:` - **Condición**: ¿seguimos? Mientras `numero` sea 10 o menos, sí.
3. `print("Número:", numero)` - **Acción**: mostramos el número actual.
4. `numero = numero + 1` - **Actualización**: aumentamos el contador para acercarnos al final. ESTA línea es la que evita el ciclo infinito.
5. `print("¡Terminamos de contar!")` - Fuera del ciclo (sin indentación): se ejecuta una sola vez al terminar.

Recorrido mental: `numero=1` -> imprime -> `numero=2` ->... -> `numero=10` -> imprime -> `numero=11` -> condición `11 <= 10` es `False` -> sale -> imprime el mensaje final. Total: 10 vueltas exactas.

Ejemplo 4: Adivinanza con `while` y `break` (el juego)

El juego clásico: la computadora piensa un número y tú lo adivinas. El ciclo se repite hasta que adivinas, y el `break` nos saca en cuanto acertamos. Además damos pistas de "más alto / más bajo".

```

# La computadora "piensa" un número (fijo para el ejemplo)
secreto = 7
intentos = 0

print("Adivina el número entre 1 y 10.")

while True: # ciclo "infinito" controlado con break
    intento = int(input("Tu número: "))
    intentos = intentos + 1

    if intento == secreto:
        print("¡Correcto! Lo adivinaste en", intentos, "intentos.")
        break # salimos del ciclo: ya ganó
    elif intento < secreto:
        print("Más alto... sigue intentando.")
    else:
        print("Más bajo... sigue intentando.")

print("Fin del juego. ¡Gracias por jugar!")

```

Explicación línea por línea:

1. `secreto = 7` - El número a adivinar (en un juego real usarías `random.randint(1, 10)`).
2. `intentos = 0` - Contador de intentos, empieza en cero.
3. `print(...)` - Mensaje inicial.
4. `while True:` - Ciclo que en teoría no termina solo; confiamos en el `break` de adentro. Es el patrón válido de `while True` con salida clara.
5. `intento = int(input("Tu número: "))` - Pedimos el número y lo convertimos a entero.
6. `intentos = intentos + 1` - Contamos este intento.
7. `if intento == secreto:` - ¿Adivinó?
8. `print("¡Correcto!...")` - Felicitamos y decimos cuántos intentos tomó.
9. `break` - **Salida de emergencia:** rompemos el ciclo porque el juego terminó.
- 10-11. `elif intento < secreto:` - Si no adivinó y su número es menor, pista: el secreto es más alto.
- 12-13. `else:` - Si su número es mayor, pista: el secreto es más bajo.
14. `print("Fin del juego...")` - Fuera del ciclo: solo se llega aquí después del `break`.

Prueba mental: secreto=7. Usuario dice 3 -> "Más alto". Dice 9 -> "Más bajo". Dice 7 -> "¡Correcto! Lo adivinaste en 3 intentos." -> break -> "Fin del juego".

Ejemplo 5: Tabla de multiplicar con for (ciclos anidados)

Imprimir las tablas del 1 al 5 usando dos ciclos for anidados.

```
# Ciclo externo: qué tabla estamos haciendo (1 a 5)
for tabla in range(1, 6):
    print("=== Tabla del", tabla, "===")

    # Ciclo interno: multiplicar del 1 al 10
    for numero in range(1, 11):
        resultado = tabla * numero
        print(tabla, "x", numero, "=", resultado)

    print() # línea vacía para separar las tablas
```

Explicación línea por línea:

1. for tabla in range(1, 6): - **Ciclo externo**: tabla tomará los valores 1, 2, 3, 4, 5.
2. print("=== Tabla del", tabla, "===") - Encabezado de cada tabla.
3. for numero in range(1, 11): - **Ciclo interno**: por CADA valor de tabla, numero recorre 1 a 10 completo.
4. resultado = tabla * numero - Multiplicamos.
5. print(tabla, "x", numero, "=", resultado) - Mostramos "3 x 4 = 12".
6. print() - Línea vacía (fuera del ciclo interno, dentro del externo) para separar visualmente.

Recorrido mental: tabla=1 -> ciclo interno imprime 1x1...1x10 -> línea vacía -> tabla=2 -> ciclo interno otra vez 2x1...2x10 ->... Total: $5 \times 10 = 50$ multiplicaciones. El ciclo interno siempre se completa antes de que el externo avance: esa es la regla de los anidados.

Ejemplo 6: for con range de paso 2 y hacia atrás

Dos trucos de range() en un solo programa: contar de 2 en 2 y hacer cuenta regresiva.

```

print("Números pares del 0 al 20:")
for par in range(0, 21, 2):
    print(par, end=" ")
print() # salto de línea
print()

print("Cuenta regresiva para el cohete:")
for cuenta in range(10, 0, -1):
    print(cuenta)
print("¡Despegue! ")

```

Explicación línea por línea:

1. `print("Números pares del 0 al 20:")` - Título.
2. `for par in range(0, 21, 2):` - range con **paso 2**: empieza en 0, termina antes del 21, avanzando de 2 en 2 -> 0, 2, 4,..., 20. (El 21 es el fin no incluido; el último valor real es 20.)
3. `print(par, end=" ")` - `end=""` hace que los números salgan en la misma línea separados por espacio, en vez de uno por línea.
- 4-5. `print()` dos veces - Saltos de línea para separar secciones.
6. `print("Cuenta regresiva...")` - Título.
7. `for cuenta in range(10, 0, -1):` - range con **paso negativo**: empieza en 10, baja de 1 en 1, se detiene antes de llegar al 0 -> 10, 9, 8,..., 1. (El 0 no se incluye.)
8. `print(cuenta)` - Cada número en su línea, como la cuenta de Año Nuevo.
9. `print("¡Despegue! ")` - Fuera del ciclo: el gran final.

Salida:

```
Números pares del 0 al 20:  
0 2 4 6 8 10 12 14 16 18 20
```

```
Cuenta regresiva para el cohete:  
10  
9  
8  
7  
6  
5  
4  
3  
2  
1  
¡Despegue!
```

Práctica

Ejercicio 1: ¿Puedes entrar al concierto?

Enunciado: Escribe un programa que decida si una persona puede entrar a un concierto. Las reglas: puede entrar si tiene boleto **o** está en la lista de invitados, **y** además debe ser mayor de edad **y** no estar en la lista de vetados. Pide los cuatro datos al usuario (responde s/n) e imprime si puede entrar o no.

Solución:

```
# Pedimos los datos y los convertimos a booleanos  
tiene_boleto = input("¿Tienes boleto? (s/n): ") == "s"  
en_lista = input("¿Estás en la lista de invitados? (s/n): ") == "s"  
es_mayor = input("¿Eres mayor de edad? (s/n): ") == "s"  
vetado = input("¿Estás vetado? (s/n): ") == "s"  
  
# Regla: (boleto O lista) Y mayor Y NO vetado  
puede_entrar = (tiene_boleto or en_lista) and es_mayor and not vetado  
  
if puede_entrar:  
    print("¡Adelante! Disfruta el concierto. ")  
else:  
    print("Lo sentimos, no puedes entrar.")
```

Explicación paso a paso:

1. Las primeras cuatro líneas convierten cada respuesta "s/n" en True / False con el truco `input(...) == "s"`.
2. La expresión clave: `(tiene_boleto or en_lista) and es_mayor and not vetado`. Los paréntesis agrupan la primera parte: basta con tener boleto o estar en la lista. Luego ese resultado se combina con `and`: además debe ser mayor y no estar vetado (not vetado).
3. Sin los paréntesis, la precedencia (and antes que or) cambiaría el significado: sería `tiene_boleto or (en_lista and es_mayor and not vetado)`, o sea que con solo tener boleto entrarías aunque fueras menor. ¡Los paréntesis importan!
4. Prueba mental: boleto=s, lista=n, mayor=s, vetado=n -> `(True or False) and True and not False -> True and True and True -> True -> ¡adentro!`
5. Prueba mental 2: boleto=s, lista=n, mayor=n, vetado=n -> `(True) and False and True -> False -> no entra (es menor, ni modo).`

Ejercicio 2: Tabla de verdad del or con código

Enunciado: Escribe un programa que imprima la tabla de verdad completa del operador `or` usando dos ciclos `for` anidados. Debe mostrar las 4 combinaciones de True / False.

Solución:

```
print("A\t\tB\t\tA or B")
print("-" * 30)

for a in [True, False]:
    for b in [True, False]:
        resultado = a or b
        print(a, "\t", b, "\t", resultado)
```

Explicación paso a paso:

1. `for a in [True, False]:` - El ciclo externo recorre los dos valores posibles de A. (Un `for` puede recorrer una lista directamente, no solo `range()`.)
2. `for b in [True, False]:` - El ciclo interno recorre los dos valores de B **por cada** valor de A. Total: $2 \times 2 = 4$ combinaciones.
3. `resultado = a or b` - Evaluamos el `or` con la combinación actual.

4. `print(a, "\t", b, "\t", resultado)` - `"\t"` es un tabulador: alinea las columnas para que se vea como tabla.
5. Salida:

```
A  B  A or B
-----
True True  True
True False  True
False True   True
False False  False
```

6. Observa: el `or` solo es `False` en la última fila, cuando ambos son falsos. ¡La tabla de verdad generada por código!

Ejercicio 3: Suma con `while` y acumulador

Enunciado: Pide al usuario 5 números (uno por uno) usando un ciclo `while`, súmalos con un acumulador e imprime el total y el promedio al final.

Solución:

```
# Inicializamos contador y acumulador
contador = 1
suma = 0

# Pedimos 5 números
while contador <= 5:
    numero = float(input("Dame el número " + str(contador) + ": "))
    suma = suma + numero      # acumulador: junta todo
    contador = contador + 1   # incremento: evita el infinito

promedio = suma / 5
print("Suma total:", suma)
print("Promedio:", promedio)
```

Explicación paso a paso:

1. `contador = 1` y `suma = 0` - Inicializamos las dos variables del patrón: el contador controla las vueltas, el acumulador junta los valores. El acumulador **siempre** empieza en 0 para sumas (elemento neutro).
2. `while contador <= 5:` - Repetimos exactamente 5 veces.

3. `numero = float(input(...))` - Pedimos un número. Usamos `float` para aceptar decimales. El `"Dame el número"+ str(contador)` muestra "Dame el número 1:", "Dame el número 2:"... (`str()` convierte el entero a texto para poder concatenar.)
4. `suma = suma + numero` - El acumulador crece con cada número.
5. `contador = contador + 1` - Sin esta línea, ciclo infinito. Con ella, el ciclo termina tras 5 vueltas.
6. `promedio = suma / 5` - Fuera del ciclo: calculamos el promedio una sola vez.
7. Prueba mental: números 10, 20, 30, 40, 50 -> suma=150, promedio=30.0.

Ejercicio 4: Adivina el número con límite de intentos

Enunciado: Modifica el juego de adivinanza: el jugador tiene máximo 5 intentos. Si adivina, felicítalo con `break`; si se acaban los intentos, usa el `else` del `while` para decirle cuál era el número.

Solución:

```
secreto = 7
intentos = 0
max_intentos = 5

print("Adivina el número (1-10). Tienes", max_intentos, "intentos.")

while intentos < max_intentos:
    intento = int(input("Intento " + str(intentos + 1) + ": "))
    intentos = intentos + 1

    if intento == secreto:
        print("¡Correcto! Ganaste en", intentos, "intentos. ")
        break
    elif intento < secreto:
        print("Más alto...")
    else:
        print("Más bajo...")
else:
    # Este else solo se ejecuta si NO hubo break (no adivinó)
    print("Se acabaron los intentos. El número era", secreto, ". ¡Suerte para la próxima!")
```

Explicación paso a paso:

1. `intentos = 0, max_intentos = 5` - Controlamos las vueltas con el contador de intentos.
2. `while intentos < max_intentos:` - El ciclo termina solo de dos formas: adivinando (`break`) o agotando los 5 intentos (condición falsa).
3. Dentro: pedimos el número, contamos el intento, comparamos con el secreto dando pistas.
4. `if intento == secreto: ... break` - Si adivina, felicitamos y rompemos el ciclo inmediatamente.
5. El `else:` del `while` - Aquí está la magia: ****solo se ejecuta si el ciclo terminó sin `break`**, es decir, si se acabaron los intentos sin adivinar. Es el "final triste" del juego.**
6. Prueba mental (gana): intentos 3, 9, 7 -> en el tercer intento `break` -> el `else` NO se ejecuta -> "¡Correcto! Ganaste en 3 intentos."
7. Prueba mental (pierde): 1, 2, 3, 4, 5 -> intentos llega a 5 -> condición falsa -> sale sin `break` -> se ejecuta el `else` -> "Se acabaron los intentos. El número era 7."

Ejercicio 5: Pirámide de asteriscos con ciclos anidados

Enunciado: Pide al usuario un número `n` e imprime una pirámide (triángulo) de asteriscos de `n` filas: la fila 1 tiene 1 asterisco, la fila 2 tiene 2, etc.

Solución:

```
n = int(input("¿De cuántas filas quieres la pirámide? "))

for fila in range(1, n + 1):
    linea = ""
    for columna in range(fila):
        linea = linea + "* "
    print(linea)
```

Explicación paso a paso:

1. `n = int(input(...))` - Pedimos el tamaño.
2. `for fila in range(1, n + 1):` - El ciclo externo recorre las filas: 1, 2, 3,..., `n`. (Usamos `n + 1` porque `range` no incluye el fin.)
3. `linea = ""` - Empezamos cada fila con un string vacío (acumulador de texto, reiniciado en cada vuelta externa).

4. `for column in range(fila):` - El ciclo interno repite **tantas veces como indique la fila actual**: en la fila 1, una vez; en la fila 2, dos veces; etc. ¡El `range` interno depende de la variable del externo!
5. `linea = linea + "*" "` - Acumulamos un asterisco por vuelta interna.
6. `print(linea)` - Al terminar el ciclo interno, imprimimos la fila completa.
7. Con `n=4`, la salida es:

```
*  
* *  
* * *  
* * * *
```

Ejercicio 6: Validador de contraseña con `while` y booleanos

Enunciado: Pide una contraseña hasta que cumpla TODAS estas reglas: al menos 8 caracteres **y** contiene al menos un número **y** no contiene espacios. Usa `while` para repetir y operadores booleanos para las reglas. Al final confirma que es válida.

Solución:

```

print("Crea tu contraseña.")
print("Reglas: mínimo 8 caracteres, al menos un número, sin espacios.")

while True:
    pwd = input("Contraseña: ")

    # Evaluamos cada regla por separado (banderas con nombre)
    largo_ok = len(pwd) >= 8
    tiene_numero = False
    for caracter in pwd:
        if caracter >= "0" and caracter <= "9":
            tiene_numero = True
            break
    sin_espacios = " " not in pwd

    # Las TRES reglas deben cumplirse
    if largo_ok and tiene_numero and sin_espacios:
        print("¡Contraseña válida! [OK]")
        break
    else:
        print("No cumple las reglas. Revisa:")
        if not largo_ok:
            print("- Debe tener al menos 8 caracteres.")
        if not tiene_numero:
            print("- Debe contener al menos un número.")
        if not sin_espacios:
            print("- No debe contener espacios.")

```

Explicación paso a paso:

1. `while True:` - Repetimos hasta que la contraseña sea válida; el `break` nos saca.
2. `largo_ok = len(pwd) >= 8` - `len()` mide la longitud del string; la regla es booleana.
3. Para `tiene_numero`: recorremos la contraseña **letra por letra** con `for`; si un caracter está entre `"0"` y `"9"` (comparación de strings que funciona por orden alfabético/númeroico), marcamos `True` y rompemos con `break` (ya encontramos uno, no hay que seguir buscando).
4. `sin_espacios = " " not in pwd` - `in / not in` con strings: pregunta si el espacio aparece en el texto. Elegante y legible.
5. `if largo_ok and tiene_numero and sin_espacios:` - Las tres banderas con `and`: todas deben ser `True`.

6. Si falla, el `else` dice **exactamente qué reglas fallaron** usando `not` en cada bandera: retroalimentación útil en vez de un genérico "está mal".
7. Prueba mental: "taco12" -> `largo_ok=False` ($6 < 8$), `tiene_numero=True`, `sin_espacios=True` -> `False and True and True` -> `False` -> muestra "- Debe tener al menos 8 caracteres." y repite. "tacos1234" -> las tres `True` -> "¡Contraseña válida!" -> `break`.

Resumen de la unidad

Llegaste al final de la Unidad 5. Aquí está todo lo esencial, compacto:

Lógica booleana:

- Los valores booleanos son `True` y `False`; toda comparación produce uno.
- `and` (y): `True` solo si **ambas** condiciones son verdaderas. El taquero exigente.
- `or` (o): `True` si **al menos una** es verdadera. La abuelita buena onda.
- `not` (no): invierte el valor. El rebelde.
- Precedencia: `not` primero, luego `and`, al final `or`. Ante la duda, **usa paréntesis**.
- Cortocircuito: Python no evalúa lo que no necesita. Pon primero la condición barata y segura (`if lista and lista[0] == ...`).

Ciclos:

- `while condicion`: repite **mientras** la condición sea verdadera. Ideal cuando no sabes cuántas vueltas serán (validar entrada, juegos, menús).
- Regla de oro del `while`: algo dentro del ciclo debe acercar la condición a volverse falsa. Si no, **ciclo infinito**. Se interrumpe con **Ctrl + C**.
- Patrones `while`: **contador** (`c = c + 1`) y **acumulador** (`suma = suma + valor`).
- `for variable in range(...)`: repite un número conocido de veces. `range(n)` -> 0 a `n-1`; `range(inicio, fin)` -> inicio a `fin-1`; `range(inicio, fin, paso)` -> con saltos (paso negativo = hacia atrás).
- El `for` también recorre strings letra por letra y listas elemento por elemento.
- `break`: sale del ciclo inmediatamente (salida de emergencia).
- `continue`: se salta solo la vuelta actual.
- `else` en ciclos: se ejecuta ****si no hubo `break`**** (ideal para "no se encontró").

- Ciclos anidados: el interno se completa por cada vuelta del externo (tablas de multiplicar, dibujos, filas × columnas).
- ¿while o for? ¿Sabes cuántas veces? -> for. ¿Repites hasta que pase algo? -> while.

Errores que ya no vas a cometer: olvidar el incremento (infinito), el off-by-one de `range()`, modificar la variable del `for`, confundir `and` con `or` al validar, y el `=` en vez de `==`.

En la próxima unidad seguiremos construyendo sobre esta base: ahora que tus programas deciden y repiten, les vamos a dar memoria más sofisticada. Por lo pronto, practica: combina booleanos, anida ciclos, rompe cosas a propósito (con Ctrl + C a la mano) y vuelve a armarlas. Esa es la forma en que se aprende a programar de verdad: decidiendo y repitiendo, igual que tus programas. ¡Nos vemos en la Unidad 6!

UNIDAD 6: Listas y tuplas - tu caja de herramientas de colecciones

Introducción

Imagina que vas al súper. Llevas en la cabeza la lista: leche, huevos, tortillas, queso, jamón, pan. Si solo fueran tres cosas, las recuerdas de memoria. Pero cuando la lista crece a treinta productos, tu cerebro dice "basta" y sacas tu celular para anotarla.

En Python pasa exactamente lo mismo. Cuando tu programa necesita manejar un solo dato, una variable es suficiente: `edad = 25`. Pero cuando necesitas manejar las edades de todos los alumnos de tu salón, ¿vas a crear `edad1 = 18`, `edad2 = 19`, `edad3 = 20`... hasta `edad50`? Claro que no. Eso sería como llevar la lista del súper repartida en cincuenta papelitos sueltos. Pierdes uno y ya no sabes ni qué compraste.

Para eso existen las **listas**: una sola variable que guarda muchos valores ordenados, como una lista del súper. Y al lado de las listas viven las **tuplas**: primas hermanas de las listas, pero que una vez escritas ya no se pueden cambiar, como una foto impresa de tu álbum familiar - queda fija para siempre.

Piensa en tu celular. El álbum de fotos es una lista: puedes agregar fotos nuevas, borrar las que salieron borrosas, reordenarlas. Pero la **fecha de nacimiento** que registraste en tu perfil es como una tupla: es un dato fijo que no debería cambiar por accidente. Esa es la diferencia central de esta unidad, y créeme, la vas a usar en casi todos los programas que escribas de aquí en adelante.

En esta unidad vas a dominar:

- Qué es una lista, cómo crearla y por qué es mil veces mejor que cien variables sueltas.
- Los **índices**: la dirección exacta de cada elemento, incluyendo el truco de los índices negativos (empezar a contar desde el final).
- El **slicing**: cómo "cortar rebanadas" de una lista, igual que partes una torta.
- La **mutabilidad**: cómo cambiar, agregar y quitar elementos en caliente.
- Todos los métodos importantes: `append()`, `insert()`, `remove()`, `pop()`, `clear()`, `index()`, `count()`, `sort()`, `reverse()`, `extend()`, `copy()`.
- Las funciones que trabajan con listas: `len()`, `min()`, `max()`, `sum()`.

- Las **tuplas**: sintaxis, inmutabilidad, la famosa coma de la tupla de un elemento, y cuándo elegir tupla en vez de lista.
- Los errores clásicos que atorran a todo el mundo: el `IndexError`, confundir `remove()` con `pop()`, el `sort()` que devuelve `None`, y la trampa del alias contra la copia.

Al terminar esta unidad vas a poder hacer cosas como: calcular el promedio de las calificaciones de un grupo, mantener una agenda de contactos, procesar una lista de compras, manejar coordenadas de un mapa y eliminar duplicados de una lista sin perder el orden. Todo eso con unas cuantas líneas de código.

¿Listo? Saca tu lista del súper mental y vamos a programar.

Explicación detallada

Qué es una lista y por qué supera a 100 variables

Una **lista** es una colección ordenada de valores, encerrada entre **corchetes** `[]` y con los elementos separados por comas. Ordenada significa que cada elemento tiene una posición fija: el primero, el segundo, el tercero... y ese orden se conserva.

```
compras = ["leche", "huevos", "tortillas", "queso", "jamón"]
```

En una sola variable, `compras`, guardaste cinco cadenas. Ahora compáralo con la alternativa: crear cinco variables sueltas.

```
# La forma dolorosa (NO la uses):
compra1 = "leche"
compra2 = "huevos"
compra3 = "tortillas"
compra4 = "queso"
compra5 = "jamón"
```

Con cinco elementos ya se ve feo. Con cien, imposible. ¿Y cómo harías un ciclo para imprimir las cien variables? Tendrías que escribir cien `print()`. Con una lista, un solo `for` recorre todo:

```
compras = ["leche", "huevos", "tortillas", "queso", "jamón"]

for producto in compras:
    print("Comprar:", producto)
```

Resultado:

```
Comprar: leche
Comprar: huevos
Comprar: tortillas
Comprar: queso
Comprar: jamón
```

Esa es la primera gran lección: **una lista convierte cien variables en una, y cien líneas repetidas en un ciclo**. Los programas de verdad trabajan con colecciones de datos, no con variables sueltas.

Las listas pueden guardar **cualquier tipo de dato**, y hasta mezclarlos:

```
numeros = [10, 20, 30, 40, 50]
nombres = ["Ana", "Luis", "María", "Pedro"]
mezcla = ["Hola", 42, 3.14, True]
vacía = []
```

La lista vacía `[]` parece inútil, pero es de las más usadas: la creas vacía y la vas llenando poco a poco con `append()`, como una canasta del súper que empieza vacía y se llena pasillo por pasillo.

Índices: la dirección de cada elemento

Cada elemento de una lista tiene una **dirección** llamada **índice**. Y aquí viene la primera costumbre rara de Python (y de casi todos los lenguajes): **los índices empiezan en 0, no en 1**.

```
frutas = ["manzana", "plátano", "naranja", "uva", "mango"]
```

Índice	0	1	2	3	4	---	---	---	---	---	---	Valor
	manzana	plátano	naranja	uva	mango							

Para acceder a un elemento usas los corchetes con el índice:


```
frutas = ["manzana", "plátano", "naranja", "uva", "mango"]
```

```
print(frutas[0])    # manzana   (el PRIMERO)
print(frutas[2])    # naranja   (el tercero)
print(frutas[4])    # mango     (el último)
```

¿Por qué empezar en 0? Piensa en una regla de medir: el primer centímetro empieza en el 0 de la regla, no en el 1. O piensa en las casas de una calle: la distancia desde el inicio de la cuadra hasta la primera casa es 0 casas de distancia. El índice no es "el número del elemento", es **cuántos lugares hay que avanzar desde el inicio**. La primera casilla está a 0 pasos del inicio. Cuesta acostumbrarse, pero después de una semana lo haces sin pensar.

Índices negativos: contar desde el final

Python tiene un truco genial: los **índices negativos** cuentan desde el final hacia atrás. El -1 es el último elemento, el -2 el penúltimo, y así.

```
frutas = ["manzana", "plátano", "naranja", "uva", "mango"]

print(frutas[-1])   # mango     (el último)
print(frutas[-2])   # uva       (el penúltimo)
print(frutas[-5])   # manzana   (el primero, contado desde atrás)
```

Índice negativo	-5	-4	-3	-2	-1	---	---	---	---	---	---	Valor
	manzana	plátano	naranja	uva	mango							

Esto es oro puro cuando no sabes cuántos elementos tiene la lista. ¿Quieres el último elemento de una lista de mil? `lista[-1]`, sin necesidad de saber que tiene mil. En otros lenguajes tendrías que calcular `lista[len(lista) - 1]`. Python te ahorra ese rollo.

Slicing: cortar rebanadas de la lista

El **slicing** (rebanado) te permite obtener una **porción** de la lista usando la sintaxis `[inicio:fin]`. Es como partir una torta: pides de la rebanada 1 a la 3 y te llevas ese pedazo.

La regla de oro: **el inicio se incluye, el fin se excluye**. `[1:4]` toma los índices 1, 2 y 3 (el 4 se queda fuera).

```

numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numeros[2:6])    # [2, 3, 4, 5]    (del índice 2 al 5)
print(numeros[0:3])    # [0, 1, 2]      (los primeros tres)
print(numeros[5:])     # [5, 6, 7, 8, 9] (del 5 hasta el final)
print(numeros[:4])     # [0, 1, 2, 3]   (del inicio hasta el 3)
print(numeros[:])      # [0, 1, ..., 9] (copia completa)
print(numeros[-3:])    # [7, 8, 9]     (los últimos tres)
print(numeros[:-2])    # [0, 1, ..., 7] (todo menos los últimos dos)

```

Visualízalo así. La lista es una fila de casilleros:

Índice:	0	1	2	3	4	5	6	7	8	9	
Valor:	0	1	2	3	4	5	6	7	8	9	
											<- [2:6] toma 2,3,4,5
											<- [5:] toma del 5 al final

Si omites el inicio, empieza desde el 0. Si omites el fin, llega hasta el último. Y ojo: el slicing **siempre devuelve una lista nueva**, la original no se modifica.

El tercer parámetro: el paso

El slicing acepta un tercer número, el **paso** (step): [inicio:fin:paso]. El paso dice "avanza de tantos en tantos".

```

numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(numeros[::2])    # [0, 2, 4, 6, 8] (uno sí, uno no)
print(numeros[1::2])   # [1, 3, 5, 7, 9] (los impares)
print(numeros[::-1])   # [9, 8, 7, ..., 0] (¡lista invertida!)

```

`numeros[::-1]` es el truco más famoso de Python para invertir una lista: paso `-1` significa "camina hacia atrás". Memorízalo, lo vas a usar mucho.

Mutabilidad: las listas se pueden cambiar en caliente

Mutable significa "que se puede mutar, cambiar". Las listas son mutables: puedes cambiar cualquier elemento por su índice, sin crear una lista nueva.

```
colores = ["rojo", "verde", "azul"]

colores[1] = "amarillo" # cambio el segundo elemento
print(colores)          # ['rojo', 'amarillo', 'azul']
```

También puedes cambiar **rebanadas completas** de un jalón:

```
numeros = [1, 2, 3, 4, 5]

numeros[1:4] = [20, 30] # reemplazo los índices 1,2,3 por dos valores
print(numeros)          # [1, 20, 30, 5]
```

Fíjate: reemplazaste tres elementos por dos. La lista se ajusta sola, se hace más chica o más grande según lo que le asignes. Es como un acordeón: se estira y se encoge.

Esta mutabilidad es la diferencia más importante con las tuplas (que veremos más adelante) y es lo que hace a las listas tan útiles para programas que cambian datos constantemente: marcadores de un juego, inventario de una tienda, lista de tareas pendientes.

Recorrer una lista con for (y con enumerate)

Ya viste el `for` básico. Pero hay dos formas más de recorrer que te van a servir muchísimo.

Recorrido con índice (cuando necesitas saber en qué posición vas):

```
frutas = ["manzana", "plátano", "naranja"]

for i in range(len(frutas)):
    print(i, frutas[i])
```

```
0 manzana
1 plátano
2 naranja
```

****Recorrido con `enumerate()` **** (la forma elegante de Python):

```
frutas = ["manzana", "plátano", "naranja"]

for i, fruta in enumerate(frutas):
    print(f"Posición {i}: {fruta}")
```

```
Posición 0: manzana
Posición 1: plátano
Posición 2: naranja
```

`enumerate()` te da el índice y el valor al mismo tiempo. Es la forma "pitónica" (así le dicen los programadores a la forma elegante de hacer las cosas en Python) y te recomiendo acostumbrarte a ella desde ahora.

Listas dentro de listas: matrices simples

Una lista puede contener **otras listas**. Esto sirve para representar tablas, tableros o cuadrículas, como una hoja de cálculo o el tablero del gato (tic-tac-toe).

```
# Tabla de calificaciones: cada fila es un alumno, cada columna una materia
calificaciones = [
    [8, 9, 7],      # Ana:    mate 8, español 9, historia 7
    [10, 6, 8],     # Luis:   mate 10, español 6, historia 8
    [7, 7, 9],      # María:  mate 7, español 7, historia 9
]

print(calificaciones[0])      # [8, 9, 7]  (toda la fila de Ana)
print(calificaciones[1][0])   # 10        (fila 1, columna 0: mate de Luis)
print(calificaciones[2][2])   # 9          (fila 2, columna 2: historia de María)
```

Para acceder a un dato usas **doble índice**: `[fila][columna]`. Primero eliges la lista interior, luego el elemento dentro de ella. Es como decir "el cajón 2, la camisa 3" de tu clóset.

Un tablero de gato vacío se vería así:

```
tablero = [
    [" ", " ", " "],
    [" ", " ", " "],
    [" ", " ", " "],
]

tablero[0][0] = "X"  # pongo X en la esquina superior izquierda
tablero[1][1] = "O"  # pongo O en el centro

for fila in tablero:
    print(fila)
```

```
['x', ' ', ' ']  
[' ', 'o', ' ']  
[' ', ' ', ' ']
```

No necesitas dominar las matrices a fondo todavía; con saber que existen y cómo leerlas tienes suficiente por ahora.

Métodos de listas, uno por uno

Los **métodos** son funciones pegadas a la lista que la modifican o le sacan información. Se llaman con la sintaxis `lista.metodo()`. Vamos con cada uno, con ejemplo y caso de uso real.

`append()`: agregar al final

`append()` agrega **un** elemento al final de la lista. Es el método más usado de todos.

```
compras = ["leche", "huevos"]  
compras.append("tortillas")  
compras.append("queso")  
print(compras)    # ['leche', 'huevos', 'tortillas', 'queso']
```

Caso de uso: ir llenando una lista poco a poco, como cuando el usuario va agregando productos, o cuando un ciclo va juntando resultados.

`insert()`: agregar en una posición exacta

`insert(posición, valor)` mete el elemento **en el índice que le digas**, empujando a los demás una posición a la derecha.

```
fila = ["Ana", "Luis", "Pedro"]  
fila.insert(1, "María")    # María se mete en la posición 1  
print(fila)                # ['Ana', 'María', 'Luis', 'Pedro']
```

Caso de uso: cuando el orden importa y algo debe ir en un lugar específico, como insertar a alguien que llegó con cita en una fila de espera.

remove(): quitar por valor

`remove(valor)` busca la **primera aparición** del valor y la elimina. Ojo: trabaja con el **valor**, no con el índice.

```
compras = ["leche", "huevos", "tortillas", "huevos"]
compras.remove("huevos")    # quita el PRIMER "huevos"
print(compras)              # ['leche', 'tortillas', 'huevos']
```

Caso de uso: quitar un elemento cuando sabes qué dice pero no dónde está, como tachar "huevos" de la lista del súper.

Si el valor no existe, Python lanza un error `ValueError`. Por eso, antes de usar `remove()`, muchos programadores verifican con `in` (lo veremos más abajo).

pop(): quitar por posición (y quedarse con el valor)

`pop(índice)` quita el elemento de esa posición **y te lo devuelve**, para que lo guardes en una variable si quieres. Si no le das índice, quita el **último**.

```
tareas = ["barrer", "trapear", "sacudir"]
hecha = tareas.pop(0)    # quito la primera y la guardo
print(hecha)             # barrer
print(tareas)            # ['trapear', 'sacudir']

ultima = tareas.pop()    # sin índice: quita la última
print(ultima)            # sacudir
print(tareas)            # ['trapear']
```

Caso de uso: procesar pendientes uno por uno ("saco la primera tarea, la hago, sigo con la siguiente"), o usar la lista como pila (el último que entra es el primero que sale, como una torre de platos: `append()` pone el plato arriba, `pop()` lo quita de arriba).

****No confundas `remove()` con `pop()`**** `remove()` busca por **valor** y no devuelve nada; `pop()` quita por **índice** y **devuelve** el valor quitado. Este es uno de los errores más comunes, así que grábatelo: **remove = por valor, pop = por posición y te regresa el dato**.

clear(): vaciar toda la lista

`clear()` borra **todos** los elementos y deja la lista vacía.

```
carrito = ["leche", "pan", "queso"]
carrito.clear()
print(carrito)    # []
```

Caso de uso: reiniciar una lista para reusarla, como vaciar el carrito del súper para empezar una compra nueva.

index(): encontrar la posición de un valor

`index(valor)` te dice **en qué índice está** la primera aparición del valor.

```
frutas = ["manzana", "plátano", "naranja", "plátano"]
print(frutas.index("naranja"))    # 2
print(frutas.index("plátano"))    # 1 (el primero que encuentra)
```

Caso de uso: saber dónde está algo para después modificarlo o quitarlo por posición. Si el valor no existe, lanza `ValueError`.

count(): contar cuántas veces aparece un valor

`count(valor)` cuenta las apariciones del valor en la lista.

```
votos = ["Ana", "Luis", "Ana", "Pedro", "Ana"]
print(votos.count("Ana"))    # 3
print(votos.count("Luis"))   # 1
print(votos.count("Hugo"))   # 0 (no falla, devuelve 0)
```

Caso de uso: votaciones, encuestas, contar cuántas veces salió un número en un dado, estadísticas rápidas. A diferencia de `index()` y `remove()`, `count()` **no** lanza error si el valor no existe: devuelve 0.

sort(): ordenar la lista (¡en su lugar!)

`sort()` ordena los elementos **de menor a mayor** (números) o **alfabéticamente** (cadenas), modificando la lista original.

```

numeros = [5, 2, 8, 1, 9]
numeros.sort()
print(numeros)    # [1, 2, 5, 8, 9]

nombres = ["Pedro", "Ana", "Luis"]
nombres.sort()
print(nombres)    # ['Ana', 'Luis', 'Pedro']

```

Parámetros útiles:

```

numeros = [5, 2, 8, 1, 9]
numeros.sort(reverse=True)    # de mayor a menor
print(numeros)                # [9, 8, 5, 2, 1]

```

¡TRAMPA FAMOSA! `sort()` **no devuelve la lista ordenada**, devuelve `None`. Ordena la lista *en su lugar* (modifica la original). Esto atora a todo principiante:

```

numeros = [5, 2, 8, 1, 9]
resultado = numeros.sort()    # ¡MAL! resultado es None
print(resultado)              # None
print(numeros)                # [1, 2, 5, 8, 9] (la original SÍ se ordenó)

```

Si necesitas la lista ordenada en una variable nueva sin tocar la original, usa `sorted()` (lo vemos abajo).

reverse(): invertir el orden

`reverse()` invierte la lista **en su lugar**: el primero queda al final y viceversa. Ojo: **no ordena**, solo invierte el orden actual.

```

numeros = [1, 2, 3, 4, 5]
numeros.reverse()
print(numeros)    # [5, 4, 3, 2, 1]

# Ojo: si la lista NO estaba ordenada, reverse NO la ordena:
desorden = [3, 1, 4, 1, 5]
desorden.reverse()
print(desorden)    # [5, 1, 4, 1, 3] (invertida, no ordenada)

```

Igual que `sort()`, `reverse()` devuelve `None` y modifica la original.

extend(): agregar muchos elementos de un jalón

extend(otra_lista) agrega **todos** los elementos de otra lista (o cualquier colección), uno por uno, al final.

```
frutas = ["manzana", "plátano"]
mas_frutas = ["naranja", "uva"]
frutas.extend(mas_frutas)
print(frutas)    # ['manzana', 'plátano', 'naranja', 'uva']
```

****Diferencia clave con append():**** append() agrega **un** elemento (aunque ese elemento sea una lista completa, la mete como un solo bulto); extend() **desempaca** y agrega elemento por elemento.

```
a = [1, 2, 3]
a.append([4, 5])    # mete la lista [4, 5] como UN solo elemento
print(a)           # [1, 2, 3, [4, 5]] <- ¡lista dentro de lista!

b = [1, 2, 3]
b.extend([4, 5])    # desempaca y agrega 4 y 5 por separado
print(b)           # [1, 2, 3, 4, 5] <- lista plana
```

Este es otro error clásico: usar append() cuando querías extend() y terminar con listas anidadas sin querer. Regla: *¿quieres agregar un solo dato? append(). ¿Quieres pegar todos los elementos de otra lista? extend().*

copy(): copiar de verdad una lista

copy() crea una **copia independiente** de la lista.

```
original = [1, 2, 3]
copia = original.copy()
copia.append(4)
print(original)    # [1, 2, 3]      (no se afectó)
print(copia)       # [1, 2, 3, 4]
```

¿Por qué existe? Por la **trampa del alias** (la vemos en detalle en errores comunes): si haces copia = original **sin**.copy(), no estás copiando nada; estás creando un segundo nombre para la **misma** lista. Cambias una y cambia la otra. copy() evita ese susto.

Funciones que trabajan con listas: len(), min(), max(), sum()

Además de los métodos, hay funciones integradas que reciben la lista como argumento.

```
califs = [8, 9, 7, 10, 6]

print(len(califs))    # 5    (cuántos elementos hay)
print(min(califs))    # 6    (el menor)
print(max(califs))    # 10   (el mayor)
print(sum(califs))    # 40   (la suma de todos)
```

Con esto, el promedio de un grupo es una línea:

```
califs = [8, 9, 7, 10, 6]
promedio = sum(califs) / len(califs)
print(promedio)    # 8.0
```

len() funciona con cualquier lista (y con cadenas, tuplas, etc.). min(), max() y sum() necesitan que los elementos sean números (o comparables entre sí). Si mezclas números con cadenas, min() lanza TypeError porque no puede comparar peras con manzanas.

El operador in: preguntar si algo está en la lista

in te dice si un valor **existe** dentro de la lista. Devuelve True o False.

```
compras = ["leche", "huevos", "tortillas"]

print("leche" in compras)    # True
print("queso" in compras)    # False
print("queso" not in compras) # True
```

Caso de uso: verificar antes de actuar, para evitar errores:

```
compras = ["leche", "huevos", "tortillas"]

if "queso" in compras:
    compras.remove("queso")
else:
    print("El queso no está en la lista, no hay nada que quitar")
```

También sirve para filtrar:

```

invitados = ["Ana", "Luis", "María", "Pedro"]
nombre = input("¿Cómo te llamas? ")

if nombre in invitados:
    print("¡Bienvenido a la fiesta!")
else:
    print("Lo siento, no estás en la lista")

```

Desempaquetado básico: repartir la lista en variables

Si sabes cuántos elementos tiene una lista, puedes **desempaquetarla** en variables de un jalón:

```

datos = ["Ana", 25, "CDMX"]
nombre, edad, ciudad = datos

print(nombre)    # Ana
print(edad)      # 25
print(ciudad)    # CDMX

```

Python asigna en orden: el primer elemento a la primera variable, el segundo a la segunda, etc. El número de variables **debe coincidir** con el número de elementos, o lanza `ValueError`.

```

# Con asterisco te quedas con "el resto":
numeros = [1, 2, 3, 4, 5]
primero, *resto = numeros
print(primero)    # 1
print(resto)      # [2, 3, 4, 5]

```

El `*resto` junta todos los sobrantes en una lista nueva. Muy útil cuando te interesa el primer dato y "todo lo demás".

Tuplas: las primas inmutables de las listas

Una **tupla** es una colección ordenada como la lista, pero con dos diferencias: se escribe entre **paréntesis** `()` en vez de corchetes, y es **inmutable** (no se puede cambiar después de creada).

```
coordenadas = (19.4326, -99.1332) # latitud y longitud de CDMX
print(coordenadas[0]) # 19.4326
print(coordenadas[1]) # -99.1332
```

Puedes leer elementos por índice, hacer slicing, usar `len()`, `in`, recorrerla con `for...` todo igual que una lista. Lo que **NO** puedes hacer es modificarla:

```
coordenadas = (19.4326, -99.1332)
coordenadas[0] = 20.0 # TypeError: 'tuple' object does not support item
assignment
```

Tampoco tiene `append()`, `remove()`, `sort()` ni ningún método que modifique. Si lo intentas, Python te marca error. Y eso, aunque suene como limitación, es su **superpoder**.

Por qué la inmutabilidad es una ventaja

Hay datos que **no deben cambiar por accidente** en tu programa. Ejemplos:

- **Coordenadas geográficas:** (19.4326, -99.1332). Si algo en tu código las modificara por error, tu mapa apuntaría a otro lado del mundo.
- **Colores RGB:** (255, 0, 0) es rojo puro. No quieres que a medio programa se convierta en otra cosa.
- **Días de la semana:** ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo"). Los días no cambian; si usas una lista, alguien podría hacer `dias.append("lunes2")` por error y romper todo.
- **Datos de configuración:** límites, constantes, credenciales fijas.

La tupla es como un **contrato**: "estos valores quedan así, punto". Si alguna parte del programa intenta cambiarlos, Python truena de inmediato y te avisa del bug, en vez de dejar que el dato corrupto se propague en silencio y te dé resultados malos a la mitad de la noche. En programación, un error que grita es mil veces mejor que un error silencioso.

Además, las tuplas son un poco más **rápidas** y ocupan menos memoria que las listas, y se pueden usar como **claves de diccionarios** (las listas no pueden, pero eso lo verás en la unidad de diccionarios).

La tupla de un elemento: la famosa coma

Aquí hay una trampa legendaria. ¿Cómo creas una tupla con **un solo** elemento?

```
# ¡CUIDADO!
t1 = (5)
print(type(t1))    # <class 'int'> <- ¡NO es tupla! Es un entero entre paréntesis

t2 = (5,)
print(type(t2))    # <class 'tuple'> <- ESTA sí es tupla
```

Sin la coma, Python cree que los paréntesis son solo agrupación matemática, como en $(5 + 3) * 2$. La **coma** es lo que realmente crea la tupla, no los paréntesis. De hecho:

```
t3 = 5,
print(type(t3))    # <class 'tuple'> (funciona hasta sin paréntesis)

t4 = 1, 2, 3
print(t4)          # (1, 2, 3) (tupla creada solo con comas)
```

Regla de oro: **tupla de un elemento siempre lleva coma al final**: $(5,)$. Sin coma no hay tupla. Este detalle sale en exámenes y entrevistas; ya lo sabes.

Cuándo usar lista y cuándo tupla

| Pregunta | Respuesta | |---|---| | ¿Los datos van a cambiar (agregar, quitar, modificar)? | **Lista** `[]` | | ¿Los datos son fijos y no deben cambiar? | **Tupla** `()` | | ¿Necesitas métodos como `append()`, `sort()`? | **Lista** | | ¿Son coordenadas, RGB, días, constantes? | **Tupla** | | ¿No estás seguro? | **Lista** (es la opción por defecto) |

En la práctica usarás listas el 90% del tiempo. Las tuplas aparecen para datos fijos, para devolver múltiples valores de una función (lo verás en la unidad de funciones) y para desempaqueado.

Errores comunes (y cómo evitarlos)

1. IndexError: índice fuera de rango

El error más famoso de las listas. Pasa cuando pides un índice que no existe.

```
frutas = ["manzana", "plátano", "naranja"]    # índices válidos: 0, 1, 2
print(frutas[3])    # IndexError: list index out of range
```

La lista tiene 3 elementos, así que el último índice válido es el **2** (recuerda: se empieza en 0). El error clásico es usar `len(lista)` como índice:

```
frutas = ["manzana", "plátano", "naranja"]
ultimo = frutas[len(frutas)] # ¡MAL! len es 3, pero el último índice es 2
```

Soluciones: usa `frutas[-1]` para el último, o `frutas[len(frutas) - 1]`. Y si recorres con `for`, usa `range(len(lista))` o mejor aún, itera directo sobre la lista.

2. Confundir `remove()` con `pop()`

Ya lo vimos, pero vale repetirlo porque atora a todos:

- `remove("valor")` -> busca por **valor**, no devuelve nada, lanza `ValueError` si no lo encuentra.
- `pop(índice)` -> quita por **índice**, **devuelve** el valor quitado, lanza `IndexError` si el índice no existe.

```
letras = ["a", "b", "c"]
letras.remove("b") # quita la "b" (por valor) -> ['a', 'c']
x = letras.pop(0) # quita el índice 0 (la "a") y x = "a" -> ['c']
```

3. `sort()` devuelve `None`

```
nums = [3, 1, 2]
ordenada = nums.sort() # ¡MAL!
print(ordenada) # None (la lista original SÍ quedó ordenada)
```

Si quieres una **nueva** lista ordenada sin modificar la original, usa `sorted()`:

```
nums = [3, 1, 2]
ordenada = sorted(nums) # BIEN: devuelve [1, 2, 3]
print(ordenada) # [1, 2, 3]
print(nums) # [3, 1, 2] (la original intacta)
```

Resumen: `lista.sort()` ordena *en su lugar* y devuelve `None`. `sorted(lista)` devuelve una *nueva* lista ordenada y no toca la original.

4. El alias vs la copia: dos nombres, una sola lista

Esta es la trampa más traicionera de la unidad. Mira con atención:

```
a = [1, 2, 3]
b = a          # ¡OJO! Esto NO copia la lista
b.append(4)
print(a)       # [1, 2, 3, 4]  <- ¡"a" también cambió!
print(b)       # [1, 2, 3, 4]
```

`b = a` no crea una lista nueva; crea un **segundo nombre** (alias) para la **misma** lista en memoria. Es como si tu casa tuviera dos direcciones postales distintas: el cartero llega al mismo lugar por cualquiera de las dos. Cambias algo por `b` y lo ves reflejado en `a`, porque son la misma lista.

Soluciones para copiar de verdad:

```
a = [1, 2, 3]

b = a.copy()    # opción 1: método copy()
c = a[:]        # opción 2: slicing completo (truco clásico)
d = list(a)     # opción 3: constructor list()

b.append(4)
print(a)        # [1, 2, 3]      (intacta)
print(b)        # [1, 2, 3, 4]
```

Las tres opciones crean una lista **nueva e independiente**. Úsalas siempre que necesites una copia que puedas modificar sin afectar la original.

5. Modificar una lista mientras la recorres

```
# ¡PELIGROSO!
numeros = [1, 2, 3, 4, 5]
for n in numeros:
    if n % 2 == 0:
        numeros.remove(n)    # quitar mientras recorres salta elementos
print(numeros)              # [1, 3, 5]... o algo raro e impredecible
```

Cuando quitas elementos mientras recorres, los índices se recorren y el **for se salta elementos** sin avisar. **Solución:** recorre una **copia** y modifica la original, o construye una lista nueva:

```
# BIEN: construir una lista nueva con lo que SÍ quieres
numeros = [1, 2, 3, 4, 5]
impares = [n for n in numeros if n % 2 != 0]
print(impares)    # [1, 3, 5]
```

(Esa sintaxis de corchetes con **for** adentro se llama **list comprehension**; la verás a fondo más adelante. Por ahora quédate con la idea: no modifiques la lista que estás recorriendo.)

Buenas prácticas con listas y tuplas

1. **Nombres en plural.** Una lista guarda muchos elementos, así que nómbrala en plural: compras, nombres, calificaciones. Cuando veas `for compra in compras:` se lee solo.
2. **No mezcles tipos sin razón.** Python lo permite (`["hola", 42, True]`), pero una lista de un solo tipo es más fácil de ordenar, sumar y entender. Mezcla solo cuando tengas un buen motivo (como `[nombre, edad, ciudad]` que luego desempaquetas).
3. **Usa tuplas para datos fijos.** Coordenadas, colores, días de la semana, configuraciones: si no debe cambiar, tupla. Es documentación viva: quien lea tu código entiende "esto es fijo".
4. ****Verifica con `in` antes de `remove()` o `index()`**** Evitas el `ValueError` con una línea.
5. **Copia antes de modificar si la original debe sobrevivir.** `nueva = original.copy()` o `nueva = sorted(original)` en vez de `original.sort()`.
6. ****Prefiere `enumerate()` sobre `range(len(...))`**** Es más legible y más "pitónico".
7. **No modifiques la lista que estás recorriendo.** Construye una nueva o recorre una copia.

Convertir entre listas, tuplas y cadenas: `split()` y `join()`

Las listas no viven solas: constantemente vas a convertir cadenas en listas y listas en cadenas. Para eso existen dos métodos que trabajan en equipo: `split()` y `join()`.

**** `split()`: de cadena a lista.**** Parte una cadena en pedazos usando un separador.


```

frase = "la casa es roja"
palabras = frase.split()          # separa por espacios
print(palabras)                   # ['la', 'casa', 'es', 'roja']

fecha = "2026-09-19"
partes = fecha.split("-")         # separa por guiones
print(partes)                     # ['2026', '09', '19']

csv = "Ana,Luis,María,Pedro"
nombres = csv.split(",")          # separa por comas (formato CSV)
print(nombres)                    # ['Ana', 'Luis', 'María', 'Pedro']

```

Caso de uso: leer datos del usuario o de un archivo. Si el usuario escribe sus calificaciones separadas por comas (8,9,7,10), un `split(",")` te da la lista y luego conviertes cada pedazo a número.

```

entrada = input("Escribe tus calificaciones separadas por comas: ") # ej: 8,9,7,10
textos = entrada.split(",")
califs = [float(t) for t in textos] # convertimos cada texto a decimal
print("Promedio:", sum(califs) / len(califs))

```

**** join():** de lista a cadena. ****** Es lo contrario: pega los elementos de una lista en una sola cadena, usando la cadena como "pegamento".

```

palabras = ["la", "casa", "es", "roja"]
frase = " ".join(palabras)        # pega con espacios
print(frase)                      # la casa es roja

nombres = ["Ana", "Luis", "María"]
lista_csv = ",".join(nombres)     # pega con comas
print(lista_csv)                  # Ana,Luis,María

```

Ojo con la sintaxis: el pegamento va **antes** del punto (`" ".join(...)`, no `palabras.join(" ")`). Se lee como: *"une estas palabras usando un espacio como pegamento"*. Los elementos deben ser cadenas; si hay números, conviértelos primero con `str()`.

Conversiones entre lista y tupla:

```
lista = [1, 2, 3]
tupla = tuple(lista)    # lista -> tupla: (1, 2, 3)

tupla2 = (4, 5, 6)
lista2 = list(tupla2)   # tupla -> lista: [4, 5, 6]
```

¿Para qué querrías convertir? Ejemplo típico: tienes una tupla de datos fijos pero necesitas ordenarlos (y `sort()` no existe en tuplas). La conviertes a lista, la ordenas, y si quieres la regresas a tupla:

```
puntos = (5, 2, 8, 1)
ordenados = tuple(sorted(puntos))
print(ordenados)    # (1, 2, 5, 8)
```

Ordenar a tu manera: `sort()` con `key`

`sort()` ordena números de menor a mayor y cadenas alfabéticamente. Pero ¿qué pasa si quieres ordenar por otro criterio? Por ejemplo, ordenar nombres **por longitud** (del más corto al más largo). Para eso está el parámetro `key`: le pasas una función que dice "ordena según ESTO".

```
nombres = ["Ana", "Guadalupe", "Luis", "María"]

nombres.sort(key=len)    # ordena según la longitud de cada nombre
print(nombres)           # ['Ana', 'Luis', 'María', 'Guadalupe']
```

`key=len` significa: "para ordenar, mide cada elemento con `len()` y ordena por ese número". También puedes ordenar una lista de listas por alguna columna:

```
# Alumnos: [nombre, promedio]. Ordenar por promedio (columna 1), de mayor a menor
alumnos = [["Ana", 8.5], ["Luis", 9.2], ["María", 7.8]]

alumnos.sort(key=lambda a: a[1], reverse=True)
print(alumnos)    # [['Luis', 9.2], ['Ana', 8.5], ['María', 7.8]]
```

Ese `lambda a: a[1]` es una mini-función anónima que dice "dame el elemento 1 de cada sublista". No te preocupes si `lambda` se ve raro: lo verás a fondo en la unidad de funciones. Por ahora quédate con la idea de que `key` te deja ordenar por el criterio que tú quieras, y eso es utilísimo en programas reales (ordenar productos por precio, jugadores por puntaje, tareas por prioridad).

Traza de código: piensa como la computadora

Una habilidad que separa a los programadores que avanzan de los que se atorán es la **traza**: ejecutar el código mentalmente, línea por línea, como si fueras la computadora. Practiquemos con tres mini-retos. Tapa la respuesta, traza en tu cabeza o en papel, y luego verifica.

Reto 1:

```
a = [10, 20, 30]
b = a
b[1] = 99
print(a)
```

Traza: `a` apunta a la lista `[10, 20, 30]`. `b = a` NO copia: `b` es un alias de la misma lista. `b[1] = 99` cambia el segundo elemento de ESA lista. Al imprimir `a` ves el cambio.

Respuesta: `[10, 99, 30]`

Reto 2:

```
nums = [3, 1, 2]
x = sorted(nums)
y = nums.sort()
print(x)
print(y)
print(nums)
```

Traza: `sorted(nums)` devuelve una NUEVA lista ordenada -> `x = [1, 2, 3]`, `nums` intacta. `nums.sort()` ordena `nums` en su lugar y devuelve `None` -> `y = None`, `nums` ahora es `[1, 2, 3]`.

Respuesta:

```
[1, 2, 3]
None
[1, 2, 3]
```

Reto 3:

```
datos = [5, 10, 15, 20, 25]
resultado = datos[1:4]
resultado.append(99)
print(datos)
print(resultado)
```

Traza: `datos[1:4]` crea una lista NUEVA con los índices 1, 2, 3 -> `[10, 15, 20]`. El slicing siempre devuelve copia, así que `resultado` es independiente. `append(99)` solo afecta a `resultado`. `datos` queda intacta.

Respuesta:

```
[5, 10, 15, 20, 25]
[10, 15, 20, 99]
```

Si trazaste los tres bien, ya piensas como programador. Si fallaste alguno, no te preocupes: vuelve a leer la sección del tema (alias, `sort()` vs `sorted()`, slicing) y reinténtalo. La traza es un músculo: se entrena.

Preguntas que siempre hacen (y sus respuestas)

"¿Por qué los índices empiezan en 0 y no en 1?" Es la pregunta más vieja de la programación. La respuesta corta: el índice mide el *desplazamiento* desde el inicio, y el primer elemento está a 0 casillas de distancia. La respuesta práctica: casi todos los lenguajes lo hacen así, así que es una costumbre que te sirve en todos lados. Después de dos semanas programando, el 0 se siente natural y el 1 se siente raro.

"¿Cuándo uso corchetes, paréntesis o llaves?"

- Corchetes `[]` -> **lista** (mutable, ordenada).
- Paréntesis `()` -> **tupla** (inmutable, ordenada).
- Llaves `{}` -> **diccionario o set** (los verás en las próximas unidades).

Un truco para no confundirlos: los corchetes son "abiertos", como una canasta donde metes y sacas cosas (lista mutable). Los paréntesis son "cerrados", como un paquete sellado que ya no se abre (tupla inmutable).

"¿append() o +=?" Ambos agregan al final, pero hay una diferencia sutil:

```
a = [1, 2]
a.append(3)      # agrega UN elemento: [1, 2, 3]
a += [4, 5]      # agrega VARIOS: [1, 2, 3, 4, 5] (equivale a extend)
```

`append()` agrega un solo bulto; `+=` con una lista agrega elemento por elemento (como `extend()`). Ojo: `a += 6` truena, porque `+=` espera una colección, no un valor suelto.

****"¿Por qué `sort()` no funciona con listas mezcladas?"****

```
mezcla = [3, "hola", 1]
mezcla.sort()  # TypeError: '<' not supported between 'str' and 'int'
```

Porque Python no sabe si "hola" va antes o después del 3. ¿Cómo comparas una palabra con un número? No se puede, y Python prefiere tronar con un error claro antes que inventarse un orden sin sentido. Moraleja: ordena listas de un solo tipo.

"¿Las tuplas de verdad son más rápidas?" Sí, un poco: como son inmutables, Python las guarda de forma más compacta y accede a ellas ligeramente más rápido. En programas pequeños la diferencia es invisible (microsegundos), así que no elijas tupla por velocidad: elígela por *significado* (datos fijos). La velocidad es un bonus, no la razón.

"¿Puedo tener una tupla dentro de una lista y viceversa?" Claro. Ya viste listas de tuplas (las ciudades) y puedes tener tuplas de listas:

```
mixta = ([1, 2], [3, 4])  # tupla que contiene listas
mixta[0].append(99)       # ¡ESTO SÍ funciona!
print(mixta)              # ([1, 2, 99], [3, 4])
```

La tupla es inmutable (no puedes cambiar *qué* listas contiene), pero las listas de adentro siguen siendo mutables (puedes cambiar su *contenido*). Es como una caja sellada con cajones que sí se abren: no puedes cambiar los cajones, pero sí lo que hay dentro de cada uno. Este detalle es avanzado; con saber que existe basta por ahora.

"¿Cómo sé cuánta memoria usa mi lista?" Con `sys.getsizeof()`, aunque es un dato curioso más que una necesidad diaria:

```
import sys
print(sys.getsizeof([1, 2, 3]))      # tamaño en bytes de la lista
print(sys.getsizeof((1, 2, 3)))      # la tupla equivalente usa menos
```

Copia superficial vs copia profunda (el detalle que muerde)

Ya sabes que `b = a` crea un alias y que `b = a.copy()` crea una copia independiente. Pero hay un nivel más de profundidad: ¿qué pasa si la lista contiene **otras listas**?

```
original = [[1, 2], [3, 4]]
copia = original.copy()    # copia SUPERFICIAL

copia[0].append(99)
print(original)            # [[1, 2, 99], [3, 4]]  <- ¡la original TAMBIÉN cambió!
print(copia)               # [[1, 2, 99], [3, 4]]
```

¿Qué pasó? `copy()` duplica la lista **exterior**, pero las listas **interiores** siguen siendo las mismas (compartidas). Es como fotocopiar la portada de una carpeta pero dejar los documentos originales adentro: cambias un documento y se ve reflejado en ambas carpetas. A esto se le llama **copia superficial** (shallow copy).

Para copiar **todo**, hasta el fondo, necesitas una **copia profunda** (deep copy) con el módulo `copy`:

```
import copy

original = [[1, 2], [3, 4]]
profunda = copy.deepcopy(original)    # copia TODO, nivel por nivel

profunda[0].append(99)
print(original)            # [[1, 2], [3, 4]]      <- intacta
print(profunda)            # [[1, 2, 99], [3, 4]]
```

`deepcopy()` clona la lista exterior **y** cada lista interior, así que nada se comparte. Regla práctica:

- Lista plana (solo números, cadenas) -> `.copy()` o `[:]` es suficiente.
- Lista con listas adentro (matrices, agenda de contactos) -> usa `copy.deepcopy()` si vas a modificar las interiores.

No es un tema que necesites todos los días, pero cuando te muerda (y te va a morder alguna vez con una matriz o una agenda), ya sabes su nombre y su antídoto.

Ejemplos paso a paso

Ejemplo 1: Lista del súper interactiva

Vamos a armar el programa clásico: una lista de compras que el usuario va llenando hasta que escribe "fin".

```
# 1. Empezamos con una lista vacía (la canasta vacía del súper)
compras = []

# 2. Ciclo infinito: pedimos productos hasta que el usuario diga "fin"
while True:
    producto = input("Agrega un producto (o escribe 'fin' para terminar): ")

    # 3. Si escribió "fin", rompemos el ciclo
    if producto.lower() == "fin":
        break

    # 4. Agregamos el producto al final de la lista
    compras.append(producto)
    print(f"'{producto}' agregado. Llevas {len(compras)} productos.")

# 5. Mostramos la lista final con números
print("\n--- Tu lista del súper ---")
for i, producto in enumerate(compras, start=1):
    print(f"{i}. {producto}")

print(f"\nTotal de productos: {len(compras)}")
```

Explicación línea por línea:

- `compras = []` - creamos la lista vacía. Es la canasta antes de entrar al súper.
- `while True:` - ciclo infinito que solo se detiene con `break`. Lo usamos porque no sabemos cuántos productos va a agregar el usuario.
- `producto = input(...)` - pedimos un producto por teclado.

- `producto.lower() == "fin"` - comparamos en minúsculas para que "FIN", "Fin" o "fin" funcionen igual.
- `break` - rompe el ciclo y salta a la línea después del `while`.
- `compras.append(producto)` - el corazón del programa: agrega el producto al final.
- `enumerate(compras, start=1)` - recorre con índice empezando en 1 (más natural para una lista de súper que el 0 de Python).

Si el usuario escribe: leche, huevos, tortillas, fin, la salida es:

```
--- Tu lista del súper ---  
1. leche  
2. huevos  
3. tortillas  
  
Total de productos: 3
```

Ejemplo 2: Calificaciones con promedio, máximo y mínimo

El programa que todo maestro quisiera tener: captura calificaciones y calcula estadísticas.


```

# 1. Lista de calificaciones del grupo
califs = [8.5, 9.0, 7.5, 10.0, 6.0, 8.0, 9.5, 7.0]

# 2. Estadísticas básicas con funciones integradas
total = len(califs)
promedio = sum(califs) / total
maxima = max(califs)
minima = min(califs)

# 3. Mostramos resultados con formato de 2 decimales
print(f"Número de alumnos: {total}")
print(f"Promedio del grupo: {promedio:.2f}")
print(f"Calificación más alta: {maxima}")
print(f"Calificación más baja: {minima}")

# 4. Contamos cuántos aprobaron (calificación >= 6)
aprobados = 0
for cal in califs:
    if cal >= 6:
        aprobados += 1

print(f"Aprobaron: {aprobados} de {total}")

# 5. Lista de los que sacaron 9 o más (los de excelencia)
excelencia = []
for cal in califs:
    if cal >= 9:
        excelencia.append(cal)

print(f"Calificaciones de excelencia: {excelencia}")

```

Explicación línea por línea:

- `sum(califs) / len(califs)` - la fórmula del promedio en una línea: suma total entre cantidad de elementos.
- `{promedio:.2f}` - formato f-string que muestra solo 2 decimales (8.06 en vez de 8.0625).
- El `for` con `if` cuenta aprobados: recorre cada calificación y suma 1 al contador si es mayor o igual a 6.
- `excelencia = [] + append()` - el patrón clásico: lista vacía que se llena filtrando. Este patrón aparece en el 80% de los programas que procesan listas.

Salida:

```
Número de alumnos: 8
Promedio del grupo: 8.19
Calificación más alta: 10.0
Calificación más baja: 6.0
Aprobaron: 8 de 8
Calificaciones de excelencia: [9.0, 10.0, 9.5]
```

Ejemplo 3: Slicing para obtener mitades y porciones

El slicing brilla cuando necesitas partir una lista en pedazos: mitades, tercios, los primeros N, los últimos N.

```
# 1. Lista de 10 participantes en orden de llegada
participantes = ["Ana", "Luis", "María", "Pedro", "Sofía",
                 "Diego", "Lucía", "Miguel", "Elena", "Jorge"]

# 2. Los dividimos en dos equipos con slicing
mitad = len(participantes) // 2
equipo_a = participantes[:mitad]
equipo_b = participantes[mitad:]

print("Equipo A:", equipo_a)
print("Equipo B:", equipo_b)

# 3. Los 3 primeros lugares (podio)
podio = participantes[:3]
print("Podio:", podio)

# 4. Los últimos 2 (los más lentos, ¡a entrenar!)
ultimos = participantes[-2:]
print("Últimos lugares:", ultimos)

# 5. Uno sí y uno no (para sortear turnos alternados)
turno_par = participantes[::2]
turno_impar = participantes[1::2]
print("Turno par:", turno_par)
print("Turno impar:", turno_impar)

# 6. Lista invertida (del último al primero)
al_reves = participantes[::-1]
print("Orden inverso:", al_reves)
```

Explicación línea por línea:

- `len(participantes) // 2` - división entera para obtener la mitad exacta (5). El `//` descarta decimales.
- `participantes[:mitad]` - del inicio hasta el índice 5 (sin incluirlo): los primeros 5.
- `participantes[mitad:]` - del índice 5 hasta el final: los últimos 5.
- `participantes[-2:]` - los últimos dos, sin importar cuántos haya en total.
- `[::2]` - paso de 2: toma uno, salta uno. `[1::2]` empieza en el índice 1.
- `[::-1]` - el truco para invertir: paso negativo recorre hacia atrás.

Salida:

```
Equipo A: ['Ana', 'Luis', 'María', 'Pedro', 'Sofía']
Equipo B: ['Diego', 'Lucía', 'Miguel', 'Elena', 'Jorge']
Podio: ['Ana', 'Luis', 'María']
Últimos lugares: ['Elena', 'Jorge']
Turno par: ['Ana', 'María', 'Sofía', 'Lucía', 'Elena']
Turno impar: ['Luis', 'Pedro', 'Diego', 'Miguel', 'Jorge']
Orden inverso: ['Jorge', 'Elena', 'Miguel', 'Lucía', 'Diego', 'Sofía', 'Pedro', 'María', 'Luis', 'Ana']
```

Ejemplo 4: Agenda simple con listas

Una agenda de contactos donde cada contacto es una lista `[nombre, teléfono]`, y la agenda es una lista de contactos (lista dentro de lista).

```

# 1. La agenda: lista de contactos, cada contacto es [nombre, teléfono]
agenda = [
    ["Ana", "555-1234"],
    ["Luis", "555-5678"],
    ["María", "555-9012"],
]

# 2. Función para agregar un contacto
def agregar(nombre, telefono):
    agenda.append([nombre, telefono])
    print(f"{nombre} agregado a la agenda.")

# 3. Función para buscar un contacto por nombre
def buscar(nombre):
    for contacto in agenda:
        if contacto[0].lower() == nombre.lower():
            return contacto[1] # devolvemos el teléfono
    return None # no lo encontramos

# 4. Función para mostrar toda la agenda
def mostrar():
    print("\n--- Agenda ---")
    for i, contacto in enumerate(agenda, start=1):
        nombre, telefono = contacto # desempaquetado
        print(f"{i}. {nombre}: {telefono}")

# 5. Probamos las funciones
agregar("Pedro", "555-3456")

tel = buscar("maría") # funciona aunque escribas con minúsculas
if tel:
    print(f"El teléfono de María es {tel}")
else:
    print("Contacto no encontrado")

tel = buscar("Hugo")
if tel:
    print(f"El teléfono de Hugo es {tel}")
else:
    print("Contacto no encontrado")

mostrar()

```

Explicación línea por línea:

- Cada contacto es una **lista de 2 elementos**: [nombre, teléfono]. La agenda es una lista de esas listas.
- `agenda.append([nombre, telefono])` - agregamos una lista completa como un solo elemento (aquí sí queremos `append`, no `extend`).
- En `buscar`, comparamos `contacto[0]` (el nombre) en minúsculas para que la búsqueda no distinga mayúsculas.
- `return contacto[1]` - si lo encuentra, devuelve el teléfono y la función termina ahí.
- `return None` - si el `for` termina sin encontrar nada, devolvemos `None` (nada).
- `nombre, telefono = contacto` - **desempaquetado**: la lista de 2 elementos se reparte en 2 variables de un jalón.

Salida:

```
Pedro agregado a la agenda.  
El teléfono de María es 555-9012  
Contacto no encontrado
```

```
--- Agenda ---  
1. Ana: 555-1234  
2. Luis: 555-5678  
3. María: 555-9012  
4. Pedro: 555-3456
```

Ejemplo 5: Coordenadas con tuplas (datos que no deben cambiar)

Un programa que calcula la distancia entre dos puntos usando tuplas para las coordenadas. Las tuplas garantizan que ningún punto se modifique por accidente a media operación.

```

import math

# 1. Dos puntos en el mapa, como tuplas (x, y)
# Son tuplas porque un punto no debe cambiar a medio cálculo
punto_a = (0, 0)
punto_b = (3, 4)

# 2. Desempaquetamos cada tupla en sus coordenadas
x1, y1 = punto_a
x2, y2 = punto_b

# 3. Fórmula de distancia euclidiana
distancia = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
print(f"Distancia entre {punto_a} y {punto_b}: {distancia}")

# 4. Varias ciudades como tuplas (nombre, latitud, longitud)
ciudades = [
    ("CDMX", 19.4326, -99.1332),
    ("Guadalajara", 20.6597, -103.3496),
    ("Monterrey", 25.6866, -100.3161),
]

# 5. Recorremos e imprimimos con desempaqueado en el for
print("\n--- Ciudades ---")
for nombre, lat, lon in ciudades:
    print(f"{nombre}: lat {lat}, lon {lon}")

# 6. Intentar modificar una tupla truena (y ESO es bueno)
try:
    punto_a[0] = 10
except TypeError as e:
    print(f"\nNo se puede modificar una tupla: {e}")

```

Explicación línea por línea:

- punto_a = (0, 0) - tupla de 2 elementos: la coordenada x y la y.
- x1, y1 = punto_a - desempaqueado directo de la tupla en variables.
- math.sqrt(...) - raíz cuadrada para la fórmula de distancia. Entre (0,0) y (3,4) la distancia es 5 (el famoso triángulo 3-4-5).
- ciudades - lista de tuplas: cada ciudad es una tupla fija de (nombre, latitud, longitud). La lista puede crecer (más ciudades), pero cada ciudad individual no cambia.

- for nombre, lat, lon in ciudades: - desempaquetado **dentro del for**: cada tupla de 3 elementos se reparte en 3 variables automáticamente. Elegante y legible.
- El try/except demuestra que modificar una tupla lanza TypeError. En un programa real, ese error te avisa de inmediato que algo anda mal, en vez de corromper tus datos en silencio.

Salida:

```
Distancia entre (0, 0) y (3, 4): 5.0

--- Ciudades ---
CDMX: lat 19.4326, lon -99.1332
Guadalajara: lat 20.6597, lon -103.3496
Monterrey: lat 25.6866, lon -100.3161

No se puede modificar una tupla: 'tuple' object does not support item assignment
```

Ejemplo 6: Eliminar duplicados manteniendo el orden

Problema clásico de entrevistas de trabajo: tienes una lista con repetidos y quieres dejar solo la primera aparición de cada valor, **conservando el orden original**.

```
# 1. Lista con duplicados (como saldría de un formulario con respuestas repetidas)
original = ["Ana", "Luis", "Ana", "Pedro", "Luis", "María", "Ana", "Pedro"]

# 2. Método 1: ciclo + lista nueva (mantiene el orden) [OK]
sin_duplicados = []
for nombre in original:
    if nombre not in sin_duplicados: # solo lo agrego si no está ya
        sin_duplicados.append(nombre)

print("Método 1 (orden conservado):", sin_duplicados)

# 3. Método 2: dict.fromkeys (truco pitónico, también conserva orden) [OK]
sin_duplicados2 = list(dict.fromkeys(original))
print("Método 2 (truco pitónico): ", sin_duplicados2)

# 4. Método 3: set (¡CUIDADO! pierde el orden) [!]
sin_duplicados3 = list(set(original))
print("Método 3 (set, sin orden): ", sin_duplicados3)
```

Explicación línea por línea:

- **Método 1:** recorreremos la original y solo agregamos a la nueva lista si el valor **no** está ya (`not in`). Como recorreremos en orden, la primera aparición de cada valor es la que se conserva. Simple y claro.
- **Método 2:** `dict.fromkeys(original)` crea un diccionario usando los valores como claves. Los diccionarios **no permiten claves repetidas** y (desde Python 3.7) **conservan el orden de inserción**. Luego `list(...)` lo convierte de vuelta a lista. Es el truco favorito de los programadores Python.
- **Método 3:** `set()` elimina duplicados rapidísimo, pero los **sets no tienen orden garantizado**. Si el orden te importa, no lo uses.

Salida:

```
Método 1 (orden conservado): ['Ana', 'Luis', 'Pedro', 'María']
Método 2 (truco pitónico):   ['Ana', 'Luis', 'Pedro', 'María']
Método 3 (set, sin orden):   ['María', 'Ana', 'Luis', 'Pedro'] (el orden puede
variar)
```

Moraleja: cuando el orden importa, usa el método 1 (el más didáctico) o el 2 (el más pitónico). El `set` solo cuando el orden te dé igual.

Práctica

Ejercicio 1: La lista del súper con presupuesto

Tienes 500 pesos de presupuesto. Crea una lista de productos donde cada producto es una lista `[nombre, precio]`. Agrega productos con `append()` hasta que agregar otro exceda el presupuesto. Al final muestra la lista completa y cuánto gastaste.

Solución paso a paso:


```

# 1. Presupuesto y lista vacía (cada producto será [nombre, precio])
presupuesto = 500
carrito = []

# 2. Productos disponibles en la tienda
tienda = [
    ["leche", 28],
    ["huevos", 45],
    ["tortillas", 22],
    ["queso", 85],
    ["jamón", 95],
    ["pan", 40],
    ["refresco", 35],
    ["pollo", 120],
]

# 3. Intentamos agregar cada producto si cabe en el presupuesto
for producto in tienda:
    nombre, precio = producto          # desempaquetamos
    gastado = sum(p[1] for p in carrito) # sumamos los precios actuales
    if gastado + precio <= presupuesto:
        carrito.append(producto)
        print(f"Agregado: {nombre} (${precio})")
    else:
        print(f"No cabe: {nombre} (${precio}) - se pasa del presupuesto")

# 4. Ticket final
print("\n--- Ticket ---")
total = 0
for nombre, precio in carrito:
    print(f"{nombre}: ${precio}")
    total += precio

print(f"\nTotal gastado: ${total}")
print(f"Te sobraron: ${presupuesto - total}")

```

Explicación:

- Cada producto es una lista [nombre, precio], y el carrito es una lista de esas listas.
- nombre, precio = producto desempaqueta cada producto en sus dos partes.

- `sum(p[1] for p in carrito)` suma el precio (`p[1]`, el segundo elemento) de cada producto en el carrito. Es un **generator expression**, primo hermano del list comprehension.
- Antes de cada `append()` verificamos que el nuevo total no pase de 500. Así el presupuesto se respeta solito.
- El ticket final recorre el carrito desempaquetando nombre y precio en el mismo `for`.

Ejercicio 2: Analizador de calificaciones

Pide al usuario 5 calificaciones (con `input()`), guárdalas en una lista y muestra: el promedio, la más alta, la más baja, cuántas son mayores o iguales a 8 y la lista ordenada de menor a mayor.

Solución paso a paso:

```
# 1. Lista vacía para las calificaciones
califs = []

# 2. Pedimos 5 calificaciones al usuario
for i in range(5):
    cal = float(input(f"Calificación {i + 1}: "))
    califs.append(cal)

# 3. Cálculos con funciones integradas
promedio = sum(califs) / len(califs)
print(f"\nPromedio: {promedio:.2f}")
print(f"Más alta: {max(califs)}")
print(f"Más baja: {min(califs)}")

# 4. Contamos cuántas son >= 8
altas = 0
for cal in califs:
    if cal >= 8:
        altas += 1
print(f"Calificaciones de 8 o más: {altas}")

# 5. Lista ordenada SIN modificar la original (usamos sorted, no sort)
ordenadas = sorted(califs)
print(f"De menor a mayor: {ordenadas}")
print(f"Original intacta: {califs}")
```

Explicación:

- `float(input(...))` convierte lo que escribe el usuario a número decimal. Sin `float()`, `input()` devuelve cadena y `sum()` tronaría.
- Usamos `sorted()` (con `d`) para no modificar la lista original: buena práctica cuando la lista original todavía se necesita.
- El contador `altas` sigue el patrón "inicializar en 0, sumar 1 cuando cumple la condición".

Ejercicio 3: Mitades y rebanadas

Dada la lista `numeros = list(range(1, 21))` (números del 1 al 20), obtén con slicing: la primera mitad, la segunda mitad, los primeros 5, los últimos 5, los pares, los impares y la lista invertida. Imprime cada resultado con una etiqueta.

Solución paso a paso:

```
# 1. Lista del 1 al 20 (range genera 1..20, list lo convierte en lista)
numeros = list(range(1, 21))
print("Original:", numeros)

# 2. Mitades: el punto medio es len // 2 = 10
mitad = len(numeros) // 2
primera_mitad = numeros[:mitad]
segunda_mitad = numeros[mitad:]
print("Primera mitad:", primera_mitad)
print("Segunda mitad:", segunda_mitad)

# 3. Primeros 5 y últimos 5
print("Primeros 5:", numeros[:5])
print("Últimos 5:", numeros[-5:])

# 4. Pares e impares con paso de 2
# Los pares están en los índices 1, 3, 5... (valores 2, 4, 6...)
pares = numeros[1::2]
# Los impares están en los índices 0, 2, 4... (valores 1, 3, 5...)
impares = numeros[::2]
print("Pares:", pares)
print("Impares:", impares)

# 5. Invertida
print("Invertida:", numeros[::-1])
```

Explicación:

- `list(range(1, 21))` genera `[1, 2, ..., 20]`. `range(1, 21)` llega hasta el 20 porque el fin se excluye (igual que en slicing).
- Para los pares: el valor 2 está en el índice 1, el 4 en el índice 3... por eso empezamos en `[1::2]`. Los impares (1, 3, 5...) están en índices 0, 2, 4..., de ahí `[::2]`.
- Todo se resuelve con slicing, sin ciclos. Cuando domines el slicing, estos problemas se vuelven de una línea.

Ejercicio 4: Agenda con buscar y eliminar

Extiende la agenda del Ejemplo 4: agrega una función `eliminar(nombre)` que quite el contacto por nombre (verificando primero con `in` -lógica que exista), y una función `actualizar(nombre, nuevo_telefono)` que cambie el teléfono. Prueba todo.

Solución paso a paso:

```

# 1. Agenda inicial
agenda = [
    ["Ana", "555-1234"],
    ["Luis", "555-5678"],
    ["María", "555-9012"],
]

# 2. Eliminar: buscamos el ÍNDICE del contacto y usamos pop()
def eliminar(nombre):
    for i, contacto in enumerate(agenda):
        if contacto[0].lower() == nombre.lower():
            quitado = agenda.pop(i) # pop devuelve el contacto eliminado
            print(f"{quitado[0]} eliminado de la agenda.")
            return
    print(f"{nombre} no está en la agenda.")

# 3. Actualizar: encontramos el contacto y cambiamos su teléfono
def actualizar(nombre, nuevo_telefono):
    for contacto in agenda:
        if contacto[0].lower() == nombre.lower():
            viejo = contacto[1]
            contacto[1] = nuevo_telefono # las listas son mutables
            print(f"Teléfono de {contacto[0]}: {viejo} -> {nuevo_telefono}")
            return
    print(f"{nombre} no está en la agenda.")

# 4. Mostrar agenda
def mostrar():
    print("\n--- Agenda ---")
    for nombre, telefono in agenda:
        print(f"{nombre}: {telefono}")

# 5. Probamos todo
mostrar()
eliminar("luis") # funciona con minúsculas
eliminar("Hugo") # no existe: mensaje amable, sin error
actualizar("María", "555-0000") # cambia el teléfono
actualizar("Hugo", "555-0000") # no existe: mensaje amable
mostrar()

```

Explicación:

- En eliminar necesitamos el **índice** para usar `pop(i)`, por eso recorremos con `enumerate()`. `pop()` además nos devuelve el contacto eliminado para confirmar qué se quitó.
- En actualizar aprovechamos la **mutabilidad** de las listas: `contacto[1] = nuevo_telefono` cambia el teléfono dentro de la lista original, sin crear nada nuevo.
- Ambas funciones comparan con `.lower()` para no distinguir mayúsculas, y si no encuentran el nombre muestran un mensaje en vez de tronar. Esa es la diferencia entre un programa frágil y uno robusto.
- El `return` sin valor termina la función en cuanto cumple su misión; si el ciclo termina sin `return`, es que no encontró nada.

Ejercicio 5: Coordenadas y distancias con tuplas

Crea una lista de 4 puntos (tuplas `(x, y)`). Calcula la distancia entre cada par consecutivo de puntos (del 1 al 2, del 2 al 3, del 3 al 4) y la distancia total del recorrido. Luego intenta modificar un punto y observa el error.

Solución paso a paso:

```

import math

# 1. Ruta de 4 puntos, cada uno es una tupla inmutable
ruta = [(0, 0), (3, 4), (6, 4), (6, 8)]

# 2. Función que calcula la distancia entre dos puntos (tuplas)
def distancia(p1, p2):
    x1, y1 = p1    # desempaquetado de tuplas
    x2, y2 = p2
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

# 3. Distancia entre pares consecutivos
total = 0
for i in range(len(ruta) - 1):
    d = distancia(ruta[i], ruta[i + 1])
    print(f"Tramo {i + 1}: de {ruta[i]} a {ruta[i + 1]} = {d:.2f}")
    total += d

print(f"\nDistancia total del recorrido: {total:.2f}")

# 4. Intentar modificar un punto truena (protección de la tupla)
try:
    ruta[0] = (10, 10)    # ¡OJO! Esto SÍ funciona...
    print("¿Se pudo? La LISTA es mutable, la TUPLA no.")
    print("Punto modificado:", ruta[0])
    ruta[0][0] = 99       # ...pero ESTO truena
except TypeError as e:
    print(f"La tupla no se deja modificar: {e}")

```

Explicación:

- ruta es una **lista de tuplas**: la lista puede cambiar (agregar más puntos a la ruta), pero cada punto individual es inmutable.
- `range(len(ruta) - 1)` genera los índices 0, 1, 2: comparamos el punto `i` con el punto `i + 1`. El `- 1` evita el `IndexError` en el último (no hay punto después del último).
- El punto 4 es la lección clave: `ruta[0] = (10, 10)` **sí** funciona porque modifica la **lista** (cambia qué tupla está en la posición 0). Pero `ruta[0][0] = 99` **truena** porque intenta modificar la **tupla** por dentro. Lista mutable + tupla inmutable = estructura flexible pero con datos protegidos. Esa combinación aparece en programas reales todo el tiempo.

Ejercicio 6: Sin duplicados, con conteo

Dada una lista de votos ["Ana", "Luis", "Ana", "Pedro", "Luis", "Ana"], elimina los duplicados manteniendo el orden y luego muestra cuántos votos tuvo cada candidato usando `count()`.

Solución paso a paso:

```
# 1. Votos con repetidos
votos = ["Ana", "Luis", "Ana", "Pedro", "Luis", "Ana"]

# 2. Quitamos duplicados manteniendo el orden (método del ciclo)
candidatos = []
for nombre in votos:
    if nombre not in candidatos:
        candidatos.append(nombre)

print("Candidatos:", candidatos)

# 3. Contamos los votos de cada candidato con count()
print("\n--- Resultados ---")
for candidato in candidatos:
    total = votos.count(candidato)
    print(f"{candidato}: {total} votos")

# 4. Declaramos al ganador con max() sobre una lista de tuplas
resultados = [(votos.count(c), c) for c in candidatos]
ganador_votos, ganador_nombre = max(resultados)
print(f"\nGanador: {ganador_nombre} con {ganador_votos} votos")
```

Explicación:

- Primero obtenemos la lista de candidatos únicos **en orden de aparición** (el método del `not in + append()`).
- `votos.count(candidato)` cuenta sobre la lista **original** (con duplicados), que es donde están todos los votos.
- El paso 4 usa un **list comprehension** para armar tuplas `(votos, nombre)` y `max()` elige la tupla con más votos. Ponemos los votos **primero** en la tupla porque `max()` compara el primer elemento. Luego desempaquetamos al ganador.

- Fíjate cómo se combinan las herramientas de la unidad: listas, `append()`, `in`, `count()`, tuplas, desempaquetado y `max()`. Así se programan las cosas reales: combinando piezas simples.

Resumen de la unidad

Llegaste al final de la Unidad 6. Esto es lo esencial que debes llevarte:

Listas:

- Una lista es una colección **ordenada y mutable** entre corchetes `[]`. Reemplaza a cien variables sueltas y se recorre con un solo `for`.
- Los **índices empiezan en 0**. Los **índices negativos** cuentan desde el final: `-1` es el último.
- El **slicing** `[inicio:fin]` toma una rebanada (el fin se excluye). `[::2]` salta de 2 en 2, `[::-1]` invierte.
- Las listas son **mutables**: puedes cambiar elementos, rebanadas, agregar y quitar en caliente.
- Métodos clave: `append()` (uno al final), `extend()` (muchos al final), `insert()` (en posición), `remove()` (por valor), `pop()` (por índice y te lo devuelve), `clear()` (vacía), `index()` (posición de un valor), `count()` (cuántas veces aparece), `sort()` (ordena en su lugar, devuelve `None`), `reverse()` (invierte en su lugar), `copy()` (copia independiente).
- Funciones útiles: `len()`, `min()`, `max()`, `sum()`. El operador `in` pregunta si un valor existe.
- `enumerate()` te da índice y valor al mismo tiempo en el `for`.

Tuplas:

- Una tupla es como una lista pero **inmutable**, entre paréntesis `()`. Se usa para datos que no deben cambiar: coordenadas, RGB, días, configuraciones.
- La inmutabilidad es una **ventaja**: si algo intenta cambiar el dato, Python truena de inmediato en vez de corromperlo en silencio.
- **Tupla de un elemento lleva coma**: `(5,)`. Sin coma no es tupla.
- ¿Lista o tupla? Si los datos cambian -> lista. Si son fijos -> tupla. Si dudas -> lista.

Errores que ya no te van a atorar:

- `IndexError`: pediste un índice que no existe. El último índice válido es `len(lista) - 1`; usa `-1` para el último.
- `remove()` es por **valor**, `pop()` es por **índice** (y devuelve el valor).
- `sort()` devuelve `None`; para una copia ordenada usa `sorted()`.
- `b = a` **no copia**, crea un alias: dos nombres, una sola lista. Copia con `.copy()`, `[:]`, o `list()`.
- No modifiques la lista mientras la recorres con `for`.

En la siguiente unidad vas a conocer los **diccionarios**: colecciones donde en vez de índices numéricos usas **claves con nombre**, como un directorio telefónico de verdad. Las listas guardan cosas *en orden*; los diccionarios guardan cosas *por nombre*. Combinados, son el 90% de las estructuras de datos que usarás en tu vida como programador.

Nos vemos en la Unidad 7. ¡A programar!

UNIDAD 7: Diccionarios, Sets y Texto Avanzado

- Tu Caja de Herramientas de Datos Etiquetados

Introducción

Imagina que llegas a una papelería en México y pides un diccionario de verdad, de esos gordos de la Real Academia Española. ¿Cómo lo usas? No empiezas a leer desde la primera página, de la A a la Z, esperando llegar a la palabra que buscas. No. Abres el libro directo por la letra que necesitas, buscas la palabra exacta y ahí está su definición, sin rodeos. Nadie busca una palabra en un diccionario recorriéndolo página por página como si fuera una novela. Lo consultas POR LA PALABRA, porque cada palabra tiene su entrada y cada entrada tiene su significado.

Pues así son los diccionarios en Python. Hasta ahora trabajaste con listas y tuplas, que son como filas de casilleros numerados: el elemento 0, el elemento 1, el elemento 2... Para encontrar algo tienes que saber su posición o recorrer la lista completa. Un diccionario, en cambio, es como el diccionario de la papelería: cada dato tiene una ETIQUETA, una palabra clave que lo identifica, y lo encuentras al instante con esa etiqueta, sin importar en qué orden estén guardadas las cosas. Se les dice "pares clave-valor": la clave es la palabra que buscas (como "nombre"), y el valor es lo que encuentras (como "María"). Así de directo.

Ahora piensa en otra imagen muy mexicana: tu credencial del INE. Esa tarjetita no es una lista de datos sueltos en orden misterioso. Es un conjunto de datos ETIQUETADOS: NOMBRE: Juan Pérez, CURP: XXXX, DIRECCIÓN: Calle Falsa 123, FECHA DE NACIMIENTO: 15/03/1995. Cada campo tiene su etiqueta y su valor. Si alguien te pregunta tu CURP, no recitas toda la credencial desde el inicio; vas directo a la etiqueta CURP y lees el valor. Un diccionario de Python es exactamente eso: una credencial de datos donde cada valor tiene su etiqueta clara. Por eso los diccionarios son la estructura favorita de los programadores para representar cosas del mundo real: personas, productos, configuraciones, cuentas, pedidos, registros de base de datos, respuestas de APIs. Todo lo que tiene "campos con nombre" se modela con un diccionario.

Y ahora la tercera imagen, para los sets: imagina una urna de cristal en una rifa de la escuela. Metes tu papelito con tu nombre, tu vecino mete el suyo, y así. Nadie mete dos papelitos con el mismo nombre porque sería trampa, ¿verdad? En la urna cada nombre aparece UNA sola vez, no importa cuántas veces lo intenten meter. Además, en la urna los papelitos están

revueltos: no hay orden, no hay "el primero" ni "el último". Los sets en Python son eso: colecciones de elementos ÚNICOS y SIN ORDEN. Su superpoder no es guardar cosas en orden, sino responder rapidísimo a la pregunta "¿esto está aquí o no?". Son velocísimos para verificar membresía, y además te regalan operaciones de matemáticas de conjuntos que ya viste en la prepa: unión, intersección y diferencia. Y tienen un truco estrella: quitarle los duplicados a una lista en una sola línea de código.

Por último, el texto avanzado. Ya sabes que los strings son texto entre comillas, pero en esta unidad vas a descubrir que Python trata al texto como un material que puedes moldear: convertirlo a mayúsculas o minúsculas, recortarle los espacios de los bordes, partirlo en pedazos por donde tú digas, reemplazar palabras, pegarlo de vuelta con el separador que quieras, preguntarle si empieza o termina con algo, buscar dentro de él, contar cuántas veces aparece una letra, validar si es un número, rellenarlo con ceros, centrarlo, alinearlo a la derecha o a la izquierda. Esto no es decoración: es lo que haces TODO el tiempo en programas reales. Limpiar lo que el usuario escribió en un formulario (quitarle espacios de más), normalizar un correo electrónico a minúsculas, partir una línea de un archivo CSV por las comas, generar un reporte con columnas bien alineadas, validar que una contraseña tenga solo letras y números. El que domina los métodos de strings domina la mitad del trabajo de cualquier programa que procese texto, que son casi todos.

Una advertencia de mapa antes de arrancar: esta unidad junta tres temas que parecen distintos pero que en la vida real trabajan juntos. El ejemplo clásico es el analizador de texto: tomas un párrafo, lo partes en palabras con `split()`, limpias cada palabra con `lower()` y `strip()`, y cuentas cuántas veces aparece cada una usando un diccionario donde la clave es la palabra y el valor es el contador. Ahí están los tres temas de la unidad bailando juntos en un solo programa. Cuando termines esta unidad vas a sentir que Python por fin se volvió una herramienta de adulto: ya no solo guardas numeritos en listas, ahora manejas registros etiquetados, colecciones únicas y texto moldeable como plastilina. Eso es lo que separa a quien "sabe un poco de Python" de quien puede construir cosas de verdad.

Vamos por partes, sin prisa pero sin pausa. Primero los diccionarios, que son la estrella de la unidad. Luego los sets. Luego el texto avanzado. Y al final, práctica para que todo esto se te quede grabado en los dedos, no solo en la cabeza.

Explicación detallada

Los diccionarios: datos con etiqueta

Un diccionario en Python se escribe entre llaves `{}` y guarda pares `clave: valor` separados por comas. La clave es la etiqueta (casi siempre un string, aunque puede ser un número o una tupla) y el valor puede ser CUALQUIER cosa: un número, un string, una lista, otro diccionario, un booleano. Mira la credencial del INE convertida en diccionario:

```
alumno = {  
    "nombre": "María González",  
    "edad": 22,  
    "carrera": "Ingeniería",  
    "promedio": 9.2,  
    "becada": True  
}
```

Léelo en voz alta: la clave `"nombre"` tiene el valor `"María González"`, la clave `"edad"` tiene el valor `22`, y así. Los dos puntos: separan la etiqueta de su valor, como en un formulario. Fíjate en algo importante: las claves son únicas. No puedes tener dos veces la clave `"nombre"` en el mismo diccionario, igual que tu credencial no tiene dos campos que se llamen igual. Si intentas repetir una clave, Python se queda con el último valor que le diste, sin avisarte. Eso es una fuente clásica de errores silenciosos, así que grábalo: claves únicas, siempre.

También puedes crear un diccionario vacío (`{}` o `dict()`) y llenarlo después, o usar el atajo `dict(nombre="Juan", edad=30)` cuando las claves son strings simples: te ahorra comillas y dos puntos, y el resultado es idéntico.

Acceder a los valores por su clave: búsqueda instantánea

Aquí está la magia del diccionario. Para leer un valor, pones la clave entre corchetes, igual que con las listas pones el índice, pero aquí pones la ETIQUETA:

```

alumno = {
    "nombre": "María González",
    "edad": 22,
    "carrera": "Ingeniería",
    "promedio": 9.2,
    "becada": True
}

print(alumno["nombre"])    # María González
print(alumno["edad"])      # 22
print(alumno["promedio"])  # 9.2

```

¿Ves la diferencia con una lista? En una lista dirías `alumno[0]` y tendrías que acordarte de que la posición 0 es el nombre. En el diccionario dices `alumno["nombre"]` y el código se lee solo, como español. Por eso los programas profesionales usan diccionarios para todo lo que tenga campos con nombre: el código se vuelve autodocumentado. Cuando dentro de seis meses regreses a tu programa y leas `pedido["total"]`, vas a entenderlo al instante. Si hubieras usado una lista y leyeras `pedido[4]`, tendrías que adivinar qué diablos era la posición 4.

Y aquí viene la primera gran lección de errores: si pides una clave que NO existe, Python no te devuelve un cero ni un texto vacío ni te lo perdona. Te lanza un `KeyError` en la cara y tu programa se detiene. Míralo:

```

alumno = {"nombre": "María", "edad": 22}
print(alumno["telefono"])
# KeyError: 'telefono'

```

El `KeyError` es el error más común de esta unidad y te va a pasar MUCHAS veces, sobre todo al principio. No es tu enemigo: es Python diciéndote "oye, esa etiqueta no existe en este diccionario, revisa que la escribiste bien". El 90% de las veces es un error de dedo: escribiste "Nombre" con mayúscula cuando la clave era "nombre" con minúscula, o "telefono" sin acento cuando era "teléfono". Las claves son sensibles a mayúsculas y a cada carácter: "Nombre", "nombre" y "NOMBRE" son tres claves DISTINTAS. Más adelante verás cómo defenderte del `KeyError` con el método `get()`, que es como pedir la palabra en el diccionario real con la opción de "si no la encuentras, dime esto otro en lugar de enojarte".

Agregar y modificar valores: el diccionario es mutable

Los diccionarios son mutables, como las listas: puedes cambiarlos después de crearlos. Para agregar un par nuevo o modificar uno existente usas la misma sintaxis de corchetes con asignación:

```
alumno = {"nombre": "María", "edad": 22}

# Agregar una clave nueva
alumno["telefono"] = "555-123-4567"
print(alumno)
# {'nombre': 'María', 'edad': 22, 'telefono': '555-123-4567'}

# Modificar una clave existente
alumno["edad"] = 23
print(alumno)
# {'nombre': 'María', 'edad': 23, 'telefono': '555-123-4567'}
```

Fíjate: la MISMA línea de código hace dos cosas distintas según el caso. Si la clave no existía, la crea. Si ya existía, la sobrescribe. Python no te pregunta cuál querías hacer, simplemente lo hace. Esto es poderosísimo pero también es la razón por la que debes tener cuidado con los nombres de tus claves: si por error escribes `alumno["edad"] = 23` cuando querías agregar otra cosa, acabas de pisar el valor anterior sin que nadie te avise. En programas grandes, un error de dedo en una clave puede corromper datos silenciosamente. Por eso muchos programadores definen sus claves como constantes o las escriben con mucho cuidado.

Eliminar pares: del y pop()

Hay dos formas de quitarle cosas a un diccionario. La primera es `del`, que borra el par completo (clave y valor) como si nunca hubiera existido:

```
alumno = {"nombre": "María", "edad": 22, "telefono": "555-123-4567"}
del alumno["telefono"]
print(alumno)
# {'nombre': 'María', 'edad': 22}
```

La segunda es `pop()`, que además de borrar TE DEVUELVE el valor eliminado, como cuando sacas algo de una caja y te lo quedas en la mano:

```
alumno = {"nombre": "María", "edad": 22, "telefono": "555-123-4567"}
tel = alumno.pop("telefono")
print(tel)      # 555-123-4567
print(alumno)   # {'nombre': 'María', 'edad': 22}
```

¿Cuándo usar cada uno? Usa `del` cuando solo quieres deshacerte del dato y no te importa su valor (por ejemplo, borrar un campo temporal). Usa `pop()` cuando necesitas el valor para algo antes de que desaparezca (por ejemplo, mover un dato de un diccionario a otro, o registrar en un log qué se eliminó). Y ojo con el detalle importante: tanto `del` como `pop()` lanzan `KeyError` si la clave no existe. `pop()`, sin embargo, acepta un segundo argumento como valor de respaldo para no tronar:

```
alumno = {"nombre": "María", "edad": 22}
tel = alumno.pop("telefono", "sin teléfono")
print(tel) # sin teléfono (no hubo error)
```

Ese segundo argumento de `pop()` funciona igual que el valor por defecto de `get()`, que veremos ahorita. Es el mismo truco de "si no lo encuentras, usa esto".

También existe `clear()`, que vacía el diccionario dejándolo en `{}`, y `del alumno` (sin corchetes), que borra la variable entera. Úsalos con cuidado.

keys(), values() e items(): ver el diccionario por partes

A veces no quieres un valor específico, sino ver TODO lo que tiene el diccionario: todas las etiquetas, todos los valores, o todos los pares. Para eso existen tres métodos:

```
alumno = {"nombre": "María", "edad": 22, "carrera": "Ingeniería"}

print(alumno.keys())
# dict_keys(['nombre', 'edad', 'carrera'])

print(alumno.values())
# dict_values(['María', 22, 'Ingeniería'])

print(alumno.items())
# dict_items([('nombre', 'María'), ('edad', 22), ('carrera', 'Ingeniería')])
```

`keys()` te da todas las claves, `values()` todos los valores, y `items()` los pares completos como tuplas (clave, valor). Lo que devuelven se ve raro (`dict_keys`, etc.), pero en la

práctica casi siempre los usas dentro de un `for` o los conviertes a lista con `list()`. El más útil es `items()`: te permite recorrer el diccionario con clave y valor a la vez, ideal para mostrar todos los campos de un registro sin saber de antemano cuáles son.

get(): la forma segura de preguntar por una clave

Este método merece su propia fanfarria porque te va a salvar de cientos de `KeyError`. `get()` hace lo mismo que los corchetes (buscar el valor de una clave), pero si la clave NO existe, en lugar de tronar te devuelve `None` (o el valor por defecto que tú le digas):

```
alumno = {"nombre": "María", "edad": 22}

print(alumno.get("nombre"))      # María
print(alumno.get("telefono"))    # None (sin error)
print(alumno.get("telefono", "no registrado")) # no registrado
```

El segundo argumento de `get()` es tu red de seguridad. Úsalo SIEMPRE que no estés 100% seguro de que la clave existe: datos que vienen de un usuario, de un archivo, de una API o de una base de datos, donde los campos pueden faltar. Regla de oro: si la clave viene de datos externos o inciertos, usa `get()`; si la pusiste tú mismo dos líneas arriba y sabes que existe, los corchetes están bien.

Un patrón muy común con `get()` es contar cosas, que verás en los ejemplos:
`contador[palabra] = contador.get(palabra, 0) + 1`. Léelo: "el nuevo valor es lo que había antes (o cero si es la primera vez) más uno". Sin `get()` tendrías que escribir un `if` para preguntar si la clave ya existe. Con `get()` es una sola línea elegante.

El operador in: preguntar si una clave existe

Antes de pedir un valor, puedes preguntar si la clave existe con `in`. Ojo: en diccionarios, `in` busca en las CLAVES, no en los valores:

```
alumno = {"nombre": "María", "edad": 22}

print("nombre" in alumno)      # True
print("telefono" in alumno)    # False
print("María" in alumno)       # False (busca en claves, no en valores)
```

Esa última línea sorprende a muchos: `"María" in alumno` da `False` porque `"María"` es un VALOR, no una clave. Si necesitas buscar en los valores, usa `in` sobre `alumno.values()`:

"María" in alumno.values() da True. El uso típico de in con diccionarios es validar antes de acceder: if "telefono" in alumno: y dentro del if usar los corchetes con confianza. Es la alternativa manual a get(): in + corchetes es para cuando quieres hacer algo distinto según exista o no la clave; get() es para cuando solo quieres el valor con un respaldo.

Recorrer un diccionario con for: tres sabores

El for con diccionarios tiene tres formas, y cada una sirve para algo distinto. La forma básica recorre las CLAVES:

```
alumno = {"nombre": "María", "edad": 22, "carrera": "Ingeniería"}

for clave in alumno:
    print(clave)
# nombre
# edad
# carrera
```

Si dentro del ciclo necesitas el valor, lo pides con la clave:

```
for clave in alumno:
    print(clave, "->", alumno[clave])
# nombre -> María
# edad -> 22
# carrera -> Ingeniería
```

Pero la forma más elegante y más usada es con items(), que te da clave y valor desempquetados en dos variables de una vez:

```
for clave, valor in alumno.items():
    print(f"{clave}: {valor}")
# nombre: María
# edad: 22
# carrera: Ingeniería
```

Esa línea `for clave, valor in alumno.items():` es de las más comunes en todo Python profesional. Apréndetela de memoria: es el patrón estándar para recorrer un diccionario completo. Y si solo te interesan los valores, existe `for valor in alumno.values():`. En resumen: `for k in d` para claves, `for v in d.values()` para valores, `for k, v in d.items()` para ambos. Elige según lo que necesites dentro del ciclo y tu código se leerá perfecto.

Desde Python 3.7 los diccionarios conservan el orden de inserción de las claves, pero no abusos de ello: si el orden es lo importante para tu programa, probablemente necesitas una lista. El diccionario está optimizado para búsqueda por clave, no para orden.

Diccionarios anidados: registros dentro de registros

Recuerda que un valor de diccionario puede ser CUALQUIER cosa, incluyendo OTRO diccionario. Esto te permite modelar datos del mundo real con toda su estructura, como registros de una base de datos o respuestas de una API en formato JSON (que, dato curioso, está inspirado directamente en los diccionarios de Python):

```
tienda = {
    "producto_001": {
        "nombre": "Tortillero",
        "precio": 85.50,
        "stock": 40,
        "categoria": "cocina"
    },
    "producto_002": {
        "nombre": "Molcajete",
        "precio": 320.00,
        "stock": 12,
        "categoria": "cocina"
    }
}
```

Aquí la clave externa es el código del producto y el valor es OTRO diccionario con los datos de ese producto. Para llegar a un dato profundo, encadenas los corchetes como si bajaras niveles:

```
print(tienda["producto_001"]["nombre"])    # Tortillero
print(tienda["producto_002"]["precio"])    # 320.0
```

Léelo de izquierda a derecha: "en la tienda, busca el producto_001, y dentro de él, dame el nombre". Cada par de corchetes baja un nivel. Para modificar un valor anidado haces lo mismo con asignación:

```
tienda["producto_001"]["stock"] = 35 # se vendieron 5 tortilleros
```

Y puedes combinar `get()` en cada nivel para no tronar si falta algo:

`tienda.get("producto_999", {}).get("precio", 0)` busca el producto, usa un diccionario vacío si no existe, y devuelve 0 si falta el precio. Es el patrón defensivo estándar para datos que vienen de internet. Los diccionarios anidados también se combinan con listas: una lista de diccionarios (cada alumno un diccionario) es como una tabla donde cada fila es un registro etiquetado, y es la estructura más usada en el mundo real.

¿Cuándo usar diccionario y cuándo lista? La decisión de diseño

Esta es una de las decisiones más importantes que vas a tomar como programador, así que vamos a dejarla cristalina con una regla práctica:

Usa LISTA cuando el ORDEN importa y los elementos son del mismo tipo o no necesitan etiqueta. Ejemplos: la lista de calificaciones de un alumno `[9, 8, 10, 7]`, los nombres de los invitados a una fiesta, los pasos de una receta en orden, el historial de movimientos de una cuenta. En una lista, la posición tiene significado (el primero, el segundo...) y normalmente recorres todo.

Usa DICCIONARIO cuando cada dato tiene un NOMBRE y lo vas a buscar por ese nombre. Ejemplos: los datos de un alumno (nombre, edad, promedio), la configuración de un programa (idioma, tema, volumen), un producto (nombre, precio, stock), las respuestas de una API. En un diccionario, la etiqueta tiene significado y normalmente buscas uno específico.

La prueba de fuego: si escribes comentarios como `# posición 0 = nombre, posición 1 = edad`, DEJA la lista y usa un diccionario: esos comentarios son la señal de que tus datos piden etiquetas. Y al revés: un diccionario con claves `"elemento_1"`, `"elemento_2"` probablemente quería ser una lista. En rendimiento, buscar por clave en un diccionario es casi instantáneo sin importar el tamaño, mientras que buscar en una lista exige recorrerla; si vas a buscar mucho por nombre o código, el diccionario gana por goleada.

Los sets: la urna de papelitos únicos

Un set (conjunto) en Python es una colección de elementos ÚNICOS y SIN ORDEN GARANTIZADO. Se escribe entre llaves como el diccionario, pero SIN los dos puntos de los pares clave-valor:

```
frutas = {"manzana", "pera", "mango", "manzana", "pera"}
print(frutas)
# {'mango', 'pera', 'manzana'} (el orden puede variar)
```

Metiste "manzana" y "pera" dos veces, pero el set guardó cada una UNA sola vez: los duplicados desaparecen solos, como los papelitos repetidos en la urna de la rifa. Y fíjate en el orden: salió mango, pera, manzana, no como los metiste. Los sets NO garantizan orden: NUNCA confíes en él ni intentes acceder por posición (`frutas[0]` da error, los sets no aceptan índices).

Trampa famosa: `{}` NO es un set vacío, es un DICCIONARIO vacío; el set vacío se crea con `set()`. Grábatelo: llaves vacías = dict, `set()` = set vacío (típico error de examen). También puedes crear un set desde una lista con `set(mi_lista)`, el truco estrella para eliminar duplicados.

Otra restricción: los elementos de un set deben ser INMUTABLES (strings, números, tuplas, booleanos), nunca listas ni diccionarios; si lo intentas obtienes `TypeError: unhashable type`. Los sets usan "hash" interno para encontrar elementos rapidísimo, y solo lo inmutable puede hashearse de forma confiable.

Agregar y quitar elementos: add, remove y discard

Los sets son mutables: puedes agregar y quitar elementos después de crearlos. Para agregar usas `add()` (en singular, porque agrega UN elemento):

```
colores = {"rojo", "azul"}
colores.add("verde")
print(colores) # {'rojo', 'azul', 'verde'}
colores.add("rojo") # ya existía, no pasa nada, sigue igual
```

Fíjate en ese detalle: si agregas algo que ya está, el set simplemente lo ignora, sin error y sin duplicarlo. Es su naturaleza: cada elemento existe una vez o no existe.

Para quitar hay DOS métodos y la diferencia entre ellos es un error clásico. `remove()` quita el elemento, pero si NO existe te lanza un `KeyError` (sí, `KeyError`, aunque sea un set). `discard()` quita el elemento, pero si no existe NO HACE NADA, sin error:

```
colores = {"rojo", "azul", "verde"}
colores.remove("azul")
print(colores) # {'rojo', 'verde'}
colores.remove("morado") # KeyError: 'morado' (¡trueno!)

colores.discard("verde")
print(colores) # {'rojo'}
colores.discard("morado") # no pasa nada, sigue la vida
```

Recomendación: usa `discard()` cuando no estés seguro de que el elemento está, y `remove()` cuando su ausencia sea un error de lógica que QUIERES que true. También existen `pop()` (quita y devuelve un elemento arbitrario, porque no hay orden) y `clear()` (vacía el set).

Operaciones de conjuntos: unión, intersección y diferencia

Aquí es donde los sets se ponen elegantes, porque implementan directamente las operaciones de la teoría de conjuntos que viste en matemáticas. Imagina dos grupos de alumnos: los que juegan fútbol y los que juegan básquetbol. Algunos juegan los dos.

La UNIÓN (`|` o el método `union()`) te da TODOS los elementos que están en cualquiera de los dos sets, sin duplicados:

```
futbol = {"Ana", "Luis", "Pedro", "María"}
basquet = {"Luis", "María", "Sofía", "Jorge"}

todos = futbol | basquet
print(todos)
# {'Ana', 'Luis', 'Pedro', 'María', 'Sofía', 'Jorge'}
```

Seis nombres únicos: los 4 de fútbol más los 2 de básquet que no estaban en fútbol. Caso de uso real: tienes la lista de clientes de enero y la de febrero (con repetidos entre meses) y quieres la lista de clientes ÚNICOS del bimestre. Una línea con `|` y listo.

La INTERSECCIÓN (`&` o `intersection()`) te da SOLO los elementos que están en AMBOS sets:

```
ambos = futbol & basquet
print(ambos) # {'Luis', 'María'}
```

Luis y María juegan los dos deportes. Caso de uso real: tienes los usuarios que abrieron tu correo y los que compraron algo, y quieres saber quiénes hicieron AMBAS cosas para mandarle un agradecimiento especial. O en seguridad: los permisos que pide una app INTERSECTADOS con los permisos peligrosos te dicen qué riesgos hay.

La DIFERENCIA (- o `difference()`) te da los elementos que están en el primer set pero NO en el segundo. Ojo: el orden importa, no es simétrica:

```
solo_futbol = futbol - basquet
print(solo_futbol) # {'Ana', 'Pedro'}

solo_basquet = basquet - futbol
print(solo_basquet) # {'Sofía', 'Jorge'}
```

Caso de uso real: tienes el set de todos los alumnos inscritos y el set de los que entregaron la tarea; la diferencia te da QUIÉNES NO la entregaron, para mandarles el recordatorio. O en inventarios: productos en catálogo menos productos en existencia = productos agotados que hay que resurtir. También existe la diferencia simétrica (^ o `symmetric_difference()`), que te da los elementos que están en uno U otro pero no en ambos (los "solo de un lado" de los dos lados juntos). Es menos común, pero cuando la necesitas te ahorra varias líneas.

Estos operadores también tienen sus versiones como métodos (`union()`, `intersection()`, `difference()`), que hacen exactamente lo mismo. Usa operadores (`|`, `&`, `-`) cuando el código es corto y legible, y métodos cuando quieras ser más explícito o encadenar operaciones con nombres claros.

El truco estrella: eliminar duplicados de una lista en una línea

Esta es la aplicación más famosa de los sets y la que más vas a usar en la vida real. Tienes una lista con repetidos y quieres quedarte solo con los valores únicos:

```
invitados = ["Ana", "Luis", "Ana", "Pedro", "Luis", "Ana", "Sofía"]
unicos = list(set(invitados))
print(unicos)
# ['Sofía', 'Pedro', 'Ana', 'Luis'] (orden no garantizado)
```

Tres pasos en uno: `set(invitados)` convierte la lista en set (los duplicados desaparecen por arte de magia) y `list(...)` lo convierte de vuelta en lista. Es el patrón estándar y aparece en TODO el código Python del mundo. Sus usos reales son infinitos: limpiar una lista de correos con duplicados antes de mandar un mailing, obtener las categorías únicas de una lista de productos, saber cuántos valores distintos hay en una columna de datos (`len(set(datos))`) te da el conteo de únicos en una línea).

Dos advertencias. Primera: el orden original SE PIERDE. Si te importa el orden de primera aparición, usa `list(dict.fromkeys(lista))`: los diccionarios no permiten claves repetidas pero sí conservan el orden de inserción, un truco elegante que combina ambos temas de la unidad. Segunda: si la lista contiene elementos mutables (listas o diccionarios), el truco truena con `TypeError`; ahí toca limpiar a mano.

Membresía ultrarrápida: el superpoder oculto de los sets

La otra gran aplicación de los sets es preguntar "¿esto está aquí?" con `in`. En una lista, `in` recorre elemento por elemento; en un set es casi instantáneo sin importar el tamaño, gracias al hash interno. Con miles o millones de elementos la diferencia es abismal.

```
# Patrón profesional: convierte a set UNA vez, pregunta muchas veces
usuarios_bloqueados = {"troll123", "spam_bot", "cuenta_falsa", "hater99"}

def puede_comentar(usuario):
    if usuario in usuarios_bloqueados:
        return "Comentario rechazado"
    return "Comentario publicado"

print(puede_comentar("maria_garcia")) # Comentario publicado
print(puede_comentar("troll123"))    # Comentario rechazado
```

Este patrón -un set de "cosas prohibidas o permitidas" y verificar membresía- aparece en filtros de spam, validación de usuarios, palabras prohibidas en un chat y extensiones de archivo permitidas. Regla práctica: si harás MUCHAS verificaciones de `in` sobre la misma colección que no necesita orden ni duplicados, usa un set.

Métodos de strings a fondo: moldear texto como plastilina

Python trae docenas de métodos para transformar, limpiar, partir, buscar y validar texto. Antes de empezar, el recordatorio fundamental: los strings son INMUTABLES. Ningún método modifica el original; todos DEVUELVEN un string nuevo. Si no guardas el resultado, el trabajo se pierde:

```
texto = "hola mundo"
texto.upper()      # devuelve "HOLA MUNDO" pero no lo guarda
print(texto)       # hola mundo (sigue igual, el resultado se perdió)
texto = texto.upper() # AHORA sí se guarda
print(texto)       # HOLA MUNDO
```

Ese es el error número uno con métodos de strings: llamar al método sin asignar el resultado y luego preguntarse por qué "no funcionó". Siempre que transformes texto, guarda el resultado en una variable (puede ser la misma).

Cambiar mayúsculas y minúsculas: upper, lower, title, capitalize, swapcase

`upper()` convierte TODO a mayúsculas y `lower()` convierte TODO a minúsculas. Su uso más importante no es gritar en mayúsculas, sino NORMALIZAR texto para compararlo. Los usuarios escriben "México", "MÉXICO", "méxico", "MeXiCo", y para tu programa deberían ser lo mismo:

```
respuesta = input("¿De qué país eres? ")
if respuesta.lower() == "méxico":
    print("¡Paisano!")
```

Sin el `lower()`, quien escriba "MÉXICO" no pasaría la comparación. Normalizar a minúsculas antes de comparar es obligatorio en formularios, búsquedas y validaciones, y para correos (`Juan@Gmail.com` y `juan@gmail.com` son la misma cuenta).

`title()` capitaliza la primera letra de CADA palabra: `"juan perez".title()` da `"Juan Pérez"`, perfecto para arreglar nombres en minúsculas. `capitalize()` solo capitaliza la primera letra del texto completo (`"hOLA MUNDO" -> "Hola mundo"`), útil para frases. Y `swapcase()` invierte cada letra (`"HoLa" -> "hOLA"`), más un truco divertido que una herramienta diaria.

Limpiar los bordes: strip, lstrip y rstrip

Cuando un usuario llena un formulario, casi siempre mete espacios de más al inicio o al final sin querer: "Juan Pérez". Esos espacios invisibles rompen comparaciones ("Juan" != "Juan") y ensucian bases de datos. `strip()` les corta los espacios (y otros caracteres de espacio como tabulaciones y saltos de línea) de AMBOS lados:

```
nombre = "  María  "
print(nombre.strip()) # María
```

`lstrip()` limpia solo la izquierda y `rstrip()` solo la derecha: útil al leer archivos, donde quieres quitar el `\n` del final pero conservar la indentación del inicio. Y el poder extra que muchos no conocen: `strip()` acepta los CARACTERES a recortar, no solo espacios:

```
codigo = "###ABC123###"
print(codigo.strip("#")) # ABC123

precio = "$$$150$$$"
print(precio.strip("$")) # 150

linea = "---título---"
print(linea.strip("-")) # título
```

Ojo: no recorta la secuencia "###" como bloque, sino cualquier combinación de esos caracteres desde cada borde. Ideal para limpiar datos con delimitadores o basura en los bordes: comas, guiones, asteriscos, ceros.

Reemplazar texto: replace

`replace()` busca un fragmento y lo cambia por otro, en TODAS sus apariciones. Es el "buscar y reemplazar" de Word, pero en código:

```
texto = "Me gusta el café. El café es vida."
nuevo = texto.replace("café", "chocolate")
print(nuevo) # Me gusta el chocolate. El chocolate es vida.
```

Reemplazó las DOS apariciones; para limitarlas, pasa el número máximo como tercer argumento: `replace("café", "chocolate", 1)`. Casos de uso: censurar groserías en un chat, normalizar datos (`telefono.replace("-", "").replace(" ", "")`), generar plantillas ([NOMBRE] -> nombre real), o quitar acentos encadenando: `texto.replace("á",`

"a").replace("é", "e"). Cada `replace()` devuelve un string nuevo, por eso se pueden encadenar.

Partir texto: `split` y `rsplit`

`split()` es quizá el método más usado de todo Python para procesar texto: parte un string en una LISTA de pedazos, usando un separador que tú eliges. Sin argumentos, parte por cualquier espacio en blanco (espacios, tabulaciones, saltos de línea) y además ignora los espacios de más:

```
frase = "el rápido zorro salta"
palabras = frase.split()
print(palabras) # ['el', 'rápido', 'zorro', 'salta']
```

Con espacios triples y dobles, `split()` los manejó como si fueran uno solo: por eso `split()` sin argumentos es la forma estándar de obtener las palabras de un texto. Con argumento, partes por el separador exacto que le digas:

```
linea_csv = "Juan,25,Ingeniería,9.2"
campos = linea_csv.split(",")
print(campos) # ['Juan', '25', 'Ingeniería', '9.2']

fecha = "2026-09-19"
partes = fecha.split("-")
print(partes) # ['2026', '09', '19']

ruta = "documentos/trabajo/reporte.pdf"
print(ruta.split("/")) # ['documentos', 'trabajo', 'reporte.pdf']
```

Esto es ORO para procesar archivos: CSV por comas, fechas por guiones, rutas por diagonales. El patrón clásico es: `linea.strip()` para quitar el salto de línea y `linea.split(",")` para obtener los campos.

`split()` acepta un segundo argumento con el número MÁXIMO de divisiones:

"a,b,c,d".split(",", 2) da ['a', 'b', 'c,d'], útil cuando solo quieres partir por el PRIMER separador. Y `rsplit()` parte desde la DERECHA:

"archivo.tar.gz".rsplit(".", 1) da ['archivo.tar', 'gz'], perfecto para extensiones de archivos con varios puntos.

Pegar texto: join (el inverso de split)

`join()` hace exactamente lo contrario de `split()`: toma una LISTA de strings y la pega en UN solo string, poniendo entre cada elemento el separador que tú elijas. La sintaxis es al revés de lo que esperarías: el separador va primero, como string, y la lista va dentro del paréntesis:

```
palabras = ["el", "rápido", "zorro"]
frase = " ".join(palabras)
print(frase) # el rápido zorro

campos = ["Juan", "25", "Ingeniería"]
linea = ",".join(campos)
print(linea) # Juan,25,Ingeniería

letras = ["P", "y", "t", "h", "o", "n"]
print("".join(letras)) # Python (separador vacío = pegado total)
print("-".join(letras)) # P-y-t-h-o-n
```

El error típico es escribir `palabras.join(" ")` al revés; no funciona así: el string del método es "el pegamento" que va ENTRE los elementos. Casos de uso: reconstruir líneas CSV, crear rutas (`"/".join(["docs", "trabajo", "reporte.pdf"])`), o mostrar listas bonitas (`" ".join(nombres) -> "Ana, Luis, Pedro"`). El par `split()` / `join()` es el dúo dinámico del texto: partes, transformas la lista, y vuelves a pegar.

Preguntar por el inicio y el final: `startswith` y `endswith`

Estos dos devuelven `True` o `False` según si el string empieza o termina con el fragmento que les des:

```
archivo = "reporte_final.pdf"
print(archivo.startswith("reporte")) # True
print(archivo.endswith(".pdf"))      # True
print(archivo.endswith(".docx"))     # False
```

Son los guardianes de la validación de formatos: verificar extensiones (`archivo.endswith((".jpg", ".png", ".gif"))` acepta una TUPLA de opciones), URLs que empiecen con `https://`, líneas de comentario que empiecen con `#`, o correos de cierto dominio. Mucho más legibles que comparar rebanadas a mano.

Buscar dentro del texto: find e index

`find()` te dice EN QUÉ POSICIÓN aparece un fragmento dentro del string (la posición donde EMPIEZA). Si no lo encuentra, devuelve `-1` en lugar de tronar:

```
texto = "Aprende Python con Lucas EGA"
print(texto.find("Python")) # 8
print(texto.find("Java"))   # -1 (no está, sin error)
```

`index()` hace lo mismo pero lanza `ValueError` si no encuentra: `find()` es la versión amable, `index()` la estricta. Puedes indicar DÓNDE empezar: `texto.find("a", 10)` busca desde la posición 10, útil para la segunda o tercera aparición. Y `rfind()` busca desde la derecha (la ÚLTIMA aparición): ideal para la última diagonal de una ruta o el último punto de un archivo.

Contar apariciones: count

`count()` cuenta cuántas veces aparece un fragmento (sin traslapes) dentro del string:

```
texto = "la casa de la esquina tiene la puerta azul"
print(texto.count("la")) # 4
print(texto.count("z")) # 1
```

Úsalo para contar palabras en un texto, correos de cierto dominio, o apariciones de una secuencia. Dato curioso: no cuenta traslapes; en `"aaaa"`, `count("aa")` da 2, no 3.

Validar el contenido: isdigit, isalpha, isalnum y familia

Estos métodos responden preguntas de "¿este texto está formado solo por...?" con `True` o `False`. Son los porteros de la validación de formularios:

```
print("12345".isdigit()) # True (solo dígitos)
print("12a45".isdigit()) # False (hay una letra)
print("hola".isalpha())  # True (solo letras)
print("hola123".isalpha()) # False (hay números)
print("hola123".isalnum()) # True (letras Y números, sin símbolos)
print("hola 123".isalnum()) # False (el espacio no es alfanumérico)
```

`isdigit()` valida que un campo numérico sea realmente un número ANTES de `int()`: sin esa verificación, `int("veinte")` truena con `ValueError`. `isalpha()` valida nombres (

"María".isalpha() da True gracias a Unicode, pero "María José" da False por el espacio). isalnum() es el validador clásico de usuarios y contraseñas simples: letras y números sí, símbolos y espacios no. También existen isspace(), istitle(), isupper() e islower() para cuando los necesites.

Rellenar y alinear: zfill, center, ljust, rjust

Para cerrar la caja de herramientas, los métodos de formato visual. zfill(n) rellena el string con CEROS a la izquierda hasta que mida n caracteres:

```
print("42".zfill(5))    # 00042
print("7".zfill(3))     # 007
```

Es el estándar para folios y códigos: 00042 se ve profesional, 42 suelto parece improvisado.

center(n) centra en un ancho de n, ljust(n) alinea a la izquierda y rjust(n) a la derecha:

```
print("TÍTULO".center(20)) # '      TÍTULO      '
print("Ana".ljust(10) + "25") # 'Ana      25'
print("Ana".rjust(10) + "25") # '      Ana25'
```

Puedes cambiar el caracter de relleno: "42".center(10, "-") da "----42----". Son la base para tablas y reportes en terminal, como verás en los ejemplos.

f-strings con formato: números bonitos sin esfuerzo

Ya conoces los f-strings básicos (f"Hola {nombre}"), pero dentro de las llaves puedes agregar un FORMATO después de dos puntos para controlar exactamente cómo se ve el valor. Esto es lo que separa un reporte amateur de uno profesional.

El formato: .2f muestra un número flotante con 2 decimales (la f es de "float", el .2 dice cuántos decimales):

```
precio = 85.5
print(f"Total: ${precio:.2f}") # Total: $85.50
```

Sin el formato verías \$85.5, que en un ticket de compra se ve mal. Con: .2f siempre ves dos decimales, y además REDONDEA: f"{3.14159:.2f}" da "3.14". Para dinero, usa siempre :.2f. Es la norma.

El formato: `,.2f` agrega SEPARADOR DE MILES (la coma antes del punto):

```
sueldo = 15500.5
print(f"Sueldo: ${sueldo:,.2f}") # Sueldo: $15,500.50
```

En México usamos coma para miles y punto para decimales, justo lo que hace este formato por defecto.

El formato: `.0%` convierte un decimal en PORCENTAJE (lo multiplica por 100, le pone el símbolo%, y el `.0` dice cero decimales):

```
avance = 0.756
print(f"Avance: {avance:.0%}") # Avance: 76%
print(f"Avance: {avance:.1%}") # Avance: 75.6%
```

Y la ALINEACIÓN dentro de f-strings: después de los dos puntos puedes indicar `<` (izquierda), `>` (derecha) o `^` (centrado), junto con el ancho total:

```
nombre = "Ana"
print(f"|{nombre:<10}|") # |Ana          | (izquierda, ancho 10)
print(f"|{nombre:>10}|") # |          Ana| (derecha, ancho 10)
print(f"|{nombre:^10}|") # |   Ana      | (centrado, ancho 10)
```

Hace lo mismo que `ljust` / `rjust` / `center`, pero DENTRO del f-string, ideal para reportes. Puedes combinar: `f"{precio:>10.2f}"` alinea a la derecha con 2 decimales, y `f"{folio:0>5}"` rellena con ceros (equivale a `zfill(5)`).

Errores comunes de la unidad (y cómo evitarlos)

Vamos a reunir los tropiezos típicos de esta unidad en un solo lugar, porque conocerlos de antemano te ahorra horas de frustración.

1. KeyError por pedir una clave que no existe. El clásico. Sucede por errores de dedo ("Nomre" en lugar de "Nombre"), por mayúsculas ("Nombre" vs "nombre"), o por asumir que un dato externo trae un campo que no trae. Defensa: usa `get()` con valor por defecto cuando el dato sea incierto, o verifica con `in` antes de acceder. Y cuando te salga un `KeyError`, lo primero que revisas es la ortografía exacta de la clave.

****2. Confundir {} vacío con un set vacío.**** {} crea un DICCIONARIO vacío, no un set. Si haces `mi_set = {}` y luego `mi_set.add("algo")`, Python te lanza `AttributeError` porque

los diccionarios no tienen `add`. El set vacío SIEMPRE se crea con `set()`. Este error es tan común que aparece en exámenes de certificación.

****3. Olvidar `get()` y que el programa truee con datos reales.**** En tus pruebas todo funciona porque TÚ llenaste el diccionario completo. Pero cuando el programa lo usa alguien más, o lee un archivo con registros incompletos, falta una clave y ¡pum!, `KeyError` en producción. Acostúmbrate a programar a la defensiva: si la clave puede faltar, `get()` con un valor por defecto sensato.

4. Asumir que un set tiene orden. Los sets no garantizan orden. Si conviertes una lista a set y de vuelta a lista para quitar duplicados, el orden original se pierde. Si tu programa depende del orden, usa el truco de `dict.fromkeys()` o un ciclo manual. Y nunca, NUNCA intentes `mi_set[0]`: los sets no aceptan índices.

5. Llamar a un método de string sin guardar el resultado. `texto.upper()` sin asignación no cambia `texto`. Los strings son inmutables: el método devuelve un string NUEVO que debes guardar. Este error es silencioso (no truena, simplemente "no pasa nada") y por eso es traicionero. Cada vez que transformes texto, pregúntate: "¿dónde guardé el resultado?".

****6. Escribir `join()` al revés.**** Es " ".`join(lista)`, NO `lista.join(" ")`. El separador es el string sobre el que llamas el método; la lista va adentro. Si te confundes, recuerda: el pegamento va primero.

7. Comparar texto sin normalizar. `if ciudad == "México":` falla si el usuario escribió "méxico" o "MÉXICO"(con espacio al final). La combinación defensiva estándar es `entrada.strip().lower()` antes de comparar: quita espacios de los bordes y pasa a minúsculas. Dos métodos encadenados que te salvan de la mayoría de los problemas con entrada de usuario.

8. Modificar un diccionario mientras lo recorres. Si dentro de un `for clave in mi_dict:` agregas o eliminas claves del MISMO diccionario, Python te lanza `RuntimeError` (o se comporta de forma extraña). Si necesitas filtrar un diccionario, construye uno NUEVO en el ciclo, o itera sobre una copia: `for clave in list(mi_dict.keys()):`. Modificar VALORES de claves existentes sí está permitido; lo prohibido es agregar o quitar claves a mitad del recorrido.

Buenas prácticas: el código de quien ya le sabe

Nombra tus claves como si fueran a leerse en voz alta. "nombre", "fecha_nacimiento" se entienden solos; evita abreviaturas crípticas como "nom". Usa snake_case, igual que las variables.

****Usa get() con un valor por defecto que tenga sentido para tu lógica.**** No pongas `get("edad", 0)` si una edad de 0 confunde a tu programa; a veces `None` es mejor porque significa claramente "no hay dato". El valor por defecto debe representar honestamente la ausencia del dato en TU contexto.

****Para contadores, domina el patrón `d[k] = d.get(k, 0) + 1`**** Contar frecuencias (palabras en un texto, votos por candidato, productos por categoría) es una de las tareas más comunes en programación, y este patrón de una línea es la forma idiomática de hacerlo en Python. Cuando lo veas en código ajeno, ya sabes qué hace; cuando lo necesites, ya sabes cómo escribirlo.

Elige la estructura según la pregunta que vas a hacerle a los datos. ¿Vas a buscar por nombre o código? Diccionario. ¿Vas a verificar muchas veces si algo pertenece a un grupo? Set. ¿El orden importa o hay duplicados legítimos? Lista. Esta decisión, tomada al inicio, te ahorra reescrituras dolorosas después.

Limpia la entrada del usuario en cuanto la recibes. `dato = input("...").strip()` debería ser un reflejo. Y si vas a comparar, agrega `.lower()`. Dos segundos de limpieza al inicio evitan diez minutos de depuración al final buscando por qué "Juan" no es igual a "Juan".

Documenta la forma de tus diccionarios con un ejemplo. Cuando una función recibe o devuelve un diccionario con cierta estructura, deja un comentario con un ejemplo de cómo se ve: `# alumno = {"nombre": str, "edad": int, "materias": [str]}`. Tu yo del futuro (y tus compañeros) te lo van a agradecer, porque a diferencia de una clase, un diccionario no dice por sí solo qué campos debe tener.

Ejemplos paso a paso

Ejemplo 1: Agenda de contactos con diccionario

Vamos a construir una agendita donde cada contacto se busca por su nombre. Es el caso de uso perfecto para un diccionario: la clave es el nombre (lo que buscas) y el valor es otro diccionario con los datos (lo que encuentras).

```

# 1. Creamos la agenda vacía: un diccionario donde cada clave será un nombre
agenda = {}

# 2. Agregamos contactos. La clave es el nombre; el valor, otro diccionario
#     con los datos de esa persona (diccionario anidado)
agenda["Ana Martínez"] = {
    "telefono": "555-111-2233",
    "correo": "ana.martinez@correo.com",
    "ciudad": "Guadalajara"
}
agenda["Luis Torres"] = {
    "telefono": "555-444-5566",
    "correo": "luis.torres@correo.com",
    "ciudad": "Monterrey"
}

# 3. Buscar un contacto por su nombre: acceso directo por clave
nombre = "Ana Martínez"
if nombre in agenda:
    datos = agenda[nombre]
    print(f"Teléfono de {nombre}: {datos['telefono']}")
    print(f"Correo: {datos['correo']}")
else:
    print("Contacto no encontrado")

# 4. Buscar con get() para no tronar si no existe
datos = agenda.get("Pedro Sánchez")
if datos is None:
    print("Pedro Sánchez no está en la agenda")

# 5. Actualizar un dato existente (el diccionario es mutable)
agenda["Ana Martínez"]["telefono"] = "555-999-0000"

# 6. Eliminar un contacto con pop() (nos devuelve sus datos por si los necesitamos)
eliminado = agenda.pop("Luis Torres", None)
print(f"Se eliminó a Luis Torres: {eliminado is not None}")

# 7. Mostrar TODA la agenda recorriendo con items()
print("\n--- AGENDA COMPLETA ---")
for nombre, datos in agenda.items():
    print(f"{nombre}: {datos['telefono']} | {datos['correo']} | {datos['ciudad']}")

```

Explicación línea por línea: en el paso 1 creamos el diccionario vacío que será nuestra agenda. En el paso 2 agregamos dos contactos; fíjate que cada valor es a su vez un diccionario con `telefono`, `correo` y `ciudad`: es el patrón de "registro etiquetado" que representa perfecto a una persona. En el paso 3 buscamos con `in` antes de acceder, la forma defensiva de usar corchetes. En el paso 4 usamos `get()` y comparamos con `None` para detectar la ausencia sin errores. El paso 5 muestra cómo se modifica un valor anidado encadenando corchetes: `agenda["Ana Martínez"]["telefono"]`. El paso 6 usa `pop()` con valor por defecto `None` para eliminar sin riesgo. Y el paso 7 recorre todo con `for nombre, datos in agenda.items()`, el patrón estándar. Este programita ya es, en miniatura, como funcionan las agendas y directorios reales: búsqueda por nombre, datos etiquetados, todo modificable.

Ejemplo 2: Inventario de tienda con control de stock

Ahora un inventario donde la clave es el código del producto. Incluye vender (restar stock), alertas de stock bajo y reporte con valores totales. Aquí el diccionario brilla porque cada producto tiene varios campos con nombre.

```

# 1. Inventario inicial: clave = código, valor = diccionario con datos
inventario = {
    "TORT-001": {"nombre": "Tortillero", "precio": 85.50, "stock": 40},
    "MOLC-002": {"nombre": "Molcajete", "precio": 320.00, "stock": 12},
    "CAZ-003": {"nombre": "Cazuela de barro", "precio": 210.00, "stock": 5},
}

# 2. Función para vender: resta stock si hay suficiente
def vender(codigo, cantidad):
    producto = inventario.get(codigo) # get() por si el código no existe
    if producto is None:
        return f"Error: el código {codigo} no existe"
    if producto["stock"] < cantidad:
        return f"Error: solo hay {producto['stock']} piezas de {producto['nombre']}"
    producto["stock"] -= cantidad # modificamos el valor anidado directamente
    total = producto["precio"] * cantidad
    return f"Venta: {cantidad} x {producto['nombre']} = ${total:.2f}"

print(vender("MOLC-002", 3)) # Venta: 3 x Molcajete = $960.00
print(vender("CAZ-003", 10)) # Error: solo hay 5 piezas de Cazuela de barro
print(vender("XXX-999", 1)) # Error: el código XXX-999 no existe

# 3. Alerta de stock bajo: recorrer con items() y filtrar
print("\n-- PRODUCTOS CON STOCK BAJO (menos de 10) ---")
for codigo, datos in inventario.items():
    if datos["stock"] < 10:
        print(f"{codigo}: {datos['nombre']} -> quedan {datos['stock']}")

# 4. Valor total del inventario: precio x stock de cada producto, sumando
valor_total = 0
for datos in inventario.values(): # solo nos interesan los valores
    valor_total += datos["precio"] * datos["stock"]
print(f"\nValor total del inventario: ${valor_total:.2f}")

# 5. Resurtir: agregar stock a un producto existente
inventario["CAZ-003"]["stock"] += 20
print(f"Nuevo stock de cazuelas: {inventario['CAZ-003']['stock']}")

```

Explicación: el paso 1 arma el inventario con códigos como claves, el identificador natural de un producto (como el SKU de una tienda real). El paso 2 es una función `vender` que muestra tres defensas profesionales: `get()` para código inexistente, verificación de stock antes de restar, y f-string con: `.2f` para el dinero. Fíjate que `producto["stock"] -= cantidad`

modifica el diccionario ORIGINAL, porque `producto` es una referencia al diccionario anidado, no una copia: esto es clave para entender cómo trabajan los diccionarios anidados. El paso 3 filtra con un `if` dentro del `for...items()`, el patrón de "reporte con condición". El paso 4 usa `values()` porque solo necesita los datos, no los códigos, y acumula el valor total con formato de miles. El paso 5 muestra lo fácil que es resurtir. Este ejemplo es un mini sistema de punto de venta: con esto ya entiendes la lógica que hay detrás de cualquier software de inventarios.

Ejemplo 3: Eliminar duplicados de una lista con set (y contar únicos)

El truco estrella de los sets en acción, con un caso real: una lista de asistentes a un evento donde varios se registraron dos veces por error. Queremos la lista limpia y saber cuántas personas únicas hay.

```

# 1. Lista con duplicados (como llegan los datos reales: sucios)
asistentes = [
    "Ana", "Luis", "Pedro", "Ana", "María",
    "Luis", "Sofía", "Ana", "Pedro", "Jorge"
]
print(f"Registros totales: {len(asistentes)}") # 10

# 2. El truco: convertir a set (desaparecen duplicados) y de vuelta a lista
unicos = list(set(asistentes))
print(f"Personas únicas: {len(unicos)}") # 6
print(unicos) # orden no garantizado, ej: ['Sofía', 'Pedro', 'Ana', ...]

# 3. Si el orden SÍ importa: truco con dict.fromkeys (conserva el primero)
unicos_ordenados = list(dict.fromkeys(asistentes))
print(unicos_ordenados)
# ['Ana', 'Luis', 'Pedro', 'María', 'Sofía', 'Jorge'] (orden original)

# 4. ¿Quiénes se registraron más de una vez? (los repetidos)
vistos = set()
repetidos = set()
for nombre in asistentes:
    if nombre in vistos:
        repetidos.add(nombre) # ya lo habíamos visto: es repetido
    else:
        vistos.add(nombre) # primera vez: lo marcamos como visto
print(f"Se registraron más de una vez: {repetidos}")

# 5. Comparar dos listas: ¿quiénes vinieron a ambos eventos?
evento_a = {"Ana", "Luis", "Pedro", "María"}
evento_b = {"Luis", "María", "Sofía", "Jorge"}
print(f"Vinieron a los dos: {evento_a & evento_b}")
print(f"Solo al evento A: {evento_a - evento_b}")
print(f"Asistentes únicos totales: {len(evento_a | evento_b)}")

```

Explicación: el paso 1 presenta datos realistas y sucios, como llegan siempre. El paso 2 es el truco de una línea `list(set(...))` con la advertencia honesta del orden. El paso 3 enseña la alternativa `dict.fromkeys()` para cuando el orden importa: aprovecha que los diccionarios eliminan claves repetidas pero conservan el orden de inserción, y luego lo convertimos a lista. Es un truco elegante que combina ambos temas de la unidad. El paso 4 muestra un algoritmo clásico con dos sets: `vistos` para lo ya encontrado y `repetidos` para los duplicados, usando la membresía ultrarrápida de `in` sobre sets. El paso 5 aplica unión, intersección y diferencia a un caso de negocio real (comparar asistencia de dos eventos). Con estos cinco pasos ya manejas la limpieza y comparación de datos como un profesional.

Ejemplo 4: Analizador de texto - las palabras más comunes

Este es EL ejemplo donde los tres temas de la unidad trabajan juntos: métodos de strings para limpiar y partir el texto, y un diccionario para contar. Es, en miniatura, como funcionan los analizadores de redes sociales y los conteos de palabras de los procesadores de texto.

```

# 1. El texto a analizar (puede venir de un archivo, un comentario, un tuit)
texto = """
El café de la mañana es sagrado. El café despierta, el café acompaña,
y sin el café la mañana no es mañana. ¡Que viva el café!
"""

# 2. Normalizar: minúsculas para que "El" y "el" cuenten como la misma palabra
texto = texto.lower()

# 3. Quitar signos de puntuación con replace encadenados
for signo in ". , ! ? ; : "':
    texto = texto.replace(signo, "")

# 4. Partir en palabras con split() (maneja los saltos de línea y espacios de más)
palabras = texto.split()
print(f"Total de palabras: {len(palabras)}")

# 5. Contar con el patrón estrella: dict + get()
conteo = {}
for palabra in palabras:
    conteo[palabra] = conteo.get(palabra, 0) + 1

# 6. Mostrar el conteo completo ordenado de mayor a menor
print("\n--- CONTEO DE PALABRAS ---")
for palabra, veces in sorted(conteo.items(), key=lambda par: par[1], reverse=True):
    print(f"{palabra:<12} {veces}")

# 7. Las 3 palabras más comunes
top3 = sorted(conteo.items(), key=lambda par: par[1], reverse=True)[:3]
print("\nTop 3:", [p for p, v in top3])

# 8. Palabras únicas con set: ¿cuántas palabras DISTINTAS hay?
print(f"Palabras distintas: {len(set(palabras))}")

```

Explicación línea por línea: el paso 2 normaliza con `lower()` para no contar "El" y "el" por separado, la decisión más importante del análisis. El paso 3 limpia la puntuación recorriendo una cadena de signos y reemplazando cada uno por nada; es un ciclo sobre un string (que también se puede recorrer letra por letra). El paso 4 usa `split()` sin argumentos, que maneja solito los saltos de línea `\n` y los espacios múltiples del texto original. El paso 5 es el patrón de conteo `conteo[palabra] = conteo.get(palabra, 0) + 1`: la primera vez que ve una palabra, `get` devuelve 0 y guarda 1; las siguientes veces suma uno al valor existente. El paso 6 ordena los pares por el contador usando `sorted()` con una función `lambda` que dice

"ordena por el segundo elemento de cada par" (par[1]), de mayor a menor. El paso 7 toma los primeros 3 con rebanado [:3]. El paso 8 usa set() para contar palabras distintas en una línea. Este programa es la semilla de cosas grandes: análisis de sentimientos, detección de temas en tendencia, estadísticas de texto. Y todo con herramientas que ya dominas.

Ejemplo 5: Validador de correo electrónico simple con métodos de string

Vamos a validar que un correo tenga forma válida usando solo métodos de strings, sin expresiones regulares (que son otro tema). Es un validador "simple pero honesto": no garantiza que el correo exista, pero sí que tenga la forma correcta.

```
def validar_correo(correo):
    # 1. Limpiar espacios de los bordes (el usuario siempre mete espacios de más)
    correo = correo.strip()

    # 2. No puede estar vacío después de limpiar
    if correo == "":
        return False, "El correo no puede estar vacío"

    # 3. No puede tener espacios EN MEDIO
    if " " in correo:
        return False, "El correo no puede tener espacios"

    # 4. Debe tener exactamente UNA arroba (find para la primera, rfind para la última)
    if correo.count("@") != 1:
        return False, "Debe tener exactamente una @"

    # 5. Partir en usuario y dominio por la arroba
    usuario, dominio = correo.split("@")

    # 6. El usuario no puede estar vacío
    if usuario == "":
        return False, "Falta el nombre de usuario antes de la @"

    # 7. El dominio debe tener al menos un punto (y no al inicio ni al final)
    if "." not in dominio:
        return False, "El dominio debe tener un punto (ej: gmail.com)"
    if dominio.startswith(".") or dominio.endswith("."):
        return False, "El punto del dominio está mal colocado"

    # 8. La extensión (después del último punto) debe tener al menos 2 letras
    extension = dominio.rsplit(".", 1)[1]
```

```

    if len(extension) < 2 or not extension.isalpha():
        return False, "La extensión del dominio no es válida"

    # 9. Todo en minúsculas para normalizar (los correos no distinguen mayúsculas)
    return True, correo.lower()

# Probamos con varios casos, buenos y malos
pruebas = [
    " Juan.Perez@Gmail.COM ",
    "ana@correo",
    "sin-arroba.com",
    "doble@@arroba.com",
    "@sin-usuario.com",
    "usuario@dominio.c",
    "maria lopez@mail.com",
    "pedro@empresa.com.mx",
]

for p in pruebas:
    valido, mensaje = validar_correo(p)
    estado = "VÁLIDO " if valido else "INVÁLIDO"
    print(f"{estado} | {p!r:35} -> {mensaje}")

```

Explicación: la función devuelve una tupla (valido, mensaje) para informar tanto el resultado como la razón, un patrón muy profesional. El paso 1 limpia con `strip()` (reflejo obligatorio). Los pasos 2-4 son validaciones de forma con comparaciones simples, `in` y `count("@")`. El paso 5 usa `split("@")` con desempaqueado en dos variables, posible porque ya verificamos que hay exactamente una arroba. El paso 7 usa `startswith / endswith` para la colocación del punto. El paso 8 usa `rsplit(".", 1)` para obtener la extensión correcta incluso en dominios como `empresa.com.mx` (con `split` normal habríamos tomado solo `com`). El paso 9 normaliza a minúsculas. En las pruebas, fíjate cómo `{p!r:35}` en el f-string muestra el valor con comillas (`!r` = repr) alineado a 35 espacios: un truco de formato para que la tabla de pruebas se vea ordenada. Este validador no es perfecto (los correos reales tienen reglas más complejas), pero para un formulario básico es más que suficiente y te enseña a encadenar validaciones de forma legible.

Ejemplo 6: Tabla formateada con f-strings y alineación

Vamos a generar un reporte de calificaciones con columnas perfectamente alineadas, como los reportes de la terminal de un sistema escolar. Aquí brillan los f-strings con formato y la alineación.

```

# 1. Los datos: lista de diccionarios (la estructura más común del mundo real)
alumnos = [
    {"nombre": "Ana Martínez", "matricula": "A001", "promedio": 9.2},
    {"nombre": "Luis Torres", "matricula": "A002", "promedio": 7.85},
    {"nombre": "María José Hernández", "matricula": "A003", "promedio": 10.0},
    {"nombre": "Pedro", "matricula": "A004", "promedio": 6.4},
]

# 2. Encabezado centrado y línea separadora
print("REPORTE DE CALIFICACIONES".center(52))
print("=" * 52)

# 3. Títulos de columna con alineación fija usando f-strings
print(f"{'MATRÍCULA':<10} {'NOMBRE':<25} {'PROMEDIO':>8} {'ESTADO':<8}")
print("-" * 52)

# 4. Una fila por alumno, con los mismos anchos para que todo cuadre
for a in alumnos:
    estado = "APROBADO" if a["promedio"] >= 7 else "REPROBADO"
    print(f"{a['matricula']:<10} {a['nombre']:<25} {a['promedio']:>8.2f} {estado:<8}")

print("-" * 52)

# 5. Fila de resumen: promedio general con formato de 2 decimales
promedios = [a["promedio"] for a in alumnos]
general = sum(promedios) / len(promedios)
print(f"{'PROMEDIO GENERAL':<36} {general:>8.2f}")

# 6. El mejor promedio, con porcentaje de avance
mejor = max(alumnos, key=lambda a: a["promedio"])
print(f"Mejor promedio: {mejor['nombre']} ({mejor['promedio']:.1f} "
      f"= {mejor['promedio']/10:.0%} de avance)")

```

Salida del programa:


```

# Paso 1: crear el diccionario con 3 mascotas (diccionarios anidados)
mascotas = {
    "Firulais": {"especie": "perro", "edad": 4, "dueño": "Don Beto"},
    "Mishi": {"especie": "gato", "edad": 2, "dueño": "Doña Cuca"},
    "Kiko": {"especie": "loro", "edad": 7, "dueño": "El profe"},
}

# Paso 2a: acceder a un valor anidado encadenando corchetes
print(mascotas["Mishi"]["edad"]) # 2

# Paso 2b: modificar un valor existente (misma sintaxis, con asignación)
mascotas["Firulais"]["edad"] = 5
print(mascotas["Firulais"]["edad"]) # 5 (ya cumplió años)

# Paso 2c: eliminar con pop() y guardar lo eliminado
datos_kiko = mascotas.pop("Kiko")
print(datos_kiko) # {'especie': 'loro', 'edad': 7, 'dueño': 'El profe'}

# Paso 2d: len() sobre un diccionario cuenta sus CLAVES (sus pares)
print(f"Mascotas registradas: {len(mascotas)}") # 2

```

Por qué funciona así: `len()` en un diccionario cuenta los pares clave-valor, igual que en una lista cuenta elementos. `pop()` devuelve el valor eliminado, así que `datos_kiko` conserva la ficha de Kiko por si la veterinaria la necesita después (para un historial, por ejemplo). Y la modificación del paso 2b no crea una clave nueva porque "edad" ya existía dentro del diccionario de Firulais: la sintaxis de corchetes con `=` modifica si existe y crea si no existe, en cada nivel por separado.

Ejercicio 2: Contador de votos con `get()`

En una votación escolar para elegir el color del uniforme, los votos llegan en una lista (algunos alumnos votaron dos veces por error, pero cada voto cuenta). Usa un diccionario para contar cuántos votos recibió cada color con el patrón `get()`, e imprime el color ganador.

```

# Paso 1: los votos tal como llegan (con repetidos, como datos reales)
votos = ["azul", "rojo", "azul", "verde", "azul", "rojo", "azul", "verde", "rojo",
"azul"]

# Paso 2: contar con el patrón get() -> valor anterior (o 0) + 1
conteo = {}
for color in votos:
    conteo[color] = conteo.get(color, 0) + 1
print(conteo) # {'azul': 5, 'rojo': 3, 'verde': 2}

# Paso 3: encontrar al ganador recorriendo con items()
ganador = ""
max_votos = 0
for color, cantidad in conteo.items():
    if cantidad > max_votos:
        max_votos = cantidad
        ganador = color
print(f"Ganador: {ganador} con {max_votos} votos") # Ganador: azul con 5 votos

```

Por qué funciona así: en la primera vuelta del ciclo, conteo está vacío, así que `conteo.get("azul", 0)` devuelve 0 y guarda $0 + 1 = 1$. En la segunda vez que aparece "azul", `get` devuelve 1 y guarda 2. Sin `get()` habríamos necesitado un `if color in conteo`: para distinguir la primera vez de las siguientes; el `get()` con valor por defecto colapsa esas dos ramas en una sola línea. El paso 3 es el algoritmo de "máximo" aplicado a un diccionario: llevamos el mejor candidato en dos variables y lo actualizamos cuando encontramos algo mayor. Este mismo esqueleto (contar con `get` + buscar el máximo con `items`) resuelve conteo de palabras, productos más vendidos, letras más frecuentes y decenas de problemas reales.

Ejercicio 3: Limpieza de lista de correos con set y validación

Tienes una lista de correos registrados en un formulario web: vienen con duplicados, con mayúsculas mezcladas y con espacios de más. Debes: (a) normalizar cada correo (quitar espacios y pasar a minúsculas), (b) eliminar duplicados con set, (c) quedarte solo con los que tengan forma válida (contengan "@" y terminen en ".com" o ".mx"), usando un set para la verificación.

```

# Paso 1: datos sucios como llegan de un formulario real
crudos = [
    " Ana@Gmail.COM ",
    "luis@correo.com",
    "ANA@gmail.com",
    "pedro@empresa.mx",
    "luis@correo.com",
    "sin-arroba.com",
    " ana@GMAIL.com",
    "maria@tienda.org",
]

# Paso 2a: normalizar cada correo con strip() + lower() (encadenados)
normalizados = []
for c in crudos:
    normalizados.append(c.strip().lower())
print(normalizados)
# ['ana@gmail.com', 'luis@correo.com', 'ana@gmail.com', 'pedro@empresa.mx',
#  'luis@correo.com', 'sin-arroba.com', 'ana@gmail.com', 'maria@tienda.org']

# Paso 2b: eliminar duplicados con el truco list(set(...))
unicos = list(set(normalizados))
print(f"Únicos: {len(unicos)}") # 5

# Paso 2c: filtrar los válidos; set de terminaciones permitidas para el in
terminaciones = {".com", ".mx"}
validos = []
for correo in unicos:
    if "@" in correo and correo.endswith(tuple(terminaciones)):
        validos.append(correo)
print(validos)
# ['ana@gmail.com', 'luis@correo.com', 'pedro@empresa.mx']

```

Por qué funciona así: el paso 2a es crítico y va ANTES de eliminar duplicados, porque "Ana@Gmail.COM" y "ana@gmail.com" son el mismo correo pero se ven distintos para la computadora; si quitas duplicados sin normalizar primero, los duplicados disfrazados sobrevivirían. El orden correcto siempre es: normalizar primero, deduplicar después. En el paso 2c, `endswith()` acepta una tupla de opciones (por eso convertimos el set con `tuple()`), y devuelve True si el correo termina con CUALQUIERA de ellas; así "maria@tienda.org" queda fuera porque .org no está permitido. El `in` para la arroba es la verificación mínima de forma. Fíjate que combinamos los tres temas: métodos de strings para limpiar, set para deduplicar y para las terminaciones permitidas. Este pipeline (limpiar -> normalizar ->

deduplicar -> validar) es exactamente lo que hacen los sistemas reales antes de guardar datos de usuarios.

Ejercicio 4: Promedio de calificaciones con values() y f-strings

Un profesor guarda las calificaciones de un alumno en un diccionario donde la clave es la materia y el valor la calificación. Debes: (a) calcular el promedio usando values(), (b) imprimir un reporte donde cada materia salga alineada a la izquierda en 15 espacios y la calificación con 1 decimal, (c) agregar una materia nueva, (d) mostrar solo los nombres de las materias con keys().

```
# Paso 1: diccionario materia -> calificación
califs = {"Matemáticas": 9.2, "Español": 8.5, "Historia": 7.8, "Física": 9.0}

# Paso 2a: promedio con values() (no necesitamos las claves para sumar)
suma = sum(califs.values())
promedio = suma / len(califs)
print(f"Promedio: {promedio:.1f}") # Promedio: 8.6

# Paso 2b: reporte alineado recorriendo con items()
print("--- BOLETA ---")
for materia, cal in califs.items():
    print(f"{materia:<15} {cal:.1f}")

# Paso 2c: agregar una materia (clave nueva = se crea el par)
califs["Química"] = 8.9

# Paso 2d: mostrar solo las materias con keys()
print("Materias cursadas:", list(califs.keys()))
# ['Matemáticas', 'Español', 'Historia', 'Física', 'Química']
```


Por qué funciona así: `sum(califs.values())` suma directamente los valores sin pasar por las claves, más corto y claro que un ciclo manual. En el reporte, `{materia:<15}` alinea los nombres a la izquierda en 15 espacios para que las calificaciones queden en columna aunque los nombres midan distinto; `{cal:.1f}` deja un decimal como en las boletas reales. El paso 2c demuestra otra vez la doble naturaleza de la asignación con corchetes: como "Química" no existía, se crea el par. Y `list(califs.keys())` convierte la vista de claves en una lista imprimible. Nota que el promedio del paso 2a se calculó ANTES de agregar Química; si el profesor quisiera el promedio actualizado, tendría que recalcularlo después del paso 2c (los valores ya cambiaron). Ese detalle -recalcular después de modificar- es un error típico en programas reales: promedios y totales calculados con datos viejos.

Ejercicio 5: Encuesta de gustos con operaciones de conjuntos

En un salón se preguntó qué redes sociales usan los alumnos. Tienes el set de quienes usan TikTok y el set de quienes usan Instagram. Debes: (a) encontrar quiénes usan ambas, (b) quiénes usan al menos una, (c) quiénes usan solo TikTok, (d) verificar con `in` si un alumno específico usa Instagram, (e) agregar a un alumno nuevo al set de TikTok.

```

# Paso 1: los dos conjuntos de la encuesta
tiktok = {"Ana", "Luis", "Pedro", "María", "Jorge"}
instagram = {"Luis", "María", "Sofía", "Ana", "Carmen"}

# Paso 2a: intersección -> usan AMBAS
ambas = tiktok & instagram
print(f"Usan ambas: {ambas}") # {'Ana', 'Luis', 'María'}

# Paso 2b: unión -> usan AL MENOS UNA
alguna = tiktok | instagram
print(f"Usan al menos una: {len(alguna)} alumnos") # 7

# Paso 2c: diferencia -> SOLO TikTok (en tiktok pero no en instagram)
solo_tiktok = tiktok - instagram
print(f"Solo TikTok: {solo_tiktok}") # {'Pedro', 'Jorge'}

# Paso 2d: membresía con in (rapidísima en sets)
alumno = "Sofía"
if alumno in instagram:
    print(f"{alumno} sí usa Instagram")

# Paso 2e: agregar un alumno nuevo con add()
tiktok.add("Carmen")
print(f"TikTok ahora tiene {len(tiktok)} usuarios") # 6

```

Por qué funciona así: cada operador responde una pregunta de negocio distinta con una sola línea. La intersección `&` responde "los de ambos grupos" (útil para promociones cruzadas: a estos les puedes anunciar en las dos redes). La unión `|` responde "el alcance total sin duplicados" (¿a cuántas personas distintas llego?). La diferencia `-` responde "los exclusivos de un grupo" (¿a quiénes solo les llego por TikTok?). El orden en la diferencia importa: `tiktok - instagram` no es lo mismo que `instagram - tiktok`; siempre lee "los del primero que NO están en el segundo". Y el `in` del paso 2d sobre un set es la verificación de membresía más rápida que existe en Python, ideal cuando la encuesta tuviera miles de alumnos. Si intentarás estas tres operaciones con listas, necesitarías ciclos y condicionales; con sets son tres líneas.

Ejercicio 6: Ticket de compra con split, replace y formato

Un sistema guarda cada venta como un string con formato "producto|cantidad|precio". Tienes una lista de esas ventas. Debes: (a) partir cada venta con `split("|")`, (b) limpiar el nombre del producto (viene con guiones bajos en lugar de espacios: usa `replace`), (c)

calcular el subtotal de cada línea, (d) imprimir un ticket alineado con f-strings, (e) mostrar el total con formato de dinero.

```
# Paso 1: las ventas como strings con separador | (como líneas de un archivo)
ventas = [
    "tortillero|2|85.50",
    "molcajete_grande|1|320.00",
    "cazuela_de_barro|3|210.00",
]

# Paso 2: encabezado del ticket
print("*** MI TIENDITA ***".center(40))
print("-" * 40)
print(f"{'PRODUCTO':<20} {'CANT':>4} {'PRECIO':>8} {'SUBTOTAL':>8}")
print("-" * 40)

# Paso 3: procesar cada venta
total = 0
for v in ventas:
    # Paso 3a: partir por el separador |
    partes = v.split("|")
    nombre_crudo, cantidad_txt, precio_txt = partes

    # Paso 3b: limpiar el nombre (guiones bajos -> espacios) y capitalizar
    nombre = nombre_crudo.replace("_", " ").title()

    # Paso 3c: convertir cantidad y precio a números
    cantidad = int(cantidad_txt)
    precio = float(precio_txt)
    subtotal = cantidad * precio
    total += subtotal

    # Paso 3d: imprimir la línea alineada
    print(f"{'nombre':<20} {'cantidad':>4} ${precio:>7.2f} ${subtotal:>7.2f}")

print("-" * 40)
# Paso 4: total con formato de dinero (2 decimales + separador de miles)
print(f"{'TOTAL':<20} {'':>4} {'':>8} ${total:>7,.2f}")
```

Salida del programa:

*** MI TIENDITA ***			
PRODUCTO	CANT	PRECIO	SUBTOTAL
Tortillero	2	\$ 85.50	\$ 171.00
Molcajete Grande	1	\$ 320.00	\$ 320.00
Cazuela De Barro	3	\$ 210.00	\$ 630.00

TOTAL:			\$1,121.00

Por qué funciona así: el paso 3a usa `split("|")` porque los datos vienen con ese separador (muy común en archivos de texto cuando los datos contienen comas). El desempaqueado `nombre_crudo, cantidad_txt, precio_txt = partes` funciona porque sabemos que siempre hay exactamente 3 campos; si una línea viniera incompleta, esta línea tronaría, y en un sistema real la protegerías verificando `len(partes)`. El paso 3b encadena `replace()` y `title()` para convertir `molcajete_grande` en `Molcajete Grande`, presentable para el cliente. El paso 3c convierte los textos a números porque `split()` siempre devuelve strings y no se puede multiplicar texto por texto. En el formato, fíjate en `${precio:>7.2f}`: el `$` va fuera de las llaves como texto literal, y dentro va alineación derecha + 2 decimales. El total usa `>7, .2f` con la coma de miles: `$1,121.00` se lee mucho mejor que `$1121.0`. Este ejercicio junta TODO lo de strings de la unidad en un programa útil de verdad: partir, limpiar, reemplazar, formatear.

Resumen de la unidad

Llegaste al final de la unidad 7, y si la trabajaste en serio, tu caja de herramientas de Python acaba de triplicarse. Vamos a recapitular lo esencial para que quede grabado.

Diccionarios: datos con etiqueta. Guardan pares `clave: valor` entre llaves para buscar cualquier dato POR SU NOMBRE, al instante. Crear: `{"nombre": "Ana", "edad": 22}`. Leer: `d["nombre"]`. Agregar o modificar: `d["clave"] = valor` (crea si no existe, sobrescribe si existe). Eliminar: `del d["clave"]` o `d.pop("clave")` (devuelve el valor). Ver por partes: `keys()`, `values()`, `items()`; recorrer completo: `for clave, valor in d.items():`. Defensa contra el `KeyError`: `get("clave", valor_por_defecto)`, obligatoria con datos inciertos (usuarios, archivos, APIs). Con `in` preguntas si una clave existe (busca en claves, no en valores). Los diccionarios anidados modelan registros complejos como un JSON, encadenando corchetes. Y la decisión de diseño: diccionario cuando cada dato tenga un nombre por el cual buscarlo; lista cuando el orden importe.

Sets: colecciones de únicos sin orden. Guardan elementos ÚNICOS (los duplicados desaparecen solos) y NO garantizan orden: nunca accedas por posición. El set vacío es `set()`, porque `{}` es un diccionario vacío (clásico error de examen). Agregar: `add()`; quitar: `remove()` (truena si no existe) o `discard()` (perdona si no existe). Operaciones: unión `|` (todo lo de ambos), intersección `&` (lo común), diferencia `-` (lo del primero que no está en el segundo). Superpoderes: deduplicar listas en una línea con `list(set(lista))`, y membresía con `in` a velocidad constante aunque haya millones de elementos.

Strings avanzados: texto moldeable. Los strings son inmutables: cada método devuelve un string NUEVO que debes guardar. Mayúsculas/minúsculas: `upper()`, `lower()` (normaliza antes de comparar), `title()`, `capitalize()`, `swapcase()`. Bordes: `strip()`, `lstrip()` y `rstrip()` (aceptan caracteres a recortar). Reemplazar: `replace(viejo, nuevo)`. Partir: `split()` y `rsplit()`. Pegar: `"sep".join(lista)` (el separador va primero). Inicio/fin: `startswith()`, `endswith()`. Buscar: `find()` (-1 si no hay) e `index()` (error si no hay). Contar: `count()`. Validar: `isdigit()`, `isalpha()`, `isalnum()`. Rellenar/alinear: `zfill()`, `center()`, `ljust()`, `rjust()`. Y f-strings con formato: `:.2f, :, .2f, :.0%`, y `< / > / ^` con ancho para columnas.

Los errores que ya no te van a sorprender: el `KeyError` por clave inexistente o mal escrita (defiéndete con `get()`); el `{}` que es dict y no set vacío; el método de string llamado sin guardar el resultado; el `join()` escrito al revés; la comparación de texto sin normalizar con `strip().lower()`; el set al que le pides orden o índice; y modificar las claves de un diccionario mientras lo recorres.

La idea grande que une todo: estos tres temas trabajan juntos en programas reales. El analizador de texto de los ejemplos lo demostró: `split()` parte el texto, `lower()` y `strip()` lo normalizan, el diccionario cuenta con `get()`, y el set mide lo único. Agenda de contactos, inventario de tienda, limpieza de correos, ticket de compra: todos los ejemplos de esta unidad son versiones en miniatura de sistemas reales. Ya no solo guardas datos: los etiquetas, los deduplicas y los presentas bonitos.

En la próxima unidad seguirás subiendo de nivel, pero lo que aprendiste aquí -diccionarios, sets y texto avanzado- te va a acompañar en CADA programa que escribas de aquí en adelante. Son las estructuras que más vas a usar en tu vida como programador. Prácticalas hasta que te salgan sin pensar: escribe tus propios ejemplos, rompe el código a propósito para ver los errores, y vuelve a esta unidad cada vez que dudes. Nos vemos en la que sigue.

UNIDAD 8: Funciones y código elegante: deja de repetirte y programa como profesional

Introducción

Llegaste a la unidad que separa a los que "escriben código que jala" de los que "programan de verdad". Hasta ahora ya sabes pedirle datos al usuario, guardarlos en variables, tomar decisiones con if, repetir tareas con ciclos y organizar información en listas y diccionarios. Con eso puedes resolver muchos problemas, pero hay un límite: si todo tu programa es una sola lista larguísima de instrucciones de arriba hacia abajo, en cuanto crezca se vuelve un monstruo imposible de entender, de corregir y de reutilizar. Es como una casa construida sin planos, ladrillo sobre ladrillo, donde cada arreglo implica tumbar una pared.

En esta unidad vas a aprender la herramienta que usan todos los programadores profesionales para domar ese monstruo: las funciones. Y al final, un regalo de elegancia: las list comprehensions, una forma de crear listas en una sola línea que te va a hacer sentir que hablas Python fluido.

Déjame ponerte la analogía que lo explica todo: la receta de cocina reutilizable.

Imagina que tienes una taquería. Todos los días preparas salsa roja: asas los jitomates, los mueles con ajo, cebolla y chiles, la sazonas, la hierves. Si cada vez que un cliente pide tacos tuvieras que recordar de memoria los 15 pasos de la salsa, volverías loco. Lo que hace cualquier taquero sensato es escribir la receta UNA vez en una tarjeta y pegarla en la pared. Después, cada vez que necesita salsa, no repite los 15 pasos de memoria: simplemente dice "haz la salsa" y sigue la tarjeta. La tarjeta es la función: un bloque de instrucciones con nombre, escrito una sola vez, que puedes "llamar" cuantas veces quieras. Y lo mejor: si un día decides que la salsa lleva más chile, no tienes que cambiar 50 lugares donde la preparaste; cambias la tarjeta UNA vez y todos los tacos del día salen con la mejora. Eso, en programación, tiene nombre: DRY, "Don't Repeat Yourself", no te repitas.

Aquí va la segunda analogía, igual de poderosa: los aparatos con botones.

Piensa en tu microondas. Tiene un botón que dice "palomitas". Cuando lo presionas, el microondas ejecuta una secuencia completa: cierta potencia, cierto tiempo, pausas calculadas. Tú no necesitas saber ni repetir esa secuencia; solo presionas el botón y confías en que hace su trabajo. Una función es exactamente eso: un botón con nombre. Tú defines qué hace por

dentro una sola vez, y después solo "presionas el botón" escribiendo su nombre. Además, los botones reciben "órdenes": el botón de "descongelar" te pide que le digas cuántos kilos de carne vas a descongelar. En programación eso se llama pasarle parámetros a la función: datos de entrada que cambian su comportamiento. Y el microondas también te "devuelve" algo: comida caliente. Las funciones devuelven resultados con la instrucción `return`.

¿Por qué esto cambia tu nivel como programador? Por tres razones enormes.

Primera: reutilización. Escribes una vez, usas cien veces. Si tu programa necesita calcular el IVA en veinte lugares distintos, no escribes la fórmula veinte veces: escribes una función `calcular_iva` y la llamas veinte veces. Menos código, menos errores.

Segunda: organización mental. Un programa grande dividido en funciones con nombres claros se lee como una historia: `pedir_datos()`, `validar_datos()`, `calcular_total()`, `mostrar_resultado()`. Cada función es un capítulo. Sin funciones, es un solo párrafo kilométrico donde te pierdes.

Tercera: corrección de errores. Cuando algo falla, no tienes que revisar 500 líneas: revisas la función que hace esa tarea. Y si la arreglas ahí, se arregla en todos los lugares donde se usa. Volviendo a la taquería: si la salsa sale salada, corriges la tarjeta, no cada taco.

En esta unidad vamos a construir este superpoder paso a paso. Primero, qué es una función y cómo se escribe con `def` (Clase 20): la sintaxis exacta, cómo llamarla por su nombre y cómo dejar de copiar y pegar código para siempre. Después, cómo las funciones reciben información y devuelven resultados: parámetros, argumentos y `return` (Clase 21), incluyendo los trucos profesionales como valores por defecto, argumentos por nombre, `*args` y `**kwargs`. También hablaremos del `scope`, ese tema que confunde a todos al principio: qué variables vive dentro de la función y cuáles afuera, y por qué no debes abusar de la palabra `global`. Aprenderás a documentar tus funciones con `docstrings`, a dividir problemas grandes en funciones pequeñas y a reconocer los errores clásicos (como olvidar los paréntesis al llamar una función, o que tu función "no devuelve nada" porque olvidaste el `return`).

Y para cerrar con broche de oro (Clase 27), las list comprehensions: la forma elegante y pitónica de crear listas en una sola línea. Vas a ver que un ciclo `for` de cuatro líneas para filtrar números pares se convierte en una sola línea legible y hermosa. Pero ojo: elegante no significa complicado; también te voy a enseñar cuándo SÍ usarlas y cuándo NO, porque una comprensión de tres líneas que nadie entiende es peor que un `for` clarito.

Al terminar esta unidad vas a mirar tus programas viejos y te va a dar un poco de vergüenza lo repetitivos que eran. Esa vergüenza es buena señal: significa que ya piensas como programador. Vamos a ello.

Explicación detallada

Qué es una función y por qué la necesitas

Una función es un bloque de código que tiene un nombre y que puedes ejecutar cuantas veces quieras con solo escribir ese nombre. Técnicamente: es una porción de código reutilizable que realiza una tarea específica. Pero olvídate de la definición de diccionario y quédate con la imagen de la tarjeta de receta pegada en la pared de la taquería: la escribes una vez, la usas siempre.

Mira el problema que resuelven las funciones con un ejemplo de la vida real. Supón que estás haciendo el programa de una tienda y en cinco lugares distintos necesitas mostrar un encabezado bonito:

```
print("=" * 40)
print("  TIENDA DOÑA LUPE")
print("=" * 40)
```

Sin funciones, copias esas tres líneas en cinco lugares: 15 líneas repetidas. Ahora imagina que Doña Lupe cambia el nombre de la tienda a "ABARROTOS DOÑA LUPE". Tienes que buscar y cambiar las cinco copias. Si se te olvida una, el programa muestra dos nombres distintos y quedas mal con la clienta. Eso es exactamente lo que pasa en programas reales: el código repetido se desincroniza, y cada copia es una oportunidad de cometer un error diferente.

Con una función, escribes esto una sola vez:

```
def mostrar_encabezado():
    print("=" * 40)
    print("  TIENDA DOÑA LUPE")
    print("=" * 40)
```

Y en los cinco lugares donde la necesitas, solo escribes:

```
mostrar_encabezado()
```

Una línea en vez de tres. Y cuando cambie el nombre de la tienda, lo cambias en UN solo lugar. Esa es la magia: un solo punto de verdad.

Esto tiene un nombre en la industria: DRY, que significa "Don't Repeat Yourself" (no te repitas). Es uno de los principios más importantes de la programación profesional. Su contrario también tiene nombre, y es gracioso: WET, "Write Everything Twice" (escribe todo dos veces), o "We Enjoy Typing" (disfrutamos teclear). Cada vez que te descubras copiando y pegando el mismo bloque de código, párate y pregúntate: "¿esto debería ser una función?". Nueve de cada diez veces, la respuesta es sí.

Pero DRY no es la única razón. Las funciones también te dan abstracción, una palabra elegante que significa "esconder los detalles". Cuando llamas a `mostrar_encabezado()`, no necesitas saber CÓMO dibuja las líneas de igual; solo necesitas saber QUÉ hace. Es como manejar un carro: presionas el acelerador sin saber cómo funciona la inyección de combustible. Las funciones te permiten pensar en tu programa a un nivel más alto: "primero pido los datos, luego los valido, luego calculo, luego muestro", en vez de ahogarte en los detalles de cada paso al mismo tiempo.

Y hay una tercera razón que los principiantes subestiman: las funciones hacen tu código comprobable. Probar que un bloque suelto de 200 líneas funciona es un dolor de cabeza; probar una función pequeña que recibe dos números y devuelve su suma es trivial: la llamas con 2 y 3, y verificas que devuelva 5. Los programadores profesionales escriben pruebas automáticas para sus funciones, y eso solo es posible porque el código está dividido en piezas pequeñas con entradas y salidas claras.

La sintaxis de def, pieza por pieza

En Python, las funciones se definen con la palabra reservada `def` (de "define", definir). Mira la estructura completa y luego la desarmamos:

```
def saludar():  
    print("¡Hola! Bienvenido a la academia.")
```

Vamos parte por parte:

1. `def`: le dice a Python "lo que sigue es la definición de una función". Es obligatorio, siempre en minúsculas.
2. `saludar`: el nombre de la función. Tú lo eliges. Debe seguir las mismas reglas que los nombres de variables: letras, números y guiones bajos, sin espacios, sin empezar con número. Por convención, los nombres de funciones van en minúsculas con guiones bajos entre palabras (snake_case): `calcular_total`, `validar_contrasena`, `mostrar_menu`. Y una convención todavía más importante: los nombres de funciones suelen ser verbos o

empezar con un verbo, porque las funciones HACEN cosas: saludar, calcular, mostrar, validar, convertir. Cuando lees saludar(), tu cerebro entiende al instante "esta función saluda".

3. (): los paréntesis. Son obligatorios aunque la función no reciba datos. Aquí es donde más adelante pondremos los parámetros. Mucho ojo: definir la función NO la ejecuta. def solo crea la "tarjeta de receta" y la pega en la pared; el código de adentro no se ejecuta hasta que llames a la función.
4. :: los dos puntos marcan el inicio del bloque de la función, igual que en el if y en los ciclos.
5. El cuerpo indentado: las líneas con sangría (4 espacios) son las instrucciones que forman la función. La indentación es lo que le dice a Python dónde empieza y dónde termina la función.

Un detalle crucial que confunde a muchos al principio: el orden importa. Tienes que DEFINIR la función antes de llamarla. Python lee tu programa de arriba hacia abajo, y si intentas llamar a saludar() antes de la línea def saludar():, te va a soltar un NameError porque todavía no existe esa tarjeta en la pared. La costumbre profesional es definir todas las funciones al inicio del archivo (después de los imports) y dejar las llamadas al final. Así tu programa se lee como un libro bien organizado: primero las definiciones, luego la acción.

Llamar a una función: presionar el botón

Definir es escribir la receta; llamar es cocinar. Para llamar (también se dice "invocar") a una función, escribes su nombre seguido de paréntesis:

```
def saludar():  
    print("¡Hola! Bienvenido a la academia.")  
  
saludar()  
saludar()  
saludar()
```

La salida será:

```
¡Hola! Bienvenido a la academia.  
¡Hola! Bienvenido a la academia.  
¡Hola! Bienvenido a la academia.
```

Definiste una vez, llamaste tres veces. Cada llamada ejecuta el cuerpo de la función de principio a fin, y cuando termina, Python regresa al punto donde la llamaste y sigue con la siguiente línea. Es como si el programa "viajara" a la función, hiciera lo que dice y volviera.

Fíjate en algo importante: los paréntesis en la llamada NO son opcionales. Si escribes saludar sin paréntesis, Python no ejecuta la función; solo te entrega una referencia a la función misma (algo como <function saludar at 0x...>), que es un objeto que existe pero no hace nada. Es uno de los errores más comunes y más frustrantes para principiantes: "mi función no hace nada" y resulta que olvidaron los paréntesis. Grábatelo: sin paréntesis no hay llamada.

Otro punto: puedes llamar a una función desde cualquier parte del programa DESPUÉS de haberla definido, incluso desde dentro de otra función (ya veremos eso más adelante con "funciones que llaman a funciones"). Y una función puede llamarse a sí misma, aunque eso (la recursividad) es un tema avanzado que dejaremos para después; por ahora solo sabe que existe.

Mira un ejemplo un poco más completo para que veas el flujo del programa. Imagina el sistema de la tienda de Doña Lupe:

```
def mostrar_encabezado():
    print("=" * 40)
    print("  TIENDA DOÑA LUPE")
    print("=" * 40)

def mostrar_despedida():
    print("=" * 40)
    print("  ¡Gracias por su compra!")
    print("=" * 40)

mostrar_encabezado()
print("Producto: Tortillas - $25")
print("Producto: Frijoles - $30")
mostrar_despedida()
```

La salida:

```

=====
TIENDA DOÑA LUPE
=====
Producto: Tortillas - $25
Producto: Frijoles - $30
=====
¡Gracias por su compra!
=====

```

¿Ves cómo el programa se lee casi como español? "Muestra el encabezado, imprime los productos, muestra la despedida." Eso es código elegante: el que se lee fácil. Las funciones convierten tu programa en una lista de intenciones claras en lugar de una maraña de detalles.

Parámetros y argumentos: la entrada de la función

Las funciones que no reciben datos son útiles (como `mostrar_encabezado`), pero las funciones realmente poderosas son las que reciben información, trabajan con ella y te devuelven un resultado. Volvamos al microondas: el botón de "palomitas" no necesita que le digas nada, pero el de "descongelar" sí necesita saber cuántos kilos de carne vas a meter. Esa información que le pasas a la función se maneja con dos conceptos que suenan parecidos pero NO son lo mismo: parámetros y argumentos.

Un parámetro es la variable que aparece en la DEFINICIÓN de la función: es el "lugar vacío" que la función reserva para recibir un dato. Un argumento es el valor real que le pasas cuando LLAMAS a la función: es el dato concreto que llena ese lugar. Mira:

```

def saludar(nombre):      # 'nombre' es el PARÁMETRO (lugar vacío)
    print(f"¡Hola, {nombre}!")

saludar("Esteban")        # "Esteban" es el ARGUMENTO (valor real)
saludar("María")          # "María" es otro argumento

```

La salida:

```

¡Hola, Esteban!
¡Hola, María!

```

La analogía de la receta lo deja clarísimo: la tarjeta de la salsa dice "ingredientes: jitomates, chiles, ajo". Esos son los parámetros: los huecos que la receta necesita. Cuando cocinas hoy, usas 5 jitomates y 3 chiles; mañana usas 8 jitomates y 1 chile. Esos valores concretos de cada

día son los argumentos. La receta (función) es la misma; los ingredientes (argumentos) cambian.

Algunas reglas importantes:

Primera: el número de argumentos debe coincidir con el número de parámetros (salvo que uses valores por defecto, que veremos en un momento). Si la función espera dos parámetros y le pasas uno, Python te lanza un `TypeError` y te dice exactamente qué pasó. Es como una receta que pide harina y huevo: si solo le das harina, no sale.

```
def sumar(a, b):  
    print(a + b)  
  
sumar(5, 3)      # Bien: 8  
sumar(5)         # TypeError: sumar() missing 1 required positional argument: 'b'
```

Segunda: los parámetros solo existen DENTRO de la función. Son variables locales (hablaremos del scope más adelante). Cuando la función termina, esos valores se olvidan. No puedes usar `a` o `b` fuera de `sumar`; si lo intentas, `NameError`.

Tercera: el orden importa cuando pasas argumentos por posición. En `sumar(5, 3)`, el 5 va a `a` y el 3 va a `b`. Si los inviertes en una resta, el resultado cambia:

```
def restar(a, b):  
    print(a - b)  
  
restar(10, 4)    # 6  
restar(4, 10)    # -6 ¡Cuidado con el orden!
```

Este es uno de los errores clásicos: pasar los argumentos en orden equivocado y obtener un resultado incorrecto sin que Python se queje (porque los tipos coinciden, solo el orden está mal). Por eso los nombres de parámetros deben ser descriptivos: `restar(minuendo, sustraendo)` es más claro que `restar(a, b)`, aunque en la práctica verás ambas formas.

Los parámetros convierten a tus funciones en moldes flexibles. Sin parámetros, `mostrar_encabezado` solo sirve para Doña Lupe; con un parámetro, sirve para cualquier tienda:

```
def mostrar_encabezado(nombre_tienda):
    print("=" * 40)
    print(f"    {nombre_tienda}")
    print("=" * 40)

mostrar_encabezado("TIENDA DOÑA LUPE")
mostrar_encabezado("ABARROTES EL FARO")
mostrar_encabezado("MINISUPER LA ESQUINA")
```

Una función, tres tiendas. Eso es reutilización de verdad.

Return: la salida de la función

Si los parámetros son la entrada, return es la salida. La instrucción return hace dos cosas al mismo tiempo: TERMINA la ejecución de la función inmediatamente y DEVUELVE un valor a quien la llamó. Ese valor devuelto es lo que la función "produce", como el microondas que te devuelve la comida caliente.

Mira la diferencia entre una función que IMPRIME y una que DEVUELVE. Es una distinción fundamental:

```
def sumar_imprime(a, b):
    print(a + b)

def sumar_devuelve(a, b):
    return a + b
```

sumar_imprime(5, 3) muestra 8 en pantalla, pero no te da nada que puedas guardar o reutilizar: el 8 se fue a la pantalla y se perdió. En cambio:

```
resultado = sumar_devuelve(5, 3)
print(resultado)          # 8
doble = resultado * 2
print(doble)              # 16
```

El valor devuelto por return lo puedes guardar en una variable, usarlo en operaciones, pasarlo a otra función. Es la diferencia entre un empleado que te GRITA el resultado y se va, y uno que te ENTREGA el resultado en un papel para que lo uses. Casi siempre quieres el segundo.

Un error típico de principiante es este:

```
def sumar(a, b):
    resultado = a + b    # Se calcula...

total = sumar(5, 3)
print(total)             # ¡Imprime None!
```

¿Por qué None? Porque la función calculó el resultado pero nunca lo devolvió con return; se quedó guardado en una variable local que murió cuando la función terminó. Python, ante una función sin return, devuelve automáticamente el valor especial None (que significa "nada", "vacío"). Entonces total vale None. Grábatelo con sangre: si quieres usar el resultado fuera de la función, necesitas return. "Olvidar el return y recibir None" es probablemente el error número uno de esta unidad.

Otra cosa clave: return termina la función AL INSTANTE. Todo lo que esté después de un return que se ejecute, no se ejecuta:

```
def ejemplo():
    return "terminé"
    print("Esto jamás se imprime")

print(ejemplo())    # Solo muestra: terminé
```

Esto no es un defecto, es una característica: te permite tener "salidas tempranas". Por ejemplo, si algo está mal en los datos, devuelves el error de inmediato sin seguir procesando:

```
def dividir(a, b):
    if b == 0:
        return "Error: no se puede dividir entre cero"
    return a / b
```

Si b es cero, la función devuelve el mensaje y se acabó; si no, sigue a la segunda línea y devuelve la división. Este patrón de "verificar primero, salir temprano" hace el código mucho más legible que anidar if dentro de if.

Return múltiple: devolver varias cosas con tuplas

¿Y si tu función necesita devolver más de un valor? Python tiene un truco elegante: puedes poner varios valores separados por comas en el return, y Python los empaqueta automáticamente en una tupla (¿recuerdas las tuplas de la Unidad 6? Son como listas pero inmutables):

```
def estadisticas(numeros):
    total = sum(numeros)
    promedio = total / len(numeros)
    return total, promedio, len(numeros)

resultado = estadisticas([10, 20, 30])
print(resultado)  # (60, 20.0, 3) <- ¡una tupla!
```

Y lo más bonito: puedes desempaquetar esos valores directamente en variables separadas, en una sola línea:

```
total, promedio, cantidad = estadisticas([10, 20, 30])
print(f"Total: {total}, Promedio: {promedio}, Cantidad: {cantidad}")
```

Esto se lee de maravilla y es muy pitónico. Eso sí, no abuses: si tu función devuelve 6 o 7 valores, probablemente está haciendo demasiadas cosas (viola la regla de "una función = una tarea") y sería mejor que devolviera un diccionario con claves descriptivas:

```
def estadisticas(numeros):
    return {
        "total": sum(numeros),
        "promedio": sum(numeros) / len(numeros),
        "cantidad": len(numeros)
    }
```

Return sin valor y el misterioso None

Puedes escribir return solo, sin nada después. Eso termina la función y devuelve None. Se usa cuando quieres salir temprano de una función que normalmente no devuelve nada (por ejemplo, una que solo imprime):

```
def mostrar_mayores_edad(edad):
    if edad < 0:
        print("Edad inválida")
        return  # Salimos temprano, devolvemos None
    if edad >= 18:
        print("Eres mayor de edad")
    else:
        print("Eres menor de edad")
```


Y recuerda: TODA función en Python devuelve algo. Si no hay ningún return, devuelve None automáticamente. Por eso `print(saludar())` donde `saludar` solo imprime, muestra el saludo y después la palabra `None`: primero se ejecuta el print de adentro, luego el print de afuera imprime el valor devuelto, que es `None`. Cuando veas un `None` inesperado en tu programa, tu primer sospechoso debe ser "olvidé el return".

Hay una sutileza que vale oro: la diferencia entre `print` y `return` es la diferencia entre mostrar y entregar. `print` es para los humanos (muestra en pantalla), `return` es para el programa (entrega el valor para seguir trabajando). Una función bien diseñada casi siempre usa `return` y deja que QUIEN LA LLAMA decida si imprimir el resultado o no. Así la función sirve en más contextos: hoy la usas en un programa que imprime, mañana en uno que guarda en un archivo, pasado en una página web. Si la función imprime directamente, la amarras a la pantalla para siempre.

Parámetros por defecto: valores que ya vienen incluidos

A veces un parámetro casi siempre recibe el mismo valor, y obligar a quien llama a escribirlo cada vez es tedioso. Python te permite darle un valor por defecto en la definición: si al llamar no pasas ese argumento, se usa el valor por defecto; si lo pasas, se usa el tuyo.

```
def saludar(nombre, saludo="Hola"):
    print(f"{saludo}, {nombre}!")

saludar("Esteban")           # Hola, Esteban! (usa el valor por defecto)
saludar("María", "Buenos días") # Buenos días, María! (usa el valor dado)
```

La analogía perfecta es la fonda de la esquina: pides "la comida corrida" y ya incluye agua de jamaica, a menos que pidas cambiarla por horchata. El valor por defecto es el agua de jamaica: viene incluida si no dices nada.

Reglas importantes:

Primera: los parámetros con valor por defecto van DESPUÉS de los parámetros sin valor por defecto. Esto es obligatorio en Python:

```
def funcion(a, b=10):    # Bien
def funcion(a=10, b):    # SyntaxError: non-default argument follows default argument
```

Tiene lógica: si llamas `funcion(5)`, Python asigna por posición de izquierda a derecha, así que los obligatorios deben ir primero para que no haya ambigüedad.

Segunda: cuidado con el valor por defecto MUTABLE (listas o diccionarios). Este es un error clásico y traicionero:

```
def agregar(elemento, lista=[]):    # ¡PELIGRO!
    lista.append(elemento)
    return lista

print(agregar("a"))    # ['a']
print(agregar("b"))    # ['a', 'b']    ¡¿Qué?! Esperábamos ['b']
```

¿Qué pasó? El valor por defecto [] se crea UNA sola vez, cuando se define la función, no en cada llamada. Entonces todas las llamadas comparten la MISMA lista, y los elementos se acumulan. Es como si la receta dijera "usa el mismo sartén sin lavarlo entre cada uso". La solución profesional es usar None como valor por defecto y crear la lista dentro:

```
def agregar(elemento, lista=None):
    if lista is None:
        lista = []
    lista.append(elemento)
    return lista

print(agregar("a"))    # ['a']
print(agregar("b"))    # ['b']    ¡Correcto!
```

Este patrón (None como centinela) lo vas a ver en código profesional todo el tiempo. Apréndetelo desde ahora.

Argumentos por nombre (keyword arguments): claridad total

Hasta ahora hemos pasado argumentos por posición: el primero va al primer parámetro, el segundo al segundo. Pero Python también te permite especificar A QUÉ parámetro va cada argumento, usando su nombre:

```
def describir_mascota(nombre, especie="perro", edad=3):
    print(f"{nombre} es un {especie} de {edad} años.")

describir_mascota("Firulais")
describir_mascota("Mishi", especie="gato", edad=2)
describir_mascota(edad=5, nombre="Rocky")    # ¡El orden ya no importa!
```

Cuando usas el nombre del parámetro, puedes pasar los argumentos en CUALQUIER orden, porque Python ya sabe a dónde va cada uno. Esto es oro puro para la legibilidad: `describir_mascota(edad=5, nombre="Rocky")` se entiende al instante, mientras que `describir_mascota("Rocky", "perro", 5)` te obliga a recordar qué significa cada posición.

Buenas prácticas con esto: cuando una función tiene más de dos o tres parámetros, usa argumentos por nombre en las llamadas. Tu "yo del futuro" (y tus compañeros) te lo van a agradecer. Además, puedes mezclar: primero los posicionales y después los nombrados, pero nunca un posicional después de uno nombrado:

```
describir_mascota("Firulais", edad=4)    # Bien
describir_mascota(especie="gato", "Mishi") # SyntaxError
```

***args: cuando no sabes cuántos argumentos vendrán**

A veces una función necesita aceptar CUALQUIER cantidad de argumentos. Por ejemplo, una función sumar que sume 2 números, o 5, o 20. Para eso existe `*args` (el asterisco es la magia; "`args`" es solo el nombre por convención):

```
def sumar_todo(*args):
    total = 0
    for numero in args:
        total += numero
    return total

print(sumar_todo(1, 2))           # 3
print(sumar_todo(1, 2, 3, 4, 5)) # 15
print(sumar_todo())               # 0
```

Dentro de la función, `args` es una TUPLA con todos los argumentos que le pasaste. Puedes recorrerla, medirla con `len`, indexarla. Es como una receta que dice "agrega todas las verduras que tengas": no importa si son 2 o 10, la instrucción funciona igual.

Un ejemplo más de la vida real: una función que calcula el promedio de calificaciones, sin importar cuántas materias lleve el alumno:

```
def promedio(*calificaciones):
    if len(calificaciones) == 0:
        return 0
    return sum(calificaciones) / len(calificaciones)

print(promedio(9, 8, 10))      # 9.0
print(promedio(7, 7, 8, 9, 10)) # 8.2
```

Fíjate que aquí usamos `sum()`, una función integrada de Python que suma todos los elementos de un iterable. Ya iremos conociendo más de esas.

****kwargs: argumentos con nombre en cantidad libre**

El primo de `*args` es `**kwargs` (doble asterisco; "kwargs" viene de "keyword arguments"). Recibe cualquier cantidad de argumentos POR NOMBRE y los empaqueta en un DICCIONARIO:

```
def mostrar_datos(**kwargs):
    for clave, valor in kwargs.items():
        print(f"{clave}: {valor}")

mostrar_datos(nombre="Esteban", edad=30, ciudad="CDMX")
```

Salida:

```
nombre: Esteban
edad: 30
ciudad: CDMX
```

Esto es utilísimo para funciones flexibles, como una que arma la ficha de un producto donde cada producto tiene atributos distintos:

```
def ficha_producto(nombre, **detalles):
    print(f"Producto: {nombre}")
    for atributo, valor in detalles.items():
        print(f"  - {atributo}: {valor}")

ficha_producto("Laptop", marca="Lenovo", ram="16GB", precio=15000)
ficha_producto("Taza", color="roja")
```

Puedes combinar parámetros normales con `*args` y `**kwargs`, pero el orden en la definición es sagrado: primero los normales, luego `*args`, luego `**kwargs`:

```
def funcion(a, b, *args, **kwargs): # Orden correcto
```

No necesitas dominar `*args` y `**kwargs` a la perfección hoy; con entender qué hacen y reconocerlos cuando los veas en código ajeno (aparecen MUCHÍSIMO en bibliotecas profesionales) ya ganaste. Cuando escribas tus propias funciones flexibles, te van a salir naturales.

Scope: dónde vive cada variable (locales vs globales)

Este es el tema que más confunde a los principiantes, así que vamos con calma y con una buena analogía. El scope (alcance) es la "zona" del programa donde una variable existe y puede usarse. Piensa en tu casa: lo que está en tu recámara es tuyo y privado; lo que está en la sala lo puede usar toda la familia; y lo que dejas en la banqueta lo ve todo el barrio. Las variables funcionan igual.

Una variable creada DENTRO de una función es una variable LOCAL: solo existe dentro de esa función, nace cuando la función se ejecuta y muere cuando termina. Es como los ingredientes que sacas para cocinar: existen mientras cocinas, pero cuando terminas y lavas los trastes, desaparecen de la mesa.

```
def cocinar():
    ingrediente = "jitomate" # Variable LOCAL
    print(f"Cocinando con {ingrediente}")

cocinar()
print(ingrediente) # NameError: name 'ingrediente' is not defined
```

La última línea falla porque `ingrediente` solo existe dentro de `cocinar`. Afuera, Python no sabe de qué hablas. Esto NO es una limitación: es una protección. Imagina que tienes 20 funciones y todas usan una variable llamada `total` para cosas distintas; si las variables locales "se escaparan", se pisarían entre sí y sería un caos. El scope local es como darle a cada cocinero su propia mesa de trabajo: nadie revuelve los ingredientes del otro.

Una variable creada FUERA de todas las funciones (al nivel principal del programa) es una variable GLOBAL: existe en todo el programa, y las funciones pueden LEERLA sin problema:

```

descuento = 0.10    # Variable GLOBAL

def calcular_precio(precio):
    return precio * (1 - descuento)    # Lee la global sin problema

print(calcular_precio(100))    # 90.0

```

Hasta aquí todo bien. El problema viene cuando una función intenta MODIFICAR una variable global:

```

contador = 0

def incrementar():
    contador = contador + 1    # ¡ERROR! UnboundLocalError
    print(contador)

incrementar()

```

Esto falla con un error rarísimo: `UnboundLocalError`. ¿Por qué? Porque en cuanto Python ve una ASIGNACIÓN a una variable dentro de una función (`contador = ...`), asume que esa variable es LOCAL, y entonces cuando intenta leer `contador + 1`, la variable local todavía no tiene valor. Es como si el cocinero dijera "voy a usar MI sartén" pero nunca lo sacó del cajón.

Para modificar una global desde una función necesitas la palabra reservada `global`, que le dice a Python "no crees una variable local, usa la global":

```

contador = 0

def incrementar():
    global contador
    contador = contador + 1

incrementar()
incrementar()
print(contador)    # 2

```

Funciona, sí. Pero aquí viene el consejo profesional, directo y sin rodeos: EVITA global tanto como puedas. ¿Por qué? Porque las funciones que modifican variables globales son impredecibles: su comportamiento depende de algo que está fuera de ellas, invisible en su definición. Es como una receta que dice "usa la sal que está en la alacena de tu vecino": funciona hoy, pero si el vecino cambia la sal por azúcar, tu platillo se arruina y no sabes por

qué. Los programas con muchos global se vuelven imposibles de depurar porque cualquier función puede cambiar cualquier variable en cualquier momento.

La alternativa profesional es simple: pasa los datos como PARÁMETROS y devuelve los resultados con RETURN. Así la función es transparente: todo lo que necesita entra por la puerta (parámetros) y todo lo que produce sale por la ventana (return). Nada de magia invisible:

```
# Mal: depende de una global invisible
total = 0
def agregar_malo(precio):
    global total
    total += precio

# Bien: transparente, recibe y devuelve
def agregar_bien(total, precio):
    return total + precio

total = 0
total = agregar_bien(total, 25)
total = agregar_bien(total, 30)
print(total)    # 55
```

La versión "bien" es más larga en las llamadas, pero cada línea dice exactamente qué pasa. Cuando tu programa tenga 500 líneas, vas a bendecir esta claridad.

Hay una regla de oro para el scope: las funciones deben comunicarse con el exterior SOLO a través de parámetros y return. Ni leer globales a escondidas (mejor pasarlas como parámetro), ni modificar globales (mejor devolver el nuevo valor). Las constantes de verdad globales (como $\pi = 3.14159$ o $\text{IVA} = 0.16$, que nunca cambian) son la excepción aceptable: se escriben en MAYÚSCULAS por convención y todas las funciones pueden leerlas.

Docstrings: documenta tus funciones como profesional

Cuando escribes una función, en seis meses no vas a recordar qué hacía exactamente, qué parámetros esperaba ni qué devolvía. Y si trabajas en equipo, tus compañeros menos. La solución son los docstrings: cadenas de documentación que van JUSTO debajo de la línea def, entre triples comillas:

```
def calcular_iva(precio, tasa=0.16):
    """Calcula el precio con IVA incluido.

    Args:
        precio: precio base sin impuestos (número).
        tasa: tasa de IVA como decimal (por defecto 0.16).

    Returns:
        El precio total con IVA incluido (número).
    """
    return precio * (1 + tasa)
```

El docstring no afecta la ejecución; Python lo guarda como documentación oficial de la función. Y tiene un superpoder: puedes consultarlo en cualquier momento con `help()`:

```
help(calcular_iva)
```

Esto muestra tu documentación bonita y ordenada. Los editores de código modernos también la muestran automáticamente cuando pasas el cursor sobre la función. Es como pegarle a la tarjeta de receta una notita que dice "esta salsa pica mucho, úsala con cuidado": quien la use después te lo agradece.

El formato del ejemplo (con secciones `Args:` y `Returns:`) es una convención popular, pero lo mínimo aceptable es una primera línea clara que diga QUÉ hace la función. Acostúmbrate desde ahora: cada función que escribas, su docstring. No es burocracia, es cortesía con tu yo del futuro.

Funciones que llaman a funciones y descomposición de problemas

Una de las ideas más poderosas de la programación: las funciones pueden llamar a otras funciones. Esto te permite DESCOMPONER un problema grande en piezas pequeñas, como desarmar un motor en sus partes para entenderlo.

Imagina que debes hacer un programa que procese la nómina de una empresa. Suena enorme, ¿verdad? Pero si lo descompones, son piezas manejables:


```
def calcular_sueldo_bruto(horas, tarifa):
    """Calcula el sueldo antes de impuestos."""
    return horas * tarifa

def calcular_impuestos(sueldo_bruto):
    """Calcula los impuestos (16% del sueldo bruto)."""
    return sueldo_bruto * 0.16

def calcular_sueldo_netto(horas, tarifa):
    """Calcula el sueldo final llamando a las otras funciones."""
    bruto = calcular_sueldo_bruto(horas, tarifa)
    impuestos = calcular_impuestos(bruto)
    return bruto - impuestos

print(calcular_sueldo_netto(40, 150))    # 5040.0
```

¿Ves la belleza? `calcular_sueldo_netto` no sabe CÓMO se calculan los impuestos; solo llama a la función que sí sabe. Cada función es simple (2-3 líneas), tiene UN trabajo claro y un nombre que lo dice todo. Si mañana cambia la tasa de impuestos, tocas UNA función. Si hay un error en el cálculo del bruto, sabes exactamente dónde buscar.

Esta es la estrategia profesional para enfrentar cualquier problema: divide y vencerás. Ante un problema grande, no intentes escribir la solución completa de un jalón. Pregúntate: "¿en qué pasos pequeños se divide esto?" Cada paso es una función. Luego escribes una función "directora" que las llama en orden. Los programadores experimentados pasan más tiempo pensando en CÓMO dividir el problema que escribiendo código, y esa es exactamente la habilidad que te va a diferenciar.

Un ejemplo más, de la vida diaria: validar si una contraseña es segura. En vez de una función gigante, la descompones:

```

def tiene_longitud_valida(clave):
    return len(clave) >= 8

def tiene_numero(clave):
    for caracter in clave:
        if caracter.isdigit():
            return True
    return False

def tiene_mayuscula(clave):
    for caracter in clave:
        if caracter.isupper():
            return True
    return False

def es_contrasena_segura(clave):
    """Valida que la contraseña cumpla todos los requisitos."""
    return (tiene_longitud_valida(clave)
            and tiene_numero(clave)
            and tiene_mayuscula(clave))

print(es_contrasena_segura("abc123"))      # False (muy corta, sin mayúscula)
print(es_contrasena_segura("Secreta123"))  # True

```

Cada mini-función se entiende en segundos y se puede probar por separado. La función directora las combina con `and`. Si mañana agregas el requisito de "debe tener un símbolo", agregas una mini-función y un `and` más. Así se construye software que crece sin romperse.

List comprehensions: listas elegantes en una línea

Llegamos a la joya de esta unidad (Clase 27). Una list comprehension (comprensión de lista) es una forma compacta y elegante de crear una lista en una sola línea, transformando o filtrando los elementos de otro iterable. Si los ciclos `for` son el español básico de Python, las comprehensions son hablarlo con fluidez y acento nativo.

Primero el problema que resuelven. Supón que quieres una lista con los cuadrados de los números del 1 al 5. Con lo que sabes hasta ahora, harías:

```
cuadrados = []
for numero in range(1, 6):
    cuadrados.append(numero ** 2)
print(cuadrados)    # [1, 4, 9, 16, 25]
```

Cuatro líneas para una idea simple: "los cuadrados del 1 al 5". Con una list comprehension, la misma idea cabe en una línea:

```
cuadrados = [numero ** 2 for numero in range(1, 6)]
print(cuadrados)    # [1, 4, 9, 16, 25]
```

La sintaxis tiene tres partes, y se lee casi como español: [expresión for variable in iterable]. Léela así: "el cuadrado de cada número, para cada número en el rango del 1 al 5". La expresión (numero ** 2) es lo que se guarda en la lista; el for recorre el iterable; la variable toma cada valor por turno.

Otro ejemplo, muy de la vida real: tienes una lista de precios sin IVA y quieres otra con IVA incluido:

```
precios = [100, 250, 80, 400]
con_iva = [precio * 1.16 for precio in precios]
print(con_iva)    # [116.0, 290.0, 92.8, 464.0]
```

Y uno más, convirtiendo nombres a mayúsculas:

```
nombres = ["ana", "luis", "maría"]
mayusculas = [nombre.upper() for nombre in nombres]
print(mayusculas)    # ['ANA', 'LUIS', 'MARÍA']
```

¿Ves el patrón? "Transforma cada elemento" es el caso de uso número uno de las comprehensions.

Comprehensions con if: filtrar elementos

Puedes agregar una condición al final para FILTRAR: solo los elementos que cumplan la condición entran a la lista nueva. La sintaxis es [expresión for variable in iterable if condición]:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares = [n for n in numeros if n % 2 == 0]
print(pares)    # [2, 4, 6, 8, 10]
```

Se lee: "cada número, para cada número en la lista, SI el número es par". El if va al final y actúa como portero de antro: solo deja pasar a los que cumplen el requisito.

Puedes combinar transformación Y filtro en la misma línea:

```
# Cuadrados de solo los números pares del 1 al 10
resultado = [n ** 2 for n in range(1, 11) if n % 2 == 0]
print(resultado)    # [4, 16, 36, 64, 100]
```

Ejemplo práctico: de una lista de alumnos con calificaciones, quedarte solo con los aprobados (6 o más):

```
calificaciones = [10, 5, 8, 4, 9, 6, 3]
aprobados = [c for c in calificaciones if c >= 6]
print(aprobados)    # [10, 8, 9, 6]
```

Comprehensions con if/else: elegir entre dos valores

Ojo con esto porque la sintaxis cambia de lugar y confunde a muchos. Si quieres un if/else (elegir entre dos transformaciones según una condición), el if/else va AL PRINCIPIO, antes del for:

```
numeros = [1, 2, 3, 4, 5]
etiquetas = ["par" if n % 2 == 0 else "impar" for n in numeros]
print(etiquetas)    # ['impar', 'par', 'impar', 'par', 'impar']
```

La lógica: el if/else forma parte de la EXPRESIÓN (qué valor se guarda), mientras que el if solo forma parte del FILTRO (si entra o no). Compáralos:

```
# Filtro (if al final): solo entran los pares
[n for n in numeros if n % 2 == 0]

# Elección (if/else al inicio): entran todos, transformados según el caso
["par" if n % 2 == 0 else "impar" for n in numeros]
```

Un ejemplo útil: clasificar temperaturas:

```
celsius = [0, 15, 25, 35, -5]
clima = ["calor" if t >= 25 else "templado" if t >= 10 else "frío" for t in
celsius]
print(clima)    # ['frío', 'templado', 'calor', 'calor', 'frío']
```

(Sí, se pueden encadenar, pero con moderación: si necesitas tres niveles ya estás al límite de lo legible.)

La equivalencia con el ciclo for: no es magia

Toda list comprehension tiene un ciclo for equivalente, y entender esa equivalencia te da control total. La regla de traducción es mecánica:

```
# Comprehension:
[nueva_expr for x in iterable if condicion]

# Equivale exactamente a:
resultado = []
for x in iterable:
    if condicion:
        resultado.append(nueva_expr)
```

Haz la traducción mental un par de veces y las comprehensions dejarán de parecer magia. De hecho, un buen ejercicio es tomar cualquier comprehension y escribir su for equivalente, y viceversa.

Cuándo Sí y cuándo NO usarlas: la legibilidad manda

Aquí viene la parte que separa al programador elegante del presumido. Las comprehensions son hermosas, pero tienen un límite: la LEGIBILIDAD. Python tiene una filosofía escrita en el "Zen de Python" que dice "la legibilidad cuenta" (Readability counts). Una comprehension que nadie entiende en 5 segundos es peor que un for de cuatro líneas claritas.

Úsalas cuando:

- La transformación es simple y se entiende de un vistazo: `[p * 1.16 for p in precios]`.
- El filtro es una condición corta: `[n for n in numeros if n > 0]`.
- Toda la línea cabe cómodamente en pantalla (digamos, menos de 80 caracteres).

NO las uses cuando:

- Necesitas condiciones anidadas complejas o más de un if/else encadenado. Esto es un crimen contra la legibilidad:

```
# ¡NO HAGAS ESTO! Ilegible:
resultado = [[y * 2 if y > 0 else 0 for y in x if y is not None] for x in matriz if
len(x) > 0]
```

- El cuerpo del ciclo hace varias cosas (imprimir, guardar en archivo, actualizar variables). Las comprehensions son para CREAR listas, no para ejecutar efectos secundarios.
- Apenas estás aprendiendo y el for te resulta más claro. No hay vergüenza en eso: el código claro siempre le gana al código "inteligente".

La regla de oro: si tienes que leer la comprehension dos veces para entenderla, escríbela como un for. La elegancia real no es escribir poco código; es escribir código que otros (y tu yo del futuro) entiendan rápido.

Dict y set comprehensions: mención breve pero útil

La idea de las comprehensions no es exclusiva de las listas. También puedes crear diccionarios y conjuntos en una línea:

```
# Dict comprehension: {clave: valor for ...}
cuadrados_dict = {n: n ** 2 for n in range(1, 6)}
print(cuadrados_dict)  # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Set comprehension: {expr for ...} (sin los dos puntos)
letras = {palabra[0] for palabra in ["ana", "luis", "alberto", "maría"]}
print(letras)  # {'a', 'l', 'm'} (sin duplicados, orden no garantizado)
```

El dict comprehension es perfecto para construir tablas de conversión o índices rápidos; el set comprehension, para obtener valores únicos transformados. No los vas a usar todos los días, pero cuando los necesites, te ahorran varias líneas.

Funciones integradas útiles: tus herramientas gratis

Python viene con una caja de herramientas incluida: funciones integradas (built-in) que ya están definidas y listas para usar, sin importar nada. Ya conoces print(), len(), input() y type(). Aquí van otras que se llevan de maravilla con lo que aprendiste en esta unidad:

`sum()`: suma todos los elementos de un iterable. Perfecta dentro de funciones que calculan totales y promedios:

```
ventas = [150, 300, 200, 450]
print(sum(ventas))    # 1100
```

`min()` y `max()`: el menor y el mayor de un iterable (o de varios argumentos):

```
print(min(ventas))    # 150
print(max(ventas))    # 450
```

`sorted()`: devuelve una NUEVA lista ordenada, sin modificar la original (a diferencia de `lista.sort()`, que ordena en su lugar):

```
print(sorted(ventas))          # [150, 200, 300, 450]
print(sorted(ventas, reverse=True)) # [450, 300, 200, 150]
print(ventas)                  # [150, 300, 200, 450] (intacta)
```

`enumerate()`: cuando recorres una lista y necesitas el ÍNDICE y el VALOR al mismo tiempo. Te evita el truco feo de `range(len(lista))`:

```
frutas = ["manzana", "plátano", "uva"]
for i, fruta in enumerate(frutas):
    print(f"{i}: {fruta}")
# 0: manzana
# 1: plátano
# 2: uva
```

Puedes empezar a contar desde otro número con `enumerate(frutas, start=1)`, ideal para menús numerados.

`zip()`: combina dos (o más) iterables elemento por elemento, como un cierre de chamarra:

```
nombres = ["Ana", "Luis", "María"]
edades = [25, 30, 22]
for nombre, edad in zip(nombres, edades):
    print(f"{nombre} tiene {edad} años")
```

Si las listas tienen distinto tamaño, `zip` se detiene en la más corta. Estas funciones son los "botones del microondas" que Python te regala: antes de escribir tu propio ciclo para sumar,

buscar el máximo u ordenar, pregunta si ya existe una función integrada que lo haga. Nueve de cada diez veces, sí.

Analogía de cierre: la cocina profesional

Piensa en todo lo que aprendiste como la diferencia entre cocinar en tu casa y dirigir una cocina profesional. En tu casa, haces todo de memoria y en desorden (código sin funciones). En una cocina profesional, cada platillo tiene su receta escrita (def), cada receta dice qué ingredientes necesita (parámetros) y qué platillo produce (return), las recetas complicadas se dividen en sub-recetas (descomposición), cada receta tiene notas al margen (docstrings), y los cocineros tienen sus estaciones separadas para no estorbarse (scope local). Las comprehensions son como el procesador de alimentos: hacen en segundos lo que a mano tomaría minutos, pero solo sirven para tareas simples; nadie mete un pollo entero al procesador. Y las funciones integradas son los electrodomésticos que ya vienen con la cocina: úsalos antes de hacer todo a mano.

Errores comunes de esta unidad

Vamos a repasar los tropiezos clásicos para que los reconozcas cuando te pasen (porque te van a pasar):

1. Olvidar los paréntesis al llamar: escribes `saludar` en vez de `saludar()` y "no pasa nada". Sin paréntesis no hay llamada; solo obtienes el objeto función. Si tu función "no hace nada", revisa los paréntesis primero.
2. Olvidar el `return` y recibir `None`: la función calcula todo perfecto por dentro, pero afuera recibes `None`. Cada vez que veas un `None` inesperado, pregunta: "¿mi función devuelve el valor con `return`, o solo lo imprime/calcula por dentro?".
3. Confundir `print` con `return`: `print` muestra en pantalla para humanos; `return` entrega el valor al programa. Una función que solo imprime no te da nada reutilizable. Diseña funciones que devuelvan valores y deja que quien las llama decida si imprimirlos.
4. Parámetros en orden equivocado: `restar(4, 10)` en vez de `restar(10, 4)` no da error, da un resultado incorrecto silencioso, que es peor. Usa nombres de parámetros descriptivos y, con más de dos parámetros, argumentos por nombre.
5. Usar variables locales fuera de la función: las variables creadas dentro de una función mueren cuando termina. Si necesitas el valor afuera, devuélvelo con `return`. No intentes "rescatarlo" con `global`; rediseña con parámetros y `return`.

6. El valor por defecto mutable: `def funcion(lista=[])` comparte la misma lista entre todas las llamadas. Usa `lista=None` y crea la lista dentro. Este error es traicionero porque el programa "casi" funciona.
7. Definir la función después de llamarla: Python lee de arriba hacia abajo; si la llamada está antes del `def`, `NameError`. Define primero, llama después.
8. Comprension ilegible de tres líneas: si tu comprension necesita dos lecturas para entenderse, escríbela como `for`. La brevedad sin claridad no es elegancia, es presunción.

Buenas prácticas

Cierra la unidad con los hábitos que distinguen al profesional:

- Una función = una tarea. Si tu función "calcula el total, aplica el descuento, genera el ticket y lo manda por correo", son cuatro funciones disfrazadas de una. Divídelas. Una buena señal: si no puedes describir lo que hace la función en una frase corta sin usar la palabra "y", está haciendo demasiado.
- Nombres con verbos, descriptivos. `calcular_total()` dice más que `ct()` o `funcion1()`. El nombre debe responder "¿qué hace?" sin necesidad de leer el código.
- Docstring en cada función. Mínimo una línea que diga qué hace; idealmente parámetros y valor devuelto. Tu yo del futuro te lo agradecerá.
- Funciones cortas: idealmente no más de ~20 líneas. Si crece más, probablemente esconde dos o tres funciones adentro. Las funciones cortas se entienden, se prueban y se reutilizan fácil.
- Parámetros y `return` como único canal de comunicación. Evita leer y sobre todo modificar variables globales. Funciones transparentes: todo entra por parámetros, todo sale por `return`.
- No te repitas (DRY). La segunda vez que copies un bloque de código, conviértelo en función. La tercera vez ya es un pecado.
- Prefiere `return` sobre `print` dentro de funciones. Devuelve valores; deja la impresión a quien llama. Así tus funciones sirven en programas de consola, en páginas web y en automatizaciones.
- Comprensions para transformaciones simples, `for` para lógica compleja. La legibilidad siempre le gana a la brevedad.

Ejemplos paso a paso

Ejemplo 1: Calculadora con 4 funciones

La calculadora es el "hola mundo" de las funciones: cuatro operaciones, cuatro funciones, cero repetición.

```
def sumar(a, b):
    """Devuelve la suma de dos números."""
    return a + b

def restar(a, b):
    """Devuelve la resta de dos números."""
    return a - b

def multiplicar(a, b):
    """Devuelve la multiplicación de dos números."""
    return a * b

def dividir(a, b):
    """Devuelve la división; maneja la división entre cero."""
    if b == 0:
        return "Error: no se puede dividir entre cero"
    return a / b

print(sumar(10, 5))      # 15
print(restar(10, 5))    # 5
print(multiplicar(10, 5)) # 50
print(dividir(10, 5))   # 2.0
print(dividir(10, 0))   # Error: no se puede dividir entre cero
```

Línea por línea: cada def crea una función con dos parámetros (a y b) y un docstring de una línea. El cuerpo es una sola línea con return: nada de print adentro, porque queremos el valor para reutilizarlo. En dividir hay una guarda: si b es cero, devolvemos un mensaje de error y salimos temprano; si no, devolvemos la división normal. Las cinco llamadas finales muestran que cada función es un "botón" independiente: la defines una vez y la presionas cuando quieras, con los números que quieras. Si mañana quieres agregar potencia o raíz cuadrada, agregas una función más sin tocar las existentes.

Ejemplo 2: Validador de contraseña como función reutilizable

Aquí aplicamos descomposición: un validador grande dividido en mini-funciones que la directora combina.

```
def tiene_longitud(clave):
    """True si la clave tiene al menos 8 caracteres."""
    return len(clave) >= 8

def tiene_numero(clave):
    """True si la clave contiene al menos un dígito."""
    for c in clave:
        if c.isdigit():
            return True
    return False

def tiene_mayuscula(clave):
    """True si la clave contiene al menos una mayúscula."""
    for c in clave:
        if c.isupper():
            return True
    return False

def es_segura(clave):
    """True solo si cumple los tres requisitos."""
    return tiene_longitud(clave) and tiene_numero(clave) and tiene_mayuscula(clave)

for prueba in ["abc", "secreta123", "Secreta123", "Clave$2024"]:
    print(prueba, "->", "Segura" if es_segura(prueba) else "Débil")
```

Línea por línea: `tiene_longitud` usa `len()` y devuelve directamente el resultado de la comparación (`>=` devuelve `True` o `False`, no necesitas un `if`). `tiene_numero` recorre cada carácter: si encuentra un dígito, devuelve `True` de inmediato (salida temprana); si termina el ciclo sin encontrar nada, devuelve `False`. `tiene_mayuscula` hace lo mismo con `isupper()`. La directora `es_segura` combina las tres con `and`: basta que una falle para que todo sea `False`. El ciclo final prueba cuatro contraseñas y muestra el veredicto. ¿Ves la ventaja? Cada regla se prueba por separado, y agregar una regla nueva ("debe tener un símbolo") es agregar una mini-función y un `and` más, sin tocar lo que ya funciona.

Ejemplo 3: Función que devuelve estadísticas de una lista

Return múltiple con desempaqueado: una función que analiza ventas y devuelve todo junto.

```
def estadisticas(ventas):
    """Devuelve total, promedio, máximo y mínimo de una lista de ventas."""
    total = sum(ventas)
    promedio = total / len(ventas)
    return total, promedio, max(ventas), min(ventas)

ventas_semana = [1500, 2300, 1800, 3100, 2700]
total, promedio, mayor, menor = estadisticas(ventas_semana)

print(f"Total: ${total}")
print(f"Promedio: ${promedio:.2f}")
print(f"Venta mayor: ${mayor}")
print(f"Venta menor: ${menor}")
```

Línea por línea: la función recibe la lista y calcula total con `sum()` y promedio dividiendo entre `len()`. El `return` devuelve cuatro valores separados por comas: Python los empaqueta en una tupla automáticamente. En la llamada, desempaquetamos directamente en cuatro variables con nombres claros: `total`, `promedio`, `mayor`, `menor`. Los `print` usan f-strings, y `{promedio:.2f}` formatea a dos decimales. Este patrón (una función que resume datos y devuelve varias medidas) lo vas a usar constantemente en análisis de datos.

Ejemplo 4: Menú interactivo con funciones

El ejemplo que une todo: cada opción del menú es una función, y el programa principal solo las llama.

```

def mostrar_menu():
    """Muestra las opciones disponibles."""
    print("\n--- MENÚ ---")
    print("1. Saludar")
    print("2. Calcular doble")
    print("3. Salir")

def saludar_usuario():
    """Pide el nombre y saluda."""
    nombre = input("¿Cómo te llamas? ")
    print(f"¡Hola, {nombre}!")

def calcular_doble():
    """Pide un número y muestra su doble."""
    numero = float(input("Dame un número: "))
    print(f"El doble de {numero} es {numero * 2}")

def main():
    """Bucle principal del programa."""
    while True:
        mostrar_menu()
        opcion = input("Elige una opción: ")
        if opcion == "1":
            saludar_usuario()
        elif opcion == "2":
            calcular_doble()
        elif opcion == "3":
            print("¡Nos vemos!")
            break
        else:
            print("Opción inválida, intenta de nuevo.")

main()

```

Línea por línea: primero definimos tres funciones pequeñas, cada una con UNA tarea (mostrar el menú, saludar, calcular el doble). Ninguna sabe de la existencia de las otras. Luego main() es la directora: un while True que muestra el menú, lee la opción y llama a la función correspondiente; la opción 3 rompe el ciclo con break. Al final, una sola llamada a main() arranca todo. Fíjate en la legibilidad del main: se lee como la descripción del programa en español. Si quieres agregar una opción 4, escribes su función y agregas un elif: el resto no se toca. Así se construyen programas que crecen sin romperse.

Ejemplo 5: List comprehensions en acción (cuadrados, pares y temperaturas)

Tres transformaciones típicas, cada una en una línea, con su equivalente en for para que veas la diferencia.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 1. Cuadrados de todos los números
cuadrados = [n ** 2 for n in numeros]

# 2. Solo los números pares
pares = [n for n in numeros if n % 2 == 0]

# 3. Temperaturas: de Celsius a Fahrenheit, solo las positivas
celsius = [-5, 0, 10, 20, 30]
fahrenheit = [c * 9/5 + 32 for c in celsius if c > 0]

print(cuadrados)    # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
print(pares)        # [2, 4, 6, 8, 10]
print(fahrenheit)   # [50.0, 68.0, 86.0]
```

Línea por línea: el caso 1 es transformación pura ("el cuadrado de cada número"). El caso 2 es filtro puro (el if al final deja pasar solo los pares). El caso 3 combina ambas: transforma (la fórmula de conversión) y filtra (solo temperaturas positivas; el -5 y el 0 se quedan fuera, por eso fahrenheit tiene solo 3 elementos). Compara con el for equivalente del caso 3: serían 4 líneas con lista vacía, for, if y append. La comprehension dice lo mismo en una línea que se lee de corrido. Recuerda la regla: como cada línea se entiende de un vistazo, aquí SÍ conviene la comprehension.

Ejemplo 6: *args para sumar N números (y un extra con **kwargs)

Flexibilidad total: funciones que aceptan cualquier cantidad de argumentos.

```
def sumar_todo(*args):
    """Suma cualquier cantidad de números."""
    return sum(args)

def mostrar_ticket(cliente, **productos):
    """Imprime un ticket con los productos comprados."""
    print(f"--- Ticket de {cliente} ---")
    total = 0
    for producto, precio in productos.items():
        print(f"{producto}: ${precio}")
        total += precio
    print(f"TOTAL: ${total}")

print(sumar_todo(10, 20))          # 30
print(sumar_todo(1, 2, 3, 4, 5))  # 15
mostrar_ticket("Doña Lupe", tortillas=25, frijoles=30, queso=60)
```

Línea por línea: `sumar_todo` recibe `*args`, que dentro de la función es una tupla con todos los números; `sum(args)` los suma, funcione con 2 o con 20 argumentos. `mostrar_ticket` combina un parámetro normal (`cliente`) con `**kwargs`: cada argumento por nombre (`tortillas=25`) entra al diccionario `productos`. El `for` recorre `productos.items()` imprimiendo cada línea del ticket y acumulando el total. La salida del ticket queda:

```
--- Ticket de Doña Lupe ---
tortillas: $25
frijoles: $30
queso: $60
TOTAL: $115
```

Este ejemplo muestra por qué `*args` y `**kwargs` son tan queridos: `sumar_todo` sirve para cualquier cantidad de números y `mostrar_ticket` para cualquier cantidad de productos, sin cambiar la definición.

Práctica

Intenta resolver cada ejercicio antes de mirar la solución. Si te atorras más de quince minutos, lee la solución, entiéndela, ciérrala y resuélvelo tú solo.

Ejercicio 1

Escribe una función `es_par(numero)` que devuelva `True` si el número es par y `False` si es impar. Luego úsala para contar cuántos números pares hay en la lista `[3, 8, 12, 5, 7, 10, 2]`.

Solución explicada paso a paso:

```
def es_par(numero):  
    """Devuelve True si el número es par."""  
    return numero % 2 == 0  
  
numeros = [3, 8, 12, 5, 7, 10, 2]  
conteo = 0  
for n in numeros:  
    if es_par(n):  
        conteo += 1  
  
print(f"Hay {conteo} números pares")    # Hay 4 números pares
```

Paso 1: la función recibe `numero` y devuelve directamente el resultado de la comparación `numero % 2 == 0`. No necesitas un `if` que devuelva `True` o `False`: la comparación YA es un booleano. Paso 2: recorremos la lista y por cada número llamamos a `es_par(n)`; si devuelve `True`, aumentamos el conteo. Paso 3: imprimimos el resultado. La moraleja: una función pequeña y clara (`es_par`) hace que el ciclo principal se lea solo.

Ejercicio 2

Escribe una función `calcular_propina(cuenta, porcentaje=10)` que devuelva el monto de la propina. El porcentaje debe ser opcional (por defecto 10%). Pruébala con una cuenta de \$500 sin especificar porcentaje y con una de \$800 al 15%.

Solución explicada paso a paso:

```
def calcular_propina(cuenta, porcentaje=10):  
    """Devuelve el monto de la propina según el porcentaje."""  
    return cuenta * porcentaje / 100  
  
print(calcular_propina(500))          # 50.0  (usa el 10% por defecto)  
print(calcular_propina(800, 15))     # 120.0 (usa el 15% dado)  
print(calcular_propina(800, porcentaje=20)) # 160.0 (argumento por nombre)
```


Paso 1: definimos la función con porcentaje=10 como valor por defecto; recuerda que los parámetros con defecto van después de los obligatorios. Paso 2: el cuerpo es una sola línea con return: cuenta por porcentaje entre 100. Paso 3: la primera llamada omite el porcentaje y usa el 10% por defecto (como el agua de jamaica incluida). Paso 4: la segunda llamada pasa 15 y lo sobrescribe. Paso 5 (extra): mostramos que también puedes usar el argumento por nombre para máxima claridad.

Ejercicio 3

Escribe una función rectangulo(base, altura) que devuelva el área Y el perímetro al mismo tiempo. Desempaqueta los valores al llamarla e imprímelos con etiquetas.

Solución explicada paso a paso:

```
def rectangulo(base, altura):
    """Devuelve el área y el perímetro de un rectángulo."""
    area = base * altura
    perimetro = 2 * (base + altura)
    return area, perimetro

a, p = rectangulo(5, 3)
print(f"Área: {a}, Perímetro: {p}")    # Área: 15, Perímetro: 16
```

Paso 1: calculamos el área (base por altura) y el perímetro (dos veces la suma de los lados) en variables locales. Paso 2: devolvemos ambos valores separados por coma; Python los empaqueta en una tupla. Paso 3: al llamar, desempaquetamos directamente en a y p en una sola línea. Paso 4: imprimimos con etiquetas claras. Si la función devolviera muchos valores más, considera devolver un diccionario en su lugar.

Ejercicio 4

Dada la lista de palabras ["sol", "computadora", "mar", "programación", "luz", "teclado"], crea con list comprehensions: a) una lista con las palabras en mayúsculas; b) una lista solo con las palabras de más de 4 letras; c) una lista que diga "larga" o "corta" para cada palabra (más de 4 letras = larga).

Solución explicada paso a paso:

```

palabras = ["sol", "computadora", "mar", "programación", "luz", "teclado"]

# a) Transformación pura
mayusculas = [p.upper() for p in palabras]

# b) Filtro con if al final
largas = [p for p in palabras if len(p) > 4]

# c) Elección con if/else al inicio
etiquetas = ["larga" if len(p) > 4 else "corta" for p in palabras]

print(mayusculas)      # ['SOL', 'COMPUTADORA', 'MAR', 'PROGRAMACIÓN', 'LUZ', 'TECLADO']
print(largas)          # ['computadora', 'programación', 'teclado']
print(etiquetas)       # ['corta', 'larga', 'corta', 'larga', 'corta', 'larga']

```

Paso a): transformación pura, sin filtro: se lee "la mayúscula de cada palabra". Paso b): agregamos `if len(p) > 4` AL FINAL como portero que solo deja pasar palabras largas. Paso c): el `if/else` va AL INICIO porque forma parte de la expresión (qué valor se guarda), no del filtro: todas las palabras entran, pero cada una se etiqueta según su longitud. Compara b) y c): mismo `if`, distinto lugar, distinto significado.

Ejercicio 5

Escribe una función `calificacion_final(*parciales)` que reciba cualquier cantidad de calificaciones parciales y devuelva el promedio redondeado a 1 decimal. Si no recibe ninguna, debe devolver 0. Pruébala con 3 y con 5 calificaciones.

Solución explicada paso a paso:

```

def calificacion_final(*parciales):
    """Devuelve el promedio de las calificaciones recibidas."""
    if len(parciales) == 0:
        return 0
    return round(sum(parciales) / len(parciales), 1)

print(calificacion_final(9, 8, 10))      # 9.0
print(calificacion_final(7, 8, 6, 9, 10)) # 8.0
print(calificacion_final())              # 0

```

Paso 1: *parciales empaqueta todos los argumentos en una tupla, así la función acepta 3, 5 o 100 calificaciones. Paso 2: si no llegó ninguna (tupla vacía), devolvemos 0 de inmediato para evitar dividir entre cero: es la "salida temprana". Paso 3: en el caso normal, sumamos con sum(), dividimos entre la cantidad y redondeamos con round(..., 1). Paso 4: probamos con distintas cantidades, incluyendo el caso vacío. Este es el poder de *args: una sola función para todos los casos.

Ejercicio 6

Arma un mini-sistema de tienda con descomposición: una función precio_con_iva(precio), una función total_compra(*precios) que sume los precios con IVA usando la anterior, y una función mostrar_ticket(*precios) que imprima cada precio con IVA y el total. Usa IVA de 16%.

Solución explicada paso a paso:

```
def precio_con_iva(precio, iva=0.16):
    """Devuelve el precio con IVA incluido."""
    return precio * (1 + iva)

def total_compra(*precios):
    """Devuelve el total de la compra con IVA."""
    return sum(precio_con_iva(p) for p in precios)

def mostrar_ticket(*precios):
    """Imprime el ticket desglosado y el total."""
    print("--- TICKET ---")
    for p in precios:
        print(f"Producto: ${precio_con_iva(p):.2f}")
    print(f"TOTAL: ${total_compra(*precios):.2f}")

mostrar_ticket(100, 250, 80)
```

Paso 1: precio_con_iva es la pieza base: un cálculo simple con el IVA como parámetro por defecto. Paso 2: total_compra LLAMA a precio_con_iva para cada precio; fíjate que usa una generator expression dentro de sum() (prima de la list comprehension, sin corchetes), elegante y eficiente. Paso 3: mostrar_ticket también reutiliza precio_con_iva para cada línea y llama a total_compra para el total; nota el *precios al pasar los argumentos: desempaqueta la tupla de vuelta en argumentos individuales. Paso 4: la llamada final produce el ticket completo. La salida:

```
--- TICKET ---  
Producto: $116.00  
Producto: $290.00  
Producto: $92.80  
TOTAL: $498.80
```

La lección del ejercicio: el cálculo del IVA vive en UN solo lugar. Si mañana el IVA cambia a 18%, modificas una línea y todo el sistema se actualiza. Eso es DRY en acción.

Resumen de la unidad

En esta unidad aprendiste la herramienta que separa al aficionado del profesional: las funciones, más un toque de elegancia con las list comprehensions. Esto es lo esencial que debes llevarte grabado:

- Una función es un bloque de código con nombre que defines una vez con `def` y llamas cuantas veces quieras con su nombre más paréntesis. Es tu tarjeta de receta: la escribes una vez, la usas siempre.
- El principio DRY ("no te repitas"): cada vez que copies un bloque de código, conviértelo en función. Un solo punto de verdad significa un solo lugar que corregir.
- Los parámetros son los "lugares vacíos" en la definición; los argumentos son los valores reales en la llamada. El orden importa cuando los pasas por posición.
- `return` termina la función y devuelve un valor a quien la llamó. Sin `return`, la función devuelve `None`. Diseña funciones que devuelvan valores (`return`) en vez de solo mostrarlos (`print`).
- Puedes devolver varios valores separándolos por comas (se empaquetan en tupla) y desempaquetarlos al recibirlos.
- Los parámetros por defecto (como `saludo="Hola"`) evitan repetición; van después de los obligatorios. Nunca uses listas o diccionarios como valor por defecto: usa `None` y crea el objeto dentro.
- Los argumentos por nombre (`especie="gato"`) dan claridad y permiten cualquier orden. `*args` recibe N argumentos en una tupla; `**kwargs` recibe N argumentos por nombre en un diccionario.
- Scope: las variables locales viven solo dentro de la función; las globales se pueden leer pero evita modificarlas con `global`. Comunica tus funciones solo con parámetros y `return`.

- Documenta cada función con un docstring entre triples comillas; consúltalo con `help()`.
- Descompón problemas grandes en funciones pequeñas: una función directora que llama a mini-funciones. Una función = una tarea, con nombre de verbo, idealmente menos de 20 líneas.
- Las list comprehensions crean listas en una línea: `[expresión for x in iterable]`, con `if` al final para filtrar y con `if/else` al inicio para elegir. Úsalas cuando se entiendan de un vistazo; si necesitas leerlas dos veces, escribe un `for`.
- También existen `dict` y `set` comprehensions para casos especiales.
- Funciones integradas que te ahorran trabajo: `sum`, `min`, `max`, `sorted`, `enumerate` y `zip`. Úsalas antes de reinventar la rueda.
- Errores clásicos a evitar: olvidar los paréntesis al llamar, olvidar el `return`, confundir `print` con `return`, parámetros en desorden, usar locales fuera de la función y comprehensions ilegibles.

Con funciones, tus programas dejan de ser una lista interminable de instrucciones y se convierten en historias que se leen solas: pedir datos, validar, calcular, mostrar. Esa es la diferencia entre código que jala y código profesional. En la siguiente unidad seguiremos subiendo de nivel. ¡A programar!

Esteban lucasegaacademy.com

UNIDAD 9: Que nada te tumbe: errores, archivos y módulos

Introducción

Hasta ahora, tus programas han vivido en una burbuja muy cómoda: se ejecutan, hacen sus cálculos, muestran sus resultados y se apagan. Si el usuario escribe exactamente lo que esperas, todo funciona de maravilla. Pero la vida real no es así. En la vida real el usuario escribe letras donde esperabas números, divide entre cero, abre un archivo que no existe y apaga la computadora sin avisar. Y cuando eso pasa, tu programa hace lo más vergonzoso que un programa puede hacer: se rompe en la cara del usuario con un mensaje de error en inglés que nadie entiende.

Esta unidad es la que separa a los programas de juguete de los programas de verdad. Vas a aprender tres superpoderes que todo programador profesional domina. Primero, el manejo de errores: cómo atrapar los errores antes de que rompan tu programa y responder con elegancia, como un mesero que derrama el agua y en vez de entrar en pánico dice "permítame limpiarlo de inmediato". Segundo, los archivos: cómo leer y escribir archivos de verdad en tu disco duro, para que tus datos sobrevivan cuando apagas la computadora. Tercero, los módulos: cómo importar código que otros ya escribieron (y que funciona) para no reinventar la rueda cada vez.

Piensa en un cinturón de seguridad. Cuando te subes al coche, no te lo pones porque esperes chocar: te lo pones porque sabes que los choques pasan, y cuando pasan, la diferencia entre un susto y una tragedia es ese cinturón. El manejo de errores es el cinturón de seguridad de tus programas. No lo pones porque tu código sea malo; lo pones porque los usuarios son impredecibles, los archivos desaparecen, la red se cae y los datos vienen sucios. Un programa sin manejo de errores es como manejar sin cinturón: funciona perfecto... hasta que deja de funcionar.

Y piensa en una libreta de papel. Todo lo que has guardado en variables hasta ahora es como escribir en un pizarrón: en cuanto apagas la computadora (o termina el programa), se borra todo. Las variables viven en la memoria RAM, y la RAM es olvidadiza por diseño. Los archivos, en cambio, son tu libreta: lo que escribes ahí sigue existiendo mañana, la próxima semana, el próximo año. Cuando tu programa guarda una lista de tareas en un archivo y al día siguiente la vuelve a cargar, deja de ser un truco de magia y se convierte en una herramienta real. Eso se

llama persistencia, y es la diferencia entre un programa que entretiene cinco minutos y uno que la gente usa todos los días.

Los módulos son la tercera pieza, y son un regalo. Python viene con una biblioteca estándar enorme: cientos de módulos ya escritos, probados y gratis, listos para que los uses con una sola línea de `import`. Necesitas números aleatorios para un juego: `import random`. Necesitas matemáticas avanzadas: `import math`. Necesitas la fecha de hoy: `import datetime`. Y cuando la biblioteca estándar no basta, existe `pip`, una herramienta para instalar miles de paquetes más que la comunidad regaló al mundo. Programar en Python es un poco como cocinar con una alacena ya llena: no tienes que sembrar el trigo para hacer pan.

Esta unidad agrupa las clases 24, 25 y 26 del curso de video. En la clase 24 aprendiste `try / except` para que el programa no se rompa. En la 25, `open` y `with` para leer y escribir archivos de verdad. En la 26, `import` para traer superpoderes listos. Aquí vamos a profundizar cada tema hasta que lo domines de verdad, con la filosofía detrás, los errores típicos que te van a salir (y cómo leerlos sin miedo), ejemplos paso a paso que puedes correr en tu computadora y práctica con soluciones explicadas.

Una advertencia honesta: esta unidad cambia tu relación con los errores. Hasta ahora, un error era una señal de que algo estaba mal en tu código. A partir de ahora, vas a ver los errores como lo que realmente son: mensajes. Python te está hablando cada vez que algo falla, te dice exactamente qué pasó y dónde. El programador novato ve un error y se asusta; el profesional lo lee como quien lee una nota. Al terminar esta unidad, tú vas a leer los errores como un profesional.

Otro aviso: aquí vas a trabajar con archivos de verdad en tu computadora, así que ten tu editor y tu terminal a la mano. Crea una carpeta para los ejercicios de esta unidad, algo como `unidad9`, y guarda ahí tus programas y tus archivos de prueba. Ver cómo un archivo `.txt` aparece en tu carpeta, escrito por tu propio programa, es una de esas pequeñas magias que nunca se olvidan.

Vamos a empezar por lo más importante: cambiar tu mentalidad sobre los errores. Porque antes de atrapar errores con código, tienes que dejar de tenerles miedo.

Explicación detallada

Los errores son mensajes, no fracasos

Te voy a decir algo que ojalá alguien te hubiera dicho el primer día que programaste: que tu programa tenga un error no significa que seas mal programador. Significa que estás programando. Los errores son parte del oficio, como las astillas para el carpintero. Hasta los programadores con veinte años de experiencia ven errores todos los días. La diferencia no está en quién los evita, sino en quién los entiende más rápido.

Python tiene una forma muy particular de avisarte que algo salió mal: lanza una excepción. Una excepción es un evento que interrumpe el flujo normal del programa porque algo no se pudo hacer. Cuando Python encuentra un problema que no sabe resolver solo, crea un objeto de excepción con información sobre lo que pasó y lo "lanza" hacia arriba, esperando que alguien lo atrape. Si nadie lo atrapa, el programa se detiene y Python imprime un reporte del error en la terminal. Ese reporte se llama traceback, y en un momento te enseñó a leerlo como un mapa del tesoro.

Aquí está la idea clave de esta unidad: tú puedes atrapar esas excepciones antes de que rompan tu programa. Con `try / except` le dices a Python: "intenta hacer esto, y si algo sale mal, no te mueras: haz esto otro". Es como ponerle una red de seguridad a un trapecista. El trapecista sigue haciendo su acto igual de arriesgado, pero si falla, cae en la red en vez de contra el suelo.

Hay dos tipos de errores que debes distinguir desde ya. Los errores de sintaxis son errores de escritura: te faltó un paréntesis, olvidaste los dos puntos, escribiste mal una palabra clave. Python los detecta antes de ejecutar nada y no hay `try / except` que los atrape, porque el programa ni siquiera arranca. Esos se arreglan leyendo con cuidado y con la práctica. Los errores de ejecución (también llamados errores en tiempo de ejecución) son los que ocurren mientras el programa corre: dividir entre cero, abrir un archivo que no existe, convertir a número un texto que no es número. Esos sí se pueden atrapar, y son los protagonistas de esta unidad.

Piénsalo con una analogía de cocina. Un error de sintaxis es como escribir mal una receta: "agrega dos tazas de arina" - el que la lee ni siquiera entiende qué hacer. Un error de ejecución es como seguir la receta perfecta pero descubrir a mitad del proceso que no hay huevos en el refri: la receta estaba bien escrita, pero la realidad no cooperó. El `try / except` es tu plan B para cuando no hay huevos: "si no hay huevos, haz hot cakes sin huevo en vez de quemar la cocina".

Cómo leer un traceback sin entrar en pánico

Cuando un programa de Python se rompe sin que nadie atrape el error, la terminal muestra algo como esto:

```
Traceback (most recent call last):  
  File "/home/usuario/unidad9/programa.py", line 5, in <module>  
    resultado = 10 / 0  
    ~~~^~~  
ZeroDivisionError: division by zero
```

Para un principiante, eso se ve como jeroglíficos amenazantes. Para ti, a partir de hoy, es un mensaje claro con tres partes. Vamos a desarmarlo.

La primera línea, `Traceback (most recent call last)`, solo anuncia: "aquí viene el reporte del error, de la llamada más reciente al final". Es el encabezado, ignóralo después de leerlo una vez en tu vida.

Las líneas del medio son la ruta del error: te dicen qué líneas se estaban ejecutando cuando todo se rompió, de afuera hacia adentro. En el ejemplo hay una sola: el archivo `programa.py`, línea 5, dentro del módulo principal. Si tu programa tuviera funciones llamándose entre sí, verías varias líneas, como un rastro de migas de pan que te lleva desde donde empezó todo hasta donde explotó. Se lee de arriba hacia abajo, y la última línea de esta sección es donde ocurrió el problema de verdad.

La última línea es la más importante: el tipo de error y su mensaje. `ZeroDivisionError: division by zero`. Te dice el nombre del error (división entre cero) y un mensaje que lo explica. Esta línea es tu mejor amiga: te dice exactamente qué salió mal.

El truco profesional para leer tracebacks es este: empieza por la última línea (qué pasó), luego sube a la ruta (dónde pasó, archivo y número de línea). El 90% de las veces, con esos dos datos ya sabes qué arreglar. Los programadores experimentados ni siquiera leen todo el traceback: van directo al final, ven `NameError: name 'contador' is not defined`, y piensan "ah, escribí mal el nombre de la variable o la usé antes de crearla".

Otro detalle: Python a veces subraya con `~~~^~~` la parte exacta de la línea donde detectó el problema. Es una pista visual muy útil, aunque no siempre apunta al verdadero origen (a veces el problema real está unos caracteres antes). Úsala como aproximación, no como verdad absoluta.

Acostúmbrate a leer los tracebacks en voz alta, como si Python te estuviera hablando: "Oye, en la línea 5 de programa.py intentaste dividir entre cero y eso no se puede". Cuando lo lees así, deja de dar miedo y se convierte en una conversación. Python no te está regañando; te está ayudando a encontrar el problema. Créeme: preferirías mil veces un lenguaje que te dice exactamente qué pasó y dónde, que uno que se rompe en silencio y te deja adivinando.

Los errores más comunes (conócelos por su nombre)

Python tiene docenas de tipos de excepción, pero en la práctica te vas a topar con los mismos ocho una y otra vez. Conocerlos por su nombre es como conocer a los vecinos problemáticos de tu colonia: cuando uno aparece, ya sabes quién es y qué quiere. Vamos uno por uno, con ejemplo de código que lo provoca y su explicación.

SyntaxError: te equivocaste escribiendo. Ocurre cuando el código ni siquiera tiene la forma correcta de Python. El ejemplo clásico es olvidar los dos puntos:

```
if edad >= 18
    print("Eres mayor de edad")
```

Python responde `SyntaxError: expected ':'`. No hay `try / except` que valga aquí: el programa no arranca. La solución es leer la línea que te marca y buscar el error de escritura: paréntesis sin cerrar, comillas sin cerrar, dos puntos olvidados, indentación mal puesta. Con la práctica, estos errores casi desaparecen solos.

NameError: usaste un nombre que no existe. Pasa cuando escribes mal el nombre de una variable o la usas antes de crearla:

```
nombre = "Ana"
print(nmbre)
```

`NameError: name 'nmbre' is not defined`. ¿Ves? Escribí `nmbre` en vez de `nombre`. Python te dice el nombre exacto que no reconoce. También ocurre si intentas usar una variable fuera del lugar donde fue creada, tema que ya dominas de unidades anteriores. El remedio: revisa la ortografía del nombre y verifica que la variable exista antes de usarla.

TypeError: mezclaste tipos incompatibles. Python es estricto con los tipos: no puedes sumar un número con un texto, por ejemplo:

```
edad = 25
mensaje = "Tienes " + edad + " años"
```

TypeError: can only concatenate str (not "int") to str. Traducción: "solo puedes concatenar texto con texto, no con un entero". La solución es convertir: "Tienes"+ str(edad) + "años", o mejor aún, usar un f-string: f"Tienes {edad} años". Este error es muy común cuando olvidas que input() siempre devuelve texto y tratas de operar con el resultado como si fuera número.

ValueError: el tipo está bien, pero el valor no sirve. Es primo hermano del anterior, pero más sutil: el dato es del tipo correcto, pero su contenido no tiene sentido para la operación:

```
numero = int("hola")
```

ValueError: invalid literal for int() with base 10: 'hola'. Le pediste a Python convertir el texto "hola" a número entero, y "hola" no es un número. El tipo era texto (correcto para int()), pero el valor no se puede convertir. Este es EL error que vas a atrapar con try / except cuando el usuario escriba tonterías donde esperabas un número. Grábatelo bien, porque lo verás muchísimo.

IndexError: te saliste de la lista. Ocurre cuando pides una posición que no existe:

```
frutas = ["manzana", "pera", "uva"]
print(frutas[5])
```

IndexError: list index out of range. La lista tiene 3 elementos (posiciones 0, 1 y 2) y pediste la posición 5. Es como pedir la habitación 500 en un hotel de 3 pisos. Suele pasar en ciclos o cuando asumes que una lista tiene más elementos de los que realmente tiene. La defensa: verifica la longitud con len() antes de acceder, o atrapa la excepción.

KeyError: la llave no existe en el diccionario. El equivalente del anterior pero para diccionarios:

```
alumno = {"nombre": "Luis", "edad": 20}
print(alumno["calificacion"])
```

KeyError: 'calificacion'. Pediste una llave que el diccionario no tiene. La defensa profesional es usar el método get(), que en vez de romperse devuelve None (o un valor por defecto que tú elijas): alumno.get("calificacion", "sin calificar").

ZeroDivisionError: dividiste entre cero. Las matemáticas lo prohíben y Python también:

```
promedio = suma / 0
```

ZeroDivisionError: division by zero. Aparece mucho en programas que calculan promedios o porcentajes cuando el divisor viene de datos del usuario o de una lista que resultó estar vacía. La defensa: verifica que el divisor no sea cero antes de dividir, o atrapa la excepción.

FileNotFoundError: el archivo no existe. El protagonista de la sección de archivos:

```
archivo = open("datos_secretos.txt")
```

FileNotFoundError: [Errno 2] No such file or directory: 'datos_secretos.txt'. Intentaste abrir un archivo que no está donde dijiste. Pasa por escribir mal el nombre, por estar en la carpeta equivocada o porque el archivo se borró. Es uno de los errores que más vas a atrapar con `try / except` en programas reales.

Memoriza estos ocho nombres. No de memoria ciega, sino reconociéndolos cuando aparezcan. Cada vez que Python te lance uno, en vez de frustrarte, piensa: "ah, es mi vecino el `ValueError` otra vez, ya sé qué quiere". Ese cambio de actitud vale oro.

try/except: el cinturón de seguridad

Ya entiendes qué son los errores y cómo te hablan. Ahora vamos a lo práctico: cómo evitar que rompan tu programa. La herramienta es el bloque `try / except`, y su sintaxis es sencilla:

```
try:
    numero = int(input("Dame un número: "))
    print(f"El doble es {numero * 2}")
except ValueError:
    print("Eso no es un número válido, intenta de nuevo.")
```

Esto es exactamente lo que viste en la clase 24 del curso. Vamos a desarmarlo con calma. El bloque `try:` contiene el código "peligroso": el que podría fallar. Python lo ejecuta con normalidad. Si todo sale bien, el bloque `except` se ignora por completo y el programa sigue como si nada. Pero si dentro del `try` ocurre un error del tipo que indica el `except` (en este caso, `ValueError`), Python no se rompe: salta inmediatamente al bloque `except` y ejecuta su código. El programa sobrevive.

Fíjate en el detalle crucial: si el error ocurre en la línea `numero = int(...)`, las líneas que siguen dentro del `try` ya no se ejecutan. Python salta directo al `except`. Es como si el trapecista fallara a mitad del salto: no termina el salto, cae en la red. Por eso el código dentro del `try` debe ser solo lo necesario: entre menos código "peligroso" metas, más claro es qué pudo fallar.

Sin el `try / except`, si el usuario escribe "tacos" donde esperabas un número, el programa muere con un traceback feo. Con el `try / except`, el programa responde con un mensaje amable y sigue vivo. Esa es toda la diferencia entre un programa amateur y uno profesional: el profesional anticipa que las cosas van a salir mal y tiene un plan.

Un punto importante: `except` solo atrapa el error si ocurre dentro de su `try` correspondiente. Y solo atrapa el tipo de error que declares (o sus subtipos). Si dentro del `try` ocurre un error de otro tipo, no lo atrapa y el programa se rompe igual. Eso es bueno: no quieres atrapar errores que no esperabas, porque entonces ocultarías bugs de tu propio código.

Excepciones específicas vs. el except genérico

Puedes escribir `except` de varias formas, y la diferencia importa mucho:

```
# Forma 1: atrapar un tipo específico (RECOMENDADA)
try:
    numero = int(input("Dame un número: "))
except ValueError:
    print("Eso no es un número válido.")
```

```
# Forma 2: atrapar varios tipos específicos
try:
    resultado = 10 / int(input("Dame un número: "))
except ValueError:
    print("Eso no es un número válido.")
except ZeroDivisionError:
    print("No se puede dividir entre cero.")
```

```
# Forma 3: el except genérico (ÚSALO POCO)
try:
    numero = int(input("Dame un número: "))
except Exception as e:
    print(f"Algo salió mal: {e}")
```

La forma 1 es la profesional: atrapas exactamente el error que esperas. Si el usuario escribe "tacos", `int()` lanza `ValueError`, lo atrapas y respondes con elegancia. Si ocurriera otro error distinto (digamos, un bug en tu código), no lo atraparías y el traceback te avisaría. Eso es lo que quieres: atrapar lo predecible, dejar visible lo inesperado.

La forma 2 muestra que puedes poner varios bloques `except`, uno por cada tipo de error que anticipes. Python los prueba en orden y ejecuta el primero que coincida. Es la forma de dar respuestas distintas a problemas distintos: no es lo mismo que el usuario escriba letras a que intente dividir entre cero, y tu programa puede explicarle cada caso.

La forma 3 usa `Exception`, que es la "madre" de casi todas las excepciones: atrapar `Exception` atrapa prácticamente todo. El `as` e guarda el objeto de la excepción en la variable `e` para que puedas mostrar su mensaje. ¿Cuándo usarla? Casi nunca en código normal. El problema del `except` genérico es que oculta bugs: si tu código tiene un error de lógica que lanza un `TypeError` inesperado, el `except Exception` lo traga en silencio y tú nunca te enteras de que tu programa está mal. Es como tomar un analgésico para todo: calma el síntoma pero esconde la enfermedad.

Hay una forma 4 todavía peor, que verás en código viejo de internet:

```
# Forma 4: except desnudo (NO LA USES)
try:
    numero = int(input("Dame un número: "))
except:
    print("Algo salió mal.")
```

Un `except` sin tipo atrapa absolutamente todo, incluyendo cosas que ni siquiera son errores normales (como la señal de interrumpir el programa con Ctrl+C). Es una mala práctica reconocida. Si ves esto en tutoriales viejos, ya sabes más que el tutorial.

La regla de oro: **atrapa excepciones específicas, no genéricas**. Declara exactamente qué error esperas. Tu yo del futuro (y tus compañeros de trabajo) te lo van a agradecer.

Múltiples `except`, `else` y `finally`

El bloque `try` tiene dos compañeros opcionales que lo hacen más expresivo: `else` y `finally`. La estructura completa se ve así:

```
try:
    numero = int(input("Dame un número: "))
    resultado = 100 / numero
except ValueError:
    print("Eso no es un número válido.")
except ZeroDivisionError:
    print("No puedes dividir entre cero.")
else:
    print(f"El resultado es {resultado}")
finally:
    print("Este mensaje siempre aparece.")
```

Vamos por partes. Los múltiples `except` ya los viste: cada uno maneja un tipo de error distinto. Python evalúa los `except` en el orden en que los escribiste, así que pon primero los más específicos. (Hay una jerarquía de excepciones en Python donde algunas heredan de otras; si pones una general antes que una específica, la específica nunca se alcanzará. En la práctica, con los errores comunes que ya conoces, rara vez tendrás este problema, pero es bueno saber que el orden importa.)

El bloque `else` se ejecuta solo si el `try` terminó sin errores. Es el lugar para poner el código que depende de que todo haya salido bien. ¿Por qué no ponerlo directamente dentro del `try`? Porque así separas claramente "lo que puede fallar" (en el `try`) de "lo que hago con el resultado si no falló" (en el `else`). Es una cuestión de claridad: quien lee tu código entiende de inmediato la intención. Además, si el código del `else` tuviera su propio error, no quedaría atrapado accidentalmente por los `except` del mismo bloque.

El bloque `finally` se ejecuta siempre: haya habido error o no, se haya atrapado o no. Es el lugar para la limpieza obligatoria: cerrar conexiones, liberar recursos, despedirte. Incluso si el `except` lanza otro error o si hay un `return` en medio, el `finally` se ejecuta. Piensa en él como el "pase lo que pase, esto se hace". En la práctica moderna lo verás poco para archivos (porque `with` se encarga de eso, como verás en un momento), pero es importante que sepas que existe y para qué sirve.

Un ejemplo completo y realista que junta todo:

```

try:
    archivo = open("config.txt")
    contenido = archivo.read()
except FileNotFoundError:
    print("No encontré el archivo config.txt, usaré valores por defecto.")
    contenido = "modo=normal"
else:
    print("Configuración cargada correctamente.")
finally:
    print("Fin del intento de carga.")

```

Si el archivo existe, se lee y el `else` confirma. Si no existe, el `except` pone valores por defecto y el programa sigue funcionando. El `finally` siempre se despide. Eso es robustez: el programa se adapta a la realidad en vez de morirse.

raise: lanzar tus propios errores

Hasta ahora hemos atrapado errores que Python lanza solo. Pero tú también puedes lanzar los tuyos con la palabra `raise`. ¿Para qué querrías romper tu propio programa a propósito? Para señalar que algo viola las reglas de tu programa, aunque Python no lo considere un error.

Ejemplo: una función que calcula la raíz cuadrada no tiene sentido con números negativos (en el mundo de los números reales). Python no se queja si le pasas un negativo a tu función, pero tú sí puedes quejarte:

```

def raiz_cuadrada(n):
    if n < 0:
        raise ValueError("No puedo calcular la raíz de un número negativo.")
    return n ** 0.5

```

Si alguien llama `raiz_cuadrada(-9)`, tu función lanza un `ValueError` con un mensaje claro. Quien use tu función puede atraparlo con `try / except` igual que cualquier otro error. Es una forma de ponerle reglas a tu código: "si me das datos inválidos, te aviso en voz alta en vez de devolvarte basura en silencio".

Otro uso común es validar argumentos al inicio de una función importante:


```
def retirar(saldo, cantidad):
    if cantidad <= 0:
        raise ValueError("La cantidad a retirar debe ser positiva.")
    if cantidad > saldo:
        raise ValueError("Fondos insuficientes.")
    return saldo - cantidad
```

No profundizaremos más en `raise` en esta unidad, pero ya sabes que existe y para qué sirve: es tu forma de decir "esto no se vale" con todas las de la ley, usando el mismo sistema de excepciones de Python.

El patrón profesional: `while + try` para validar entrada

Aquí viene uno de los patrones más valiosos de toda tu carrera como programador. El problema: le pides un número al usuario, pero el usuario escribe "tacos", o "doce", o deja el campo vacío, o escribe "12.5.3". Quieres seguir preguntando hasta que te dé un número válido, sin que el programa se rompa. La solución combina un ciclo `while` con `try / except`:

```
while True:
    try:
        edad = int(input("¿Cuántos años tienes? "))
        break
    except ValueError:
        print("Por favor escribe un número entero válido.")

print(f"Tienes {edad} años.")
```

Analízalo con calma porque este patrón lo vas a usar cientos de veces. El `while True` es un ciclo infinito a propósito: seguirá preguntando para siempre... hasta que algo lo detenga. Dentro, el `try` intenta convertir la entrada a entero. Si funciona, el `break` rompe el ciclo y el programa continúa con la edad ya validada. Si el usuario escribe algo inválido, `int()` lanza `ValueError`, el `except` muestra el mensaje amable, y como no hubo `break`, el ciclo vuelve a empezar y pregunta de nuevo. El usuario no puede avanzar hasta dar un dato válido, y el programa jamás se rompe.

Este patrón se puede refinar. Por ejemplo, validar también el rango:

```

while True:
    try:
        edad = int(input("¿Cuántos años tienes? "))
        if 0 <= edad <= 120:
            break
        print("Esa edad no parece real, intenta de nuevo.")
    except ValueError:
        print("Por favor escribe un número entero válido.")

```

Ahora validas dos cosas: que sea un número y que esté en un rango sensato. Si el usuario escribe 250, no hay error de Python (250 es un entero válido), pero tu validación lógica lo rechaza con un mensaje. Fíjate cómo el `try / except` maneja los errores de Python y el `if` maneja las reglas de tu programa. Juntos forman una validación completa.

Los profesionales encapsulan este patrón en una función reutilizable:

```

def pedir_entero(mensaje):
    while True:
        try:
            return int(input(mensaje))
        except ValueError:
            print("Entrada inválida: escribe un número entero.")

edad = pedir_entero("¿Cuántos años tienes? ")
cantidad = pedir_entero("¿Cuántos boletos quieres? ")

```

Una sola función, úsala donde quieras. Cada vez que necesites un número del usuario, llamas a `pedir_entero()` y te olvidas del problema. Así se construye código profesional: identificas un problema recurrente, lo resuelves una vez y lo reutilizas siempre.

Archivos: tu programa aprende a recordar

Cambiamos de tema. Hasta ahora, todo lo que tus programas "saben" vive en variables, y las variables viven en la memoria RAM. Cuando el programa termina, la RAM se libera y todo se olvida. Es como si cada vez que cierras un documento sin guardar, el trabajo se perdiera. Los archivos son el botón de "guardar" de tus programas: te permiten escribir datos en el disco duro, donde sobreviven al apagado de la computadora, y leerlos después.

En Python, todo empieza con la función `open()`. Su forma básica es:

```
archivo = open("notas.txt", "w")
```

El primer argumento es el nombre del archivo y el segundo es el modo: qué quieres hacer con él. Los tres modos fundamentales son:

- "r" (read, lectura): abre el archivo para leerlo. Es el modo por defecto: si no especificas modo, Python asume "r". Si el archivo no existe, lanza `FileNotFoundError`.
- "w" (write, escritura): abre el archivo para escribir en él. Si el archivo no existe, lo crea. **Si ya existe, borra todo su contenido.** Este es el modo peligroso, y en un momento te explico por qué merece respeto.
- "a" (append, agregar): abre el archivo para agregar contenido al final, sin borrar lo que ya tiene. Si no existe, lo crea.

Hay más modos (como "r+" o "wb" para archivos binarios), pero con estos tres resuelves el 95% de los casos. Memoriza: **r** de **read** (leer), **w** de **write** (escribir, borrando), **a** de **append** (agregar al final).

Escribir en un archivo es así de simple:

```
archivo = open("saludo.txt", "w")
archivo.write("Hola, mundo.\n")
archivo.write("Este archivo lo escribió un programa de Python.\n")
archivo.close()
```

La función `write()` escribe texto en el archivo. Fíjate en el `\n` al final de cada línea: `write()` no agrega saltos de línea solo, así que si quieres que el texto quede en líneas separadas, tú pones el `\n`. Y el `close()` al final es obligatorio: cierra el archivo y asegura que todo lo que escribiste se guarde de verdad en el disco. (Python guarda temporalmente en un buffer y solo escribe al disco al cerrar; si olvidas el `close()`, puedes perder datos.)

Leer un archivo es igual de directo:

```
archivo = open("saludo.txt", "r")
contenido = archivo.read()
archivo.close()
print(contenido)
```

`read()` devuelve todo el contenido del archivo como un solo texto. Pero hay más formas de leer, y cada una sirve para algo distinto:

- `read()`: lee todo el archivo de una vez y lo devuelve como un string. Perfecto para archivos pequeños o medianos.
- `readline()`: lee una sola línea (hasta el siguiente salto de línea). Útil cuando quieres procesar el archivo línea por línea de forma controlada.
- `readlines()`: lee todas las líneas y las devuelve como una lista de strings, donde cada elemento es una línea (incluyendo su `\n` al final).

Ejemplo con `readlines()`:

```
archivo = open("saludo.txt", "r")
lineas = archivo.readlines()
archivo.close()
for i, linea in enumerate(lineas, 1):
    print(f"Línea {i}: {linea.strip()}")
```

El `strip()` quita el salto de línea y espacios de los extremos para mostrar limpio. Y hay un truco elegante: puedes iterar directamente sobre el archivo abierto, línea por línea, sin cargar todo en memoria:

```
archivo = open("saludo.txt", "r")
for linea in archivo:
    print(linea.strip())
archivo.close()
```

Esto es eficiente con archivos grandes, porque Python lee una línea a la vez en vez de cargar todo el archivo en la RAM. Para archivos de gigabytes, esta es la forma correcta.

with: el que cierra solo

¿Notaste que en todos los ejemplos anteriores tuve que acordarme de llamar a `close()`? Los humanos somos olvidadizos, y olvidar cerrar un archivo es una de las fuentes clásicas de bugs: datos que no se guardan, archivos bloqueados, recursos desperdiciados. Python tiene una solución elegante: la declaración `with`.

```
with open("saludo.txt", "w") as archivo:
    archivo.write("Hola, mundo.\n")
    archivo.write("Segunda línea.\n")
# Aquí el archivo YA está cerrado, automáticamente
```

Esto es lo que aprendiste en la clase 25. El `with` abre el archivo y lo asigna a la variable `archivo`, ejecuta el bloque indentado, y al terminar el bloque (o si ocurre un error dentro), cierra el archivo automáticamente. No importa qué pase: el archivo siempre queda cerrado. Es como una puerta que se cierra sola: no dependes de tu memoria.

La regla profesional es simple: ****siempre usa `with` para trabajar con archivos****. Olvídate del `open()` suelto con `close()` manual, salvo casos muy especiales. El `with` no solo es más corto: es más seguro. Incluso si tu código lanza una excepción a mitad de la escritura, el archivo se cierra correctamente.

Lectura con `with`:

```
with open("saludo.txt", "r") as archivo:
    contenido = archivo.read()
print(contenido)
```

Fíjate que uso el `contenido` fuera del `with`: la variable `contenido` sigue existiendo después de que el archivo se cerró, porque lo que guardaste en ella es el texto leído, no el archivo. El archivo ya cumplió su misión.

El peligro del modo "w": borra todo sin preguntar

Te lo advertí antes y lo repito porque es importante: el modo `"w"` **borra todo el contenido del archivo** en cuanto lo abres, sin preguntar, sin papelera de reciclaje, sin forma de recuperarlo.

Mira:

```
# El archivo "mis_notas.txt" tiene 3 años de notas importantes...
with open("mis_notas.txt", "w") as archivo:
    archivo.write("nota nueva")
# ...y ahora solo dice "nota nueva". Todo lo demás se perdió.
```

Esto no es un bug de Python: es el comportamiento diseñado del modo escritura. Pero es una trampa mortal para el programador distraído. La regla es: usa `"w"` solo cuando estés seguro de que quieres empezar el archivo desde cero (o cuando sabes que no existe). Si quieres conservar lo que hay y agregar más, usa `"a"`.

El modo `"a"` (append) es tu amigo para diarios, bitácoras y registros:

```
with open("diario.txt", "a") as archivo:
    archivo.write("Hoy aprendí a usar archivos en Python.\n")
```

Cada vez que ejecutas esto, la línea se agrega al final. El contenido anterior sigue intacto. Los programas que generan bitácoras (logs) usan "a" toda la vida: abren, agregan una línea con fecha y hora, cierran.

Un error típico de principiante: usar "w" dentro de un ciclo, queriendo agregar líneas. Cada vuelta del ciclo borra lo anterior y solo sobrevive la última escritura. Si tu archivo termina con una sola línea cuando esperabas cien, ya sabes qué pasó: abriste con "w" donde debías usar "a", o abriste el archivo una vez por vuelta en vez de una vez fuera del ciclo.

Rutas y encoding: dos detalles que te van a salvar

Dos temas prácticos que separan al código que "funciona en mi máquina" del código que funciona en serio.

Rutas. Cuando escribes `open("notas.txt")`, Python busca el archivo en la carpeta de trabajo actual (donde estás parado en la terminal, o donde está tu script según cómo lo ejecutes). Si tu programa está en una carpeta y el archivo en otra, necesitas la ruta. Hay dos tipos: relativas (desde donde estás: `"datos/notas.txt"`, `"../notas.txt"`) y absolutas (la dirección completa: `"/home/usuario/documentos/notas.txt"` en Linux/Mac, `"C:\\usuarios\\documentos\\notas.txt"` en Windows).

El error más común aquí es el `FileNotFoundError` por estar parado en la carpeta equivocada. Tu código está bien, el archivo existe, pero Python lo busca en otro lado. La solución: verifica en qué carpeta estás (`pwd` en la terminal, o imprime la ruta desde Python) y usa rutas correctas. Un truco profesional es construir rutas relativas al archivo del programa, no a la carpeta de la terminal, pero eso ya es tema avanzado; por ahora, con saber que el problema existe y cómo diagnosticarlo tienes suficiente.

En Windows, cuidado con las diagonales invertidas: `"C:\\nuevo\\notas.txt"` tiene un problema, porque `\\n` es un salto de línea. Por eso en Windows se usan diagonales dobles (`"C:\\\\nuevo\\\\notas.txt"`) o strings crudos (`r"C:\\nuevo\\notas.txt"`), o simplemente diagonales normales (`"C:/nuevo/notas.txt"`), que Python en Windows también acepta. Detalle pequeño, dolor de cabeza grande si no lo sabes.

Encoding. Los archivos de texto guardan letras como números, y el encoding es el diccionario que traduce entre ambos. El estándar moderno es UTF-8, que incluye acentos, eñes y emojis. El problema: en algunas computadoras (sobre todo Windows), Python usa otro encoding por defecto, y entonces tus "áéíóú" y "ñ" se guardan o se leen como símbolos raros (Ã© en vez de é). La solución profesional es declarar siempre el encoding:

```
with open("notas.txt", "w", encoding="utf-8") as archivo:
    archivo.write("Mi mamá me mima.\n")

with open("notas.txt", "r", encoding="utf-8") as archivo:
    print(archivo.read())
```

Acostúmbrate a escribir `encoding="utf-8"` en cada `open()` que hagas con texto. Son 18 caracteres que te ahorran horas de depuración con textos en español. Cuando veas símbolos extraños en un archivo, el 99% de las veces el culpable es el encoding.

Persistencia: que tus datos sobrevivan al apagón

Ya sabes leer y escribir archivos. Ahora viene la idea que lo une todo: la persistencia. Persistencia significa que los datos de tu programa sobreviven cuando el programa termina. Sin persistencia, cada vez que ejecutas tu programa empieza de cero, como una memoria que se borra cada noche. Con persistencia, tu programa "recuerda" entre ejecuciones.

El patrón básico tiene dos mitades: guardar y cargar. Al terminar (o cada vez que algo cambia), guardas tus datos en un archivo. Al iniciar, intentas cargar ese archivo; si existe, recuperas los datos; si no existe (primera vez que se ejecuta), empiezas de cero.

Ejemplo: un contador de visitas persistente.

```
# Al iniciar: intentar cargar el contador
try:
    with open("visitas.txt", "r", encoding="utf-8") as archivo:
        visitas = int(archivo.read())
except FileNotFoundError:
    visitas = 0 # Primera vez: no hay archivo todavía

# Usar el dato
visitas += 1
print(f"Esta es la visita número {visitas}.")

# Al terminar: guardar el contador actualizado
with open("visitas.txt", "w", encoding="utf-8") as archivo:
    archivo.write(str(visitas))
```

La primera vez que ejecutas esto, no existe `visitas.txt`, así que el `except FileNotFoundError` pone el contador en 0. Luego suma 1 y guarda el 1 en el archivo. La segunda vez, el archivo existe, lee el 1, suma 1, muestra 2 y guarda el 2. Y así sucesivamente.

Tu programa ahora tiene memoria entre ejecuciones. Fíjate cómo el `try / except` y los archivos trabajan juntos: el manejo de errores cubre el caso de "todavía no hay datos guardados".

Para datos más complejos (listas, diccionarios), el formato de texto plano se queda corto. La solución estándar en Python es JSON: un formato de texto para representar datos estructurados que ya es un estándar mundial. Python trae el módulo `json` en su biblioteca estándar:

```
import json

# Guardar una lista de tareas
tareas = ["Comprar leche", "Estudiar Python", "Llamar a mamá"]
with open("tareas.json", "w", encoding="utf-8") as archivo:
    json.dump(tareas, archivo, ensure_ascii=False, indent=2)

# Cargar la lista de tareas
try:
    with open("tareas.json", "r", encoding="utf-8") as archivo:
        tareas = json.load(archivo)
except FileNotFoundError:
    tareas = []
```

`json.dump()` convierte tu lista (o diccionario) a texto JSON y la escribe en el archivo. `json.load()` hace lo contrario: lee el archivo y reconstruye la lista. El `ensure_ascii=False` permite que los acentos se guarden tal cual (sin él, "mamá" se guardaría como "mam\u00e1", que funciona pero es feo de leer). El `indent=2` formatea el archivo con indentación para que un humano también pueda leerlo.

Con esto ya puedes hacer programas de verdad: listas de tareas, agendas, inventarios, configuraciones, récords de juegos. El patrón es siempre el mismo: cargar al inicio (con `try / except` por si es la primera vez), trabajar en memoria durante la ejecución, guardar al final (o cada vez que algo cambie). Domina este patrón y habrás dado el salto de "programas que calculan" a "programas que sirven".

Módulos: no reinventes la rueda

Último gran tema de la unidad: los módulos. Un módulo en Python es simplemente un archivo .py con código que puedes reutilizar desde otros programas. Y la biblioteca estándar de Python (la "stdlib") trae cientos de ellos, gratis, probados por millones de personas. La palabra mágica para usarlos es `import`.

Hay tres formas de importar, y cada una tiene su momento:

```
# Forma 1: importar el módulo completo
import random
numero = random.randint(1, 10)
```

Con `import random`, traes todo el módulo y usas sus funciones con el prefijo `random.`. Es la forma más clara y la recomendada por defecto: siempre sabes de dónde viene cada función.

```
# Forma 2: importar solo lo que necesitas
from random import randint, choice
numero = randint(1, 10)
fruta = choice(["manzana", "pera", "uva"])
```

Con `from random import randint`, traes solo esas funciones y las usas sin prefijo. Es más corto, pero pierde claridad: si alguien lee `randint(1, 10)` sin contexto, no sabe de qué módulo viene. Úsalo cuando el módulo es muy conocido o cuando vas a usar la función muchísimas veces.

```
# Forma 3: importar con alias (apodo)
import random as rnd
numero = rnd.randint(1, 10)
```

El `as` le pone un apodo al módulo. Se usa mucho con módulos de nombre largo o por convención en ciertas comunidades (como `import numpy as np` en ciencia de datos, que verás si algún día entras a ese mundo).

Hay una forma 4 que debes evitar:

```
# Forma 4: from x import * (NO LA USES)
from random import *
```

Esto importa TODO sin prefijo. Parece cómodo, pero es una pesadilla: no sabes qué nombres trajiste, puedes sobrescribir tus propias variables sin darte cuenta (¿y si el módulo tiene una función llamada `choice` y tú también defines una?), y quien lee tu código no sabe de dónde sale cada cosa. Los profesionales la evitan.

La biblioteca estándar: tu caja de herramientas gratis

La biblioteca estándar de Python es enorme: más de 200 módulos. No necesitas aprenderlos todos; con conocer los más útiles tienes para rato. Vamos con los esenciales.

random: aleatoriedad para juegos y sorteos. Lo viste en la clase 26. Sus funciones más usadas:

```
import random

dado = random.randint(1, 6)           # Entero aleatorio entre 1 y 6 (ambos
incluidos)
numero = random.randrange(0, 100, 5) # Como range(): 0, 5, 10... hasta 95
ganador = random.choice(["Ana", "Luis", "María"]) # Elige un elemento al azar
cartas = ["A", "K", "Q", "J"]
random.shuffle(cartas)                 # Mezcla la lista EN SU LUGAR (no devuelve
nada)
probabilidad = random.random()         # Decimal aleatorio entre 0.0 y 1.0
```

Ojo con `shuffle()`: modifica la lista original y devuelve `None`. Si escribes `mezcladas = random.shuffle(cartas)`, `mezcladas` será `None` y te quedarás sin cartas. Error clásico de principiante. Úsalo así: `random.shuffle(cartas)` y luego usa `cartas`, que ya quedó mezclada.

Un detalle curioso: los números de `random` no son verdaderamente aleatorios, son pseudoaleatorios (se generan con una fórmula a partir de una "semilla"). Para juegos y sorteos casuales son perfectos. Para criptografía o seguridad real no sirven, pero eso ya es otro nivel.

math: matemáticas en serio. Funciones matemáticas que no vienen integradas:

```
import math

print(math.sqrt(16))      # 4.0 - raíz cuadrada
print(math.pi)           # 3.141592653589793 - el número pi
print(math.ceil(4.2))    # 5 - redondea hacia arriba
print(math.floor(4.9))   # 4 - redondea hacia abajo
print(math.pow(2, 10))    # 1024.0 - potencia (2**10 hace lo mismo)
print(math.factorial(5)) # 120 - factorial
```

math también trae trigonometría (sin, cos, tan), logaritmos (log, log10) y constantes como math.e y math.inf (infinito). Si alguna vez necesitas matemáticas más allá de +, -, *, /, aquí están.

datetime: fechas y horas. Esencial para bitácoras, agendas y cualquier programa que trabaje con tiempo:

```
import datetime

hoy = datetime.date.today()
print(hoy)                # 2026-09-19 (formato año-mes-día)
print(hoy.year, hoy.month, hoy.day)

ahora = datetime.datetime.now()
print(ahora.strftime("%d/%m/%Y %H:%M")) # 19/09/2026 09:49 (formato mexicano)
```

El método `strftime()` formatea fechas con códigos: %d día, %m mes, %Y año con 4 dígitos, %H hora, %M minutos. El formato %d/%m/%Y es el que usamos en México (día/mes/año), a diferencia del formato gringo. También puedes hacer operaciones con fechas: `hoy + datetime.timedelta(days=7)` te da la fecha de dentro de una semana. Muy útil para calcular vencimientos y recordatorios.

os: hablar con el sistema operativo (mención breve). El módulo `os` te deja hacer desde Python cosas como listar archivos de una carpeta o crear carpetas:

```
import os

archivos = os.listdir(".") # Lista lo que hay en la carpeta actual
print(archivos)
```

Es la puerta para automatizar tareas del sistema: renombrar cien archivos de una vez, organizar descargas, revisar qué hay en una carpeta. Úsalo con cuidado: con gran poder viene gran responsabilidad (borrar archivos desde código no tiene papelera de reciclaje).

pip: la tienda de módulos de la comunidad. Cuando la biblioteca estándar no basta, existe PyPI (el Índice de Paquetes de Python): cientos de miles de paquetes gratuitos creados por la comunidad. Se instalan con `pip`, que ya viene con Python:

```
pip install requests
```

Y luego en tu código: `import requests`. Así de simple. Hay paquetes para todo: desarrollo web, análisis de datos, inteligencia artificial, videojuegos, automatización. Solo dos consejos: instala solo paquetes conocidos y populares (revisa que tengan buena reputación), y ten cuidado con los nombres (existen paquetes maliciosos con nombres parecidos a los famosos). Con eso, tienes a tu disposición el trabajo de millones de programadores.

Crea tu propio módulo: comparte tu código contigo mismo

Aquí viene algo hermoso: tú también puedes crear módulos. Un módulo no es más que un archivo `.py` con funciones, y cualquier programa en la misma carpeta puede importarlo. Supón que escribes `mis_utilidades.py`:

```
# Archivo: mis_utilidades.py
def saludar(nombre):
    return f"Hola, {nombre}. ¡Qué bueno verte!"

def es_par(n):
    return n % 2 == 0

PI_MEXICANO = 3.1416
```

Y en otro archivo, en la misma carpeta:

```
# Archivo: programa.py
import mis_utilidades

print(mis_utilidades.saludar("Ana"))
print(mis_utilidades.es_par(10))
print(mis_utilidades.PI_MEXICANO)
```

¡Listo! Tu `programa.py` usa código de `mis_utilidades.py` como si fuera parte de la biblioteca estándar. Así es como crecen los programas reales: empiezas con todo en un archivo, luego separas las funciones útiles en módulos y los reutilizas en varios programas. Tu colección de módulos personales se convierte en tu caja de herramientas privada.

Un detalle técnico: cuando importas un módulo, Python ejecuta todo el código que está en su nivel superior (fuera de funciones). Si tu módulo tiene código de prueba al final, se ejecutaría al importarlo, lo cual no quieres. La convención profesional para evitarlo es:

```
# Al final de mis_utilidades.py
if __name__ == "__main__":
    # Código de prueba: solo se ejecuta si corres este archivo directamente
    print(saludar("Mundo"))
    print(es_par(7))
```

Esa línea mágica significa: "si este archivo se está ejecutando como programa principal, corre las pruebas; si solo se está importando como módulo, no hagas nada". La verás en casi todo código Python profesional. Por ahora, tómalo como receta: pon tus pruebas ahí dentro y tu módulo será importable sin efectos secundarios.

Mención rápida de algo que te puede morder algún día: los imports circulares. Si el módulo A importa al módulo B, y el módulo B importa al módulo A, Python se confunde y lanza un error. La solución es diseñar tus módulos sin dependencias circulares: si dos módulos se necesitan mutuamente, probablemente deberían ser uno solo, o necesitar un tercer módulo con lo compartido. No te preocupes mucho por ahora; solo recuerda el nombre del problema para cuando te aparezca.

Errores comunes de la unidad (el mapa de minas)

Antes de los ejemplos, un repaso de las trampas típicas de estos tres temas. Considéralo tu mapa de minas personal.

****El `except` sin tipo que oculta bugs.**** Ya lo vimos: `except:` desnudo o `except Exception:` atrapan demasiado y esconden errores de tu propio código. Cuando algo "no funciona pero tampoco marca error", sospecha de un `except` genérico que está tragándose la evidencia. Atrapa siempre el tipo específico que esperas.

****Olvidar cerrar el archivo (no usar `with`).**** Con `open()` suelto y sin `close()`, los datos pueden no guardarse y el archivo quedar bloqueado. La solución ya la sabes: usa `with` siempre. Si ves código con `open()` sin `with` en tutoriales, es código viejo o descuidado.

****El modo "w" accidental.**** Abrir con "w" un archivo que querías conservar borra todo su contenido al instante. Antes de usar "w", pregúntate: "¿quiero empezar de cero o agregar?". Si es agregar, usa "a". Este error duele porque no hay "deshacer".

FileNotFoundError por ruta mal. El archivo existe, tu código está bien, pero Python lo busca en otra carpeta. Diagnóstico: imprime en qué carpeta estás parado y verifica la ruta. Recuerda que las rutas relativas se resuelven desde la carpeta de trabajo actual, no desde donde está tu script.

****Olvidar el `encoding="utf-8"`.** Tus acentos se ven como jeroglíficos. En lectura y en escritura, con archivos de texto en español, declara siempre el encoding. Es un hábito barato que evita un dolor caro.

****Olvidar el `\n` al escribir líneas.**** `write()` no agrega saltos de línea solo. Si tu archivo sale como un solo churro de texto larguísimo, te faltaron los `\n`. Del otro lado, al leer con `readlines()` o iterando, cada línea trae su `\n` incluido: usa `strip()` si no lo quieres.

****Confundir `read()` con `readlines()`.** `read()` devuelve un string gigante; `readlines()` devuelve una lista de strings. Si haces `for linea in archivo.read():`, estarás iterando letra por letra (porque iterar un string itera caracteres), no línea por línea. Decide qué quieres y usa la función correcta.

****El `shuffle()` que devuelve `None`.** `random.shuffle(lista)` mezcla la lista en su lugar y no devuelve nada. No lo asignes a una variable. Error clásico que deja tu variable en `None` y tu lista mezclada donde no la esperabas.

****El `from x import *`.** Importa todo sin prefijo, ensucia tu espacio de nombres y puede sobrescribir tus variables en silencio. Evítalo siempre; usa `import x` o `from x import nombre_especifico`.

Buenas prácticas de la unidad

Cerramos la teoría con las reglas que distinguen al profesional. No son opcionales si quieres que tu código dure.

Atrapa excepciones específicas. Ya lo sabes: `except ValueError`, no `except:`. Cada `except` debe responder a la pregunta "¿qué error espero aquí y por qué?".

Dale mensajes amables al usuario. Cuando atrapes un error, no muestres el traceback al usuario final: muéstrale un mensaje en su idioma que le diga qué pasó y qué puede hacer. "Eso no es un número válido, intenta de nuevo" es mil veces mejor que `ValueError`:

`invalid literal for int()`. Los `tracebacks` son para ti, el programador; los mensajes amables son para el usuario.

****Usa `with` siempre para archivos.**** Sin excepciones (valga la redundancia). Es más corto, más claro y más seguro que `open()` / `close()` manual.

Valida todo lo que venga del usuario. Nunca uses datos del usuario sin validar: ni números sin `try / except`, ni rangos sin `if`, ni texto sin revisar que no esté vacío. El usuario no es tu enemigo, pero sí es impredecible. El patrón `while + try` es tu mejor amigo.

No dejes que un error rompa lo que no debe. Pregúntate: "si esto falla, ¿qué debería pasar?". A veces la respuesta es reintentar, a veces usar un valor por defecto, a veces avisar y continuar, a veces sí detenerse con un mensaje claro. Lo importante es que sea una decisión consciente, no un accidente.

Separa lo que puede fallar de lo que no. Mete en el `try` solo el código peligroso. Mientras más pequeño el `try`, más claro es qué pudo fallar y más fácil es depurar. Si metes cincuenta líneas en un `try`, cuando algo falle no sabrás cuál fue.

Con la teoría dominada, vamos a ver todo esto en acción con ejemplos completos que puedes correr en tu computadora.

Ejemplos paso a paso

Ejemplo 1: Calculadora a prueba de errores

Una calculadora que no se rompa pase lo que pase: si el usuario escribe letras, si divide entre cero o si elige una operación inválida, el programa responde con amabilidad y sigue funcionando.

```

def pedir_numero(mensaje):
    while True:
        try:
            return float(input(mensaje))
        except ValueError:
            print("Eso no es un número válido, intenta de nuevo.")

def calculadora():
    print("=== Calculadora a prueba de todo ===")
    a = pedir_numero("Primer número: ")
    b = pedir_numero("Segundo número: ")
    op = input("Operación (+, -, *, /): ").strip()

    try:
        if op == "+":
            resultado = a + b
        elif op == "-":
            resultado = a - b
        elif op == "*":
            resultado = a * b
        elif op == "/":
            resultado = a / b
        else:
            print("Operación no reconocida.")
            return
    except ZeroDivisionError:
        print("No se puede dividir entre cero.")
        return

    print(f"Resultado: {a} {op} {b} = {resultado}")

calculadora()

```

Línea por línea: `pedir_numero()` encapsula el patrón `while + try` que aprendiste: pregunta hasta recibir un número válido y lo devuelve como `float` (para aceptar decimales). En `calculadora()`, pedimos los dos números con esa función, así que `a` y `b` siempre son números válidos. La operación se lee como texto y se limpia con `strip()`. El bloque `try` contiene solo la parte peligrosa: la división. Si `op` es `"/"` y `b` es 0, Python lanza `ZeroDivisionError` y el `except` responde con elegancia en vez de romperse. Si la operación no es válida, avisamos y salimos con `return`. Al final, el f-string muestra la operación completa. Prueba escribir "tacos" como número y "0" como divisor: el programa ni se inmuta.

Ejemplo 2: Diario personal que guarda en archivo

Un diario donde cada entrada se agrega al final del archivo con fecha y hora, sin borrar las anteriores.

```
import datetime

def escribir_en_diario():
    entrada = input("¿Qué quieres anotar hoy? ").strip()
    if not entrada:
        print("No escribiste nada, no se guardó.")
        return

    ahora = datetime.datetime.now()
    fecha = ahora.strftime("%d/%m/%Y %H:%M")

    with open("diario.txt", "a", encoding="utf-8") as archivo:
        archivo.write(f"[{fecha}] {entrada}\n")

    print("Entrada guardada en diario.txt")

escribir_en_diario()
```

Línea por línea: importamos `datetime` para la fecha. Leemos la entrada y con `if not entrada` detectamos texto vacío (un string vacío es "falso" en Python): si el usuario no escribió nada, avisamos y salimos sin guardar basura. `strftime("%d/%m/%Y %H:%M")` formatea la fecha al estilo mexicano. Abrimos con modo "a" (append): cada ejecución agrega al final sin borrar lo anterior, y `encoding="utf-8"` cuida los acentos. El `with` cierra solo. Ejecútalo tres días seguidos y abre `diario.txt`: verás tu historial creciendo. Cambia "a" por "w" y verás el peligro en acción (hazlo con un archivo de prueba, no con tu diario real).

Ejemplo 3: Lista de tareas persistente

El ejemplo estrella de la persistencia: agregas tareas, las ves, y todo se guarda en un archivo JSON que sobrevive entre ejecuciones.

```
import json
import os

ARCHIVO = "tareas.json"

def cargar_tareas():
```

```

if not os.path.exists(ARCHIVO):
    return []
try:
    with open(ARCHIVO, "r", encoding="utf-8") as f:
        return json.load(f)
except json.JSONDecodeError:
    print("El archivo de tareas está dañado, empezamos de cero.")
    return []

def guardar_tareas(tareas):
    with open(ARCHIVO, "w", encoding="utf-8") as f:
        json.dump(tareas, f, ensure_ascii=False, indent=2)

def main():
    tareas = cargar_tareas()
    while True:
        print("\n1) Ver tareas 2) Agregar 3) Eliminar 4) Salir")
        opcion = input("Elige: ").strip()
        if opcion == "1":
            for i, t in enumerate(tareas, 1):
                print(f"{i}. {t}")
            if not tareas:
                print("(sin tareas)")
        elif opcion == "2":
            nueva = input("Nueva tarea: ").strip()
            if nueva:
                tareas.append(nueva)
                guardar_tareas(tareas)
                print("Guardada.")
        elif opcion == "3":
            try:
                n = int(input("Número de tarea a eliminar: "))
                eliminada = tareas.pop(n - 1)
                guardar_tareas(tareas)
                print(f"Eliminada: {eliminada}")
            except (ValueError, IndexError):
                print("Número inválido.")
        elif opcion == "4":
            print("¡Hasta luego!")
            break
        else:
            print("Opción no válida.")

main()

```

Línea por línea: `cargar_tareas()` primero verifica si el archivo existe con `os.path.exists()`; si no, devuelve lista vacía (primera ejecución). Si existe, lo lee con `json.load()`; y si el archivo está corrupto (alguien lo editó mal), `json.JSONDecodeError` lo detecta y empezamos de cero en vez de romperse. `guardar_tareas()` escribe la lista con `json.dump()`, con acentos legibles e indentación bonita. En `main()`, el menú en ciclo permite ver (con `enumerate` para numerar), agregar (validando que no esté vacía y guardando de inmediato), eliminar (con `try / except` que atrapa tanto texto no numérico como número fuera de rango en una sola tupla de excepciones) y salir. Cada cambio se guarda al momento: cierra el programa, vuelve a abrirlo y tus tareas siguen ahí. Eso es persistencia real.

Ejemplo 4: Generador de contraseñas con `random` y `string`

Un generador de contraseñas seguras que combina letras, dígitos y símbolos al azar.

```
import random
import string

def generar_contrasena(longitud=12):
    if longitud < 4:
        raise ValueError("La longitud mínima es 4.")
    caracteres = string.ascii_letters + string.digits + string.punctuation
    return "".join(random.choice(caracteres) for _ in range(longitud))

try:
    n = int(input("¿De cuántos caracteres? (mínimo 4): "))
    print(f"Tu contraseña: {generar_contrasena(n)}")
except ValueError as e:
    print(f"No se pudo generar: {e}")
```

Línea por línea: importamos `random` y `string` (un módulo de la biblioteca estándar con constantes de texto útiles: `ascii_letters` son todas las letras, `digits` los dígitos, `punctuation` los símbolos). La función valida con `raise ValueError` si la longitud es menor a 4: aquí ves `raise` en acción real, defendiendo las reglas de la función. `random.choice()` elige un carácter al azar por cada posición, y `"".join(...)` une todo en un solo string (la expresión dentro es un generador, primo de las comprensiones que verás en la Unidad 10). Al pedir la longitud, un solo `except ValueError as e` atrapa dos problemas distintos: texto no numérico del `int()` y el `raise` de la función; el mensaje `e` distingue cuál fue. Ejecútalo varias veces: cada contraseña es distinta.

Ejemplo 5: Juego de dados con random

Un juego completo: tú y la computadora lanzan dos dados, gana quien saque más. Incluye validación y repetición.

```

import random

def lanzar_dados():
    return random.randint(1, 6) + random.randint(1, 6)

def jugar():
    print("=== Duelo de dados ===")
    puntos_jugador = 0
    puntos_cpu = 0

    rondas = 0
    while rondas <= 0:
        try:
            rondas = int(input("¿Cuántas rondas? "))
            if rondas <= 0:
                print("Debe ser al menos 1.")
        except ValueError:
            print("Escribe un número entero.")

    for ronda in range(1, rondas + 1):
        input(f"\nRonda {ronda}: presiona Enter para lanzar...")
        tj = lanzar_dados()
        tc = lanzar_dados()
        print(f"Tú: {tj} | CPU: {tc}")
        if tj > tc:
            print("¡Ganas la ronda!")
            puntos_jugador += 1
        elif tc > tj:
            print("La CPU gana la ronda.")
            puntos_cpu += 1
        else:
            print("Empate.")

    print(f"\nMarcador final: Tú {puntos_jugador} - {puntos_cpu} CPU")
    if puntos_jugador > puntos_cpu:
        print("¡CAMPEÓN!")
    elif puntos_cpu > puntos_jugador:
        print("La CPU te venció. La revancha te espera.")
    else:
        print("Empate total. ¡Qué duelo!")

jugar()

```

Línea por línea: `lanzar_dados()` suma dos dados de 6 caras con `randint(1, 6)`. El número de rondas se valida con el patrón `while + try`, más una verificación de rango (al menos 1). El ciclo `for` con `range(1, rondas + 1)` numera las rondas desde 1. `input()` sin guardar el resultado solo pausa hasta que el usuario presiona Enter, creando suspenso. Se comparan los tiros, se acumulan puntos y al final se declara al ganador con mensajes con sabor. Fíjate cómo `random` convierte unas líneas en un juego real: sin el módulo, tendrías que inventar la aleatoriedad tú mismo.

Ejemplo 6: Lector de archivo con conteo de líneas, palabras y caracteres

Una mini-herramienta de análisis de texto: abre un archivo, lo lee y reporta estadísticas. Maneja el caso de archivo inexistente.

```
def analizar_archivo(nombre):
    try:
        with open(nombre, "r", encoding="utf-8") as archivo:
            contenido = archivo.read()
    except FileNotFoundError:
        print(f"No encontré el archivo '{nombre}'. Revisa el nombre y la carpeta.")
        return
    except UnicodeDecodeError:
        print(f"No pude leer '{nombre}': el encoding no es compatible.")
        return

    lineas = contenido.splitlines()
    palabras = contenido.split()
    caracteres = len(contenido)

    print(f"Archivo: {nombre}")
    print(f"Líneas: {len(lineas)}")
    print(f"Palabras: {len(palabras)}")
    print(f"Caracteres: {caracteres}")
    if palabras:
        mas_larga = max(palabras, key=len)
        print(f"Palabra más larga: '{mas_larga}' ({len(mas_larga)} letras)")

nombre = input("Nombre del archivo a analizar: ").strip()
analizar_archivo(nombre)
```

Línea por línea: el `try` abre y lee con `with` y `encoding="utf-8"`. Dos `except` cubren dos fallos reales: archivo inexistente y archivo con encoding incompatible (por ejemplo, un archivo

binario). En ambos casos, mensaje amable y `return` temprano: la función no sigue con datos que no tiene. `splitlines()` divide en líneas (sin los `\n`), `split()` sin argumentos divide en palabras separando por cualquier espacio en blanco, y `len()` del contenido cuenta caracteres. `max(palabras, key=len)` encuentra la palabra más larga usando la longitud como criterio. Pruébalo con tu `diario.txt` del ejemplo 2 y luego con un nombre inventado: verás el programa comportarse como una herramienta profesional en ambos casos.

Práctica

Ejercicio 1

Escribe un programa que pida dos números y muestre el resultado de dividir el primero entre el segundo. Debe manejar: texto en vez de números, división entre cero, y seguir pidiendo hasta que la operación sea válida.

Solución paso a paso: necesitamos el patrón `while + try` para los números y un `except` extra para la división. El truco es meter la división dentro del mismo `try` que la conversión, o usar dos bloques.

```
while True:
    try:
        a = float(input("Dividendo: "))
        b = float(input("Divisor: "))
        resultado = a / b
        break
    except ValueError:
        print("Ambos deben ser números, intenta de nuevo.")
    except ZeroDivisionError:
        print("El divisor no puede ser cero, intenta de nuevo.")

print(f"{a} / {b} = {resultado}")
```

Si el usuario escribe letras, `float()` lanza `ValueError`; si el divisor es 0, la división lanza `ZeroDivisionError`. En ambos casos el ciclo repite sin romperse. Solo el `break` (cuando todo salió bien) nos saca. Concepto clave: varios `except` permiten respuestas distintas para problemas distintos dentro del mismo intento.

Ejercicio 2

Crea un programa que registre calificaciones en un archivo: pide calificaciones una por una (termina con la palabra "fin"), las guarda en `calificaciones.txt` (una por línea, agregando sin borrar) y al final muestra el promedio.

Solución paso a paso: ciclo de captura con validación, apertura en modo "a" para no borrar registros anteriores, y relectura del archivo para calcular el promedio.

```
while True:
    entrada = input("Calificación (o 'fin'): ").strip()
    if entrada.lower() == "fin":
        break
    try:
        calif = float(entrada)
        if 0 <= calif <= 10:
            with open("calificaciones.txt", "a", encoding="utf-8") as f:
                f.write(f"{calif}\n")
        else:
            print("Debe estar entre 0 y 10.")
    except ValueError:
        print("Eso no es un número válido.")

with open("calificaciones.txt", "r", encoding="utf-8") as f:
    califs = [float(linea.strip()) for linea in f if linea.strip()]

if califs:
    print(f"Registradas: {len(califs)}, promedio: {sum(califs)/len(califs):.2f}")
else:
    print("No hay calificaciones registradas.")
```

El modo "a" conserva registros de ejecuciones pasadas. Al leer, filtramos líneas vacías y convertimos a `float`. Concepto clave: combinar captura validada + persistencia en archivo + relectura para estadísticas es el esqueleto de docenas de programas reales.

Ejercicio 3

Simula el lanzamiento de una moneda 1000 veces con `random` y muestra cuántas veces cayó águila y cuántas sol, con porcentajes. Luego repite el experimento con 10 lanzamientos y observa la diferencia.

Solución paso a paso: `random.choice()` para cada lanzamiento, conteo con variables, y comparación de muestras pequeña vs. grande.

```
import random

def experimento(n):
    aguilas = sum(1 for _ in range(n) if random.choice(["águila", "sol"]) == "águila")
    soles = n - aguilas
    print(f"Lanzamientos: {n}")
    print(f"  Águilas: {aguilas} ({aguilas/n:.1%})")
    print(f"  Soles:   {soles} ({soles/n:.1%})")

experimento(10)
print()
experimento(1000)
```

Con 10 lanzamientos verás proporciones disparatadas (70%-30% no es raro); con 1000, los porcentajes se pegan al 50%. El formato: `.1%` convierte a porcentaje con un decimal. Concepto clave: la aleatoriedad se "empareja" solo con muestras grandes, una lección de probabilidad que `random` te deja comprobar en segundos.

Ejercicio 4

Escribe tu propio módulo `geometria.py` con funciones para el área del círculo, el rectángulo y el triángulo (usa `math.pi`). Luego escribe `programa.py` que lo importe y calcule las tres áreas con datos del usuario validados.

Solución paso a paso: primero el módulo con funciones puras y validación con `raise`; luego el programa que lo importa.

```

# geometria.py
import math

def area_circulo(radio):
    if radio <= 0:
        raise ValueError("El radio debe ser positivo.")
    return math.pi * radio ** 2

def area_rectangulo(base, altura):
    if base <= 0 or altura <= 0:
        raise ValueError("Base y altura deben ser positivas.")
    return base * altura

def area_triangulo(base, altura):
    return area_rectangulo(base, altura) / 2

if __name__ == "__main__":
    print(area_circulo(1), area_rectangulo(2, 3), area_triangulo(2, 3))

```

```

# programa.py
import geometria

def pedir_positivo(mensaje):
    while True:
        try:
            valor = float(input(mensaje))
            if valor > 0:
                return valor
            print("Debe ser un número positivo.")
        except ValueError:
            print("Escribe un número válido.")

r = pedir_positivo("Radio del círculo: ")
b = pedir_positivo("Base del rectángulo/triángulo: ")
h = pedir_positivo("Altura: ")
print(f"Círculo: {geometria.area_circulo(r):.2f}")
print(f"Rectángulo: {geometria.area_rectangulo(b, h):.2f}")
print(f"Triángulo: {geometria.area_triangulo(b, h):.2f}")

```

El módulo reutiliza `area_rectangulo` dentro de `area_triangulo` (no repitas código) y el bloque `if __name__ == "__main__"` permite probarlo sin afectar la importación. Concepto clave: separar la lógica (módulo) de la interacción con el usuario (programa) es la base de todo proyecto organizado.

Ejercicio 5

Haz un programa que lea un archivo de texto y cree una copia llamada `copia_segura.txt` donde cada línea vaya numerada (ej. 1: primera línea). Si el archivo original no existe, debe avisarlo amablemente sin romperse.

Solución paso a paso: leer con manejo de `FileNotFoundError`, enumerar líneas y escribir la copia.

```
origen = input("Archivo a copiar: ").strip()

try:
    with open(origen, "r", encoding="utf-8") as f:
        lineas = f.readlines()
except FileNotFoundError:
    print(f"No existe '{origen}'. Nada que copiar.")
else:
    with open("copia_segura.txt", "w", encoding="utf-8") as f:
        for i, linea in enumerate(lineas, 1):
            f.write(f"{i}: {linea}")
        print(f"Copiado: {len(lineas)} líneas numeradas en copia_segura.txt")
```

El `else` del `try` garantiza que solo creamos la copia si la lectura funcionó. `enumerate(lineas, 1)` numera desde 1. Nota que `linea` ya trae su `\n`, así que no agregamos otro. Concepto clave: leer de un archivo y escribir en otro es la operación fundamental del procesamiento de archivos (conversiones, filtros, reportes).

Ejercicio 6

Crea un "adivinator de número" persistente: la computadora elige un número del 1 al 100, el usuario lo adivina con pistas de "más alto/más bajo", y el mejor récord (menos intentos) se guarda en `record.txt` entre partidas.

Solución paso a paso: juego con `random.randint`, validación de entrada, y récord persistente con el patrón cargar/guardar.

```

import random

def leer_record():
    try:
        with open("record.txt", "r", encoding="utf-8") as f:
            return int(f.read().strip())
    except (FileNotFoundError, ValueError):
        return None

def guardar_record(intentos):
    with open("record.txt", "w", encoding="utf-8") as f:
        f.write(str(intentos))

record = leer_record()
if record:
    print(f"Récord actual: {record} intentos.")

secreto = random.randint(1, 100)
intentos = 0
while True:
    try:
        n = int(input("Adivina (1-100): "))
    except ValueError:
        print("Escribe un número entero.")
        continue
    intentos += 1
    if n < secreto:
        print("Más alto...")
    elif n > secreto:
        print("Más bajo...")
    else:
        print(f"¡Correcto en {intentos} intentos!")
        if record is None or intentos < record:
            print("¡Nuevo récord!")
            guardar_record(intentos)
        break

```

`leer_record()` atrapa dos errores en una tupla: archivo inexistente (primera partida) y contenido corrupto. El juego valida cada intento y solo cuenta los válidos. Al ganar, compara con el récord y lo actualiza si lo supera. Concepto clave: este ejercicio junta TODO lo de la unidad (random, try/except, validación, archivos, persistencia) en un programa que ya se siente como un juego de verdad.

Resumen de la unidad

En esta unidad tus programas se volvieron profesionales: ya no se rompen ante lo inesperado y ya no olvidan todo al apagarse. En manejo de errores aprendiste la filosofía correcta (los errores son mensajes, no fracasos), a leer un traceback empezando por la última línea (tipo de error) y subiendo a la ruta (archivo y línea), y a reconocer por su nombre a los ocho vecinos problemáticos: `SyntaxError`, `NameError`, `TypeError`, `ValueError`, `IndexError`, `KeyError`, `ZeroDivisionError` y `FileNotFoundError`. Dominas `try / except` con excepciones específicas (nunca el `except` desnudo que oculta bugs), múltiples bloques `except`, `else` para el camino feliz y `finally` para la limpieza garantizada; sabes lanzar tus propios errores con `raise` y aplicas el patrón `while + try` para validar la entrada del usuario hasta que sea correcta.

En archivos manejas `open()` con sus tres modos (`"r"` leer, `"w"` escribir borrando, `"a"` agregar), usas `with` siempre para que el cierre sea automático, lees con `read()` / `readline()` / `readlines()` o iterando línea por línea, escribes con `write()` acordándote del `\n`, respetas el peligro del modo `"w"`, declaras `encoding="utf-8"` para tus acentos y aplicas el patrón de persistencia: cargar al inicio (con `try / except` por si es la primera vez), trabajar en memoria y guardar al final, usando `json` para listas y diccionarios.

En módulos usas `import`, `from x import y` e `import x as alias` (evitando el `from x import *`), aprovechas la biblioteca estándar (`random` con `randint/choice/shuffle`, `math` con `sqrt/pi/ceil/floor`, `datetime` para fechas, `os` para el sistema) e instalas paquetes de la comunidad con `pip`. Y lo más bonito: ya creas tus propios módulos, archivos `.py` reutilizables con el bloque `if __name__ == "__main__"` para pruebas.

La idea que une todo: un programa robusto anticipa el fracaso y tiene un plan; un programa persistente guarda lo importante; y un programador inteligente no reinventa la rueda. Con estas tres habilidades, tus programas dejan de ser ejercicios y empiezan a ser herramientas. En la Unidad 10 cerrarás el curso con las técnicas finales que pulen todo lo aprendido. ¡A programar!

UNIDAD 10: Objetos, tu primer juego y tu graduación como programador

Introducción

Llegaste. Después de nueve unidades, de decenas de ejemplos, de errores que te sacaron canas y de esos momentos en los que tu código por fin corrió y sentiste que podías con todo, estás aquí: en la última unidad del curso. Y no llegaste por suerte. Llegaste porque escribiste código, porque rompiste cosas, porque las arreglaste, porque practicaste. Así que antes de cualquier otra cosa, date el crédito: hace unas semanas Python era solo una palabra rara para ti, y hoy escribes programas que piden datos, toman decisiones, repiten tareas, guardan información en archivos y se defienden de los errores. Eso no es poca cosa. Eso es programar de verdad.

Ahora imagina esto: durante todo el curso has trabajado como un artesano muy talentoso que moldea cada pieza a mano, una por una. Necesitas un perro en tu programa y escribes las variables del perro. Necesitas otro perro y vuelves a escribir todo. Necesitas cien perros y... bueno, ya entendiste el problema. Funciona, pero es lento, repetitivo y se vuelve un desmadre en cuanto el programa crece.

En esta unidad vas a aprender a fabricar moldes.

Piensa en una fábrica de galletas. El pastelero no moldea cada galleta a mano: diseña un molde con la forma que quiere y luego lo presiona contra la masa una y otra vez. Cada galleta que sale es independiente (una se puede quemar sin que las demás se arruinen), pero todas comparten la misma forma, porque nacieron del mismo molde. Si el pastelero quiere cambiar el diseño, no retoca galleta por galleta: modifica el molde una sola vez y todas las galletas futuras salen con el nuevo diseño.

Eso, exactamente eso, es la Programación Orientada a Objetos. La clase es el molde. Los objetos son las galletas. Y tú vas a ser el pastelero.

Hay otra forma de verlo que también ayuda: piensa en los planos de una casa. El arquitecto dibuja el plano una vez: cuántos cuartos, dónde va la cocina, dónde las ventanas. Con ese mismo plano se pueden construir cien casas en cien terrenos distintos. Cada casa es independiente (en una viven los Pérez, en otra vive doña Marta, una la pintan de azul y otra de amarillo), pero todas nacieron del mismo plano. En programación, el plano es la clase y cada

casa construida es un objeto, también llamado instancia. Cuando alguien dice "crea una instancia de la clase", está diciendo "construye una casa usando ese plano".

¿Por qué esto importa tanto? Porque el mundo real está lleno de objetos, no de instrucciones sueltas. Un banco no piensa en "instrucciones para manejar dinero": piensa en cuentas, cada una con su dueño y su saldo, cada una capaz de recibir depósitos y dar retiros. Una escuela no piensa en instrucciones: piensa en estudiantes, cada uno con su nombre y sus calificaciones, cada uno capaz de calcular su promedio. La Programación Orientada a Objetos te deja modelar el mundo así, tal como es: como un conjunto de cosas (objetos) que tienen características (atributos) y pueden hacer acciones (métodos). Por eso los programas grandes, los de verdad, los que usan las empresas, están contruidos con objetos. Hoy cruzas esa puerta.

Y como esta es tu graduación, no nos vamos a quedar en la teoría. Vas a construir tu primer proyecto completo: el juego de "adivina el número". La computadora piensa un número secreto, tú intentas adivinarlo y ella te da pistas de "más alto" o "más bajo" hasta que lo encuentras. Parece sencillo, y lo es, pero por dentro usa casi todo lo que aprendiste: variables, entrada de datos, conversiones, condicionales, el ciclo `while`, el módulo `random` y manejo de errores. Es la prueba perfecta de que todo lo que estudiaste no eran temas sueltos: eran piezas de un rompecabezas que ahora armas completo.

Después del proyecto viene la graduación de verdad: un repaso de todo el curso, unidad por unidad, para que veas el mapa completo de lo que ya dominas. Y luego, lo más emocionante: tu siguiente paso. Te voy a mostrar las rutas que se abren frente a ti (proyectos reales, inteligencia artificial, desarrollo web, automatización), con ideas concretas para que no te quedes con la pregunta de "¿y ahora qué hago?".

Un consejo de graduado a futuro graduado: esta unidad se disfruta más si la vives como lo que es, un cierre. No la leas a las carreras para "terminar el curso". Léela despacio, escribe cada ejemplo con tus propias manos, juega con el juego hasta aburrirte, modifica el código y observa qué pasa. La diferencia entre alguien que "tomó un curso de Python" y alguien que "programa en Python" no está en el certificado: está en las horas que pasó rompiendo y arreglando código por su cuenta. Tú ya eres de los segundos. Solo falta que lo celebres.

Vamos a fabricar moldes. Vamos a construir tu juego. Vamos a graduarnos.

Explicación detallada

Qué es un objeto (y por qué ya los conoces)

Antes de escribir una sola línea de código, hagamos un ejercicio. Mira a tu alrededor y elige tres cosas: tu teléfono, una silla y tu taza de café. Cada una de esas cosas tiene dos tipos de información. Primero, características: tu teléfono tiene marca, modelo, color y nivel de batería; la silla tiene material, color y número de patas; la taza tiene tamaño, color y si está llena o vacía. Segundo, cosas que puede hacer: el teléfono puede llamar, tomar fotos y reproducir música; la silla... bueno, la silla no hace mucho, pero soporta peso; la taza puede contener líquido.

En programación orientada a objetos, cada una de esas cosas sería un objeto. Un objeto es una unidad que junta datos (las características) con acciones (lo que puede hacer). A las características les llamamos atributos y a las acciones les llamamos métodos. Así de simple: objeto = atributos + métodos.

Y aquí viene lo bonito: tú ya has usado objetos durante todo el curso sin darte cuenta. Cuando escribiste `"hola".upper()`, ese texto `"hola"` era un objeto de tipo string, y `upper()` era un método suyo, una acción que el objeto sabe hacer. Cuando usaste `mi_lista.append(5)`, `mi_lista` era un objeto de tipo lista y `append` era su método. Python está construido con objetos por todos lados: los strings, las listas, los diccionarios, incluso los números son objetos. Todo este tiempo estuviste usando los moldes que otros fabricaron. Hoy aprendes a fabricar los tuyos.

La clase: el molde que fabrica objetos

Si el objeto es la galleta, la clase es el molde. Si el objeto es la casa construida, la clase es el plano del arquitecto. La clase es la definición, la receta, el diseño: dice qué atributos va a tener cada objeto y qué métodos va a poder ejecutar. Pero ojo con esto, porque es la confusión número uno de los principiantes: la clase en sí no es el objeto. El plano no es la casa. El molde no es la galleta. La clase es solo la descripción de cómo van a ser los objetos que se creen a partir de ella.

Mira la sintaxis. Es sorprendentemente sencilla:

```
class Perro:
    def ladrar(self):
        print("¡Guau!")
```

Eso es todo. La palabra `class` (en minúsculas, en inglés) le dice a Python "voy a definir un molde nuevo". Después viene el nombre de la clase, que por convención se escribe con la

primera letra en mayúscula (a esto se le llama CapWords o CamelCase: Perro, CuentaBancaria, Estudiante). Los dos puntos y el bloque indentado contienen la definición, igual que en las funciones y los condicionales.

Dentro de la clase definimos métodos con `def`, exactamente igual que las funciones normales. La única diferencia visible por ahora es ese parámetro misterioso llamado `self` que aparece primero. No te preocupes: en dos secciones lo vamos a destripar sin miedo. Por lo pronto, quédate con la idea: una clase es un bloque que agrupa datos y funciones relacionadas bajo un mismo nombre.

Ahora, la magia: crear objetos a partir de la clase. A esto se le llama instanciar (crear una instancia):

```
class Perro:
    def ladrar(self):
        print("¡Guau!")

firulais = Perro()
solovino = Perro()

firulais.ladrar()
solovino.ladrar()
```

`Perro()` parece una llamada a función, y en cierto sentido lo es: le estás pidiendo a la clase que fabrique un objeto nuevo. `firulais` y `solovino` son dos objetos distintos, dos instancias independientes de la misma clase. Cada uno puede ladrar por su cuenta. Si le cambias algo a `firulais`, `solovino` ni se entera. Son galletas del mismo molde, pero cada una es suya.

Fíjate en la sintaxis para usar un método: `firulais.ladrar()`. El punto significa "el método `ladrar` del objeto `firulais`". Esa notación de punto la has usado todo el curso con strings y listas; ahora entiendes por qué existe.

`__init__`: el constructor, o cómo nace configurado cada objeto

El ejemplo del perro que ladra está bien para empezar, pero tiene un problema: todos los perros son idénticos. En el mundo real, cada perro tiene un nombre, una edad, una raza. ¿Cómo le damos a cada objeto sus propios datos al momento de crearlo? Con un método especial llamado `__init__`, el constructor.

El constructor es un método que Python ejecuta automáticamente cada vez que creas un objeto nuevo. Su trabajo es configurar al recién nacido: recibir los datos iniciales y guardarlos

como atributos del objeto. Se escribe así, con doble guion bajo al inicio y al final (a eso se le llama "dunder", de double underscore):

```
class Perro:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def ladrar(self):
        print(f"¡Guau! Soy {self.nombre}")

firulais = Perro("Firulais", 3)
solovino = Perro("Solovino", 5)

print(firulais.nombre)    # Firulais
print(solovino.nombre)    # Solovino
firulais.ladrar()          # ¡Guau! Soy Firulais
solovino.ladrar()          # ¡Guau! Soy Solovino
```

Vamos por partes, porque aquí hay tres ideas juntas. Primera: `Perro("Firulais", 3)` le pasa los datos al constructor. Python toma esos argumentos y se los entrega a `__init__` automáticamente. Tú nunca llamas a `__init__` directamente; Python lo invoca por ti en el momento de la creación. Es como el acta de nacimiento del objeto: se llena una sola vez, al nacer.

Segunda: `self.nombre = nombre` guarda el dato dentro del objeto. Del lado derecho está el parámetro que llegó (nombre, el valor "Firulais"); del lado izquierdo está `self.nombre`, que significa "el atributo nombre de este objeto en particular". Esa línea es la que hace que cada objeto tenga sus propios datos: el "Firulais" queda guardado dentro del objeto `firulais` y el "Solovino" dentro del objeto `solovino`. No se mezclan porque cada uno vive en su propio objeto.

Tercera: fíjate que al crear el objeto pasaste dos argumentos ("Firulais", 3) pero `__init__` recibe tres parámetros (self, nombre, edad). No es un error: `self` lo pone Python automáticamente. Tú no lo pasas; Python se encarga. Esto nos lleva directo a la siguiente sección, la que más miedo da y menos debería dar.

self: el pronombre del objeto, explicado sin miedo

`self` es, sin exagerar, el concepto que más confunde a los principiantes en Python, y es una lástima porque la idea es sencilla. `self` significa "yo mismo", "este objeto". Es la forma en que un objeto se refiere a sí mismo desde adentro.

Piensa en cómo hablamos las personas. Si yo digo "mi nombre es Iv", la palabra "mi" se refiere a mí, a quien está hablando. Si tú dices "mi nombre es Esteban", la misma palabra "mi" ahora se refiere a ti. La palabra no cambió; lo que cambió fue quién la está usando. `self` funciona igual: cuando el código dentro de la clase dice `self.nombre`, significa "el nombre de este objeto, del que está actuando en este momento".

Mira este ejemplo con lupa:

```
class Perro:
    def __init__(self, nombre):
        self.nombre = nombre

    def presentarse(self):
        print(f"Hola, me llamo {self.nombre}")

firulais = Perro("Firulais")
solovino = Perro("Solovino")

firulais.presentarse()    # Hola, me llamo Firulais
solovino.presentarse()    # Hola, me llamo Solovino
```

Cuando ejecutas `firulais.presentarse()`, Python hace algo entre bambalinas: toma el objeto `firulais` y lo pasa como el primer argumento del método. Es decir, `firulais.presentarse()` es equivalente a `Perro.presentarse(firulais)`. Por eso el primer parámetro del método se llama `self`: ahí llega el objeto que está llamando. Cuando es `firulais` quien llama, `self` es `firulais` y `self.nombre` vale "Firulais". Cuando es `solovino`, `self` es `solovino` y `self.nombre` vale "Solovino". Mismo método, mismo código, resultados distintos según quién lo use. Esa es toda la magia.

Tres reglas prácticas para no sufrir con `self`:

1. Siempre va primero en la definición de los métodos: `def metodo(self, otros_parametros):`. Sin excepción.
2. Nunca lo pasas tú al llamar el método: escribes `firulais.presentarse()`, no `firulais.presentarse(firulais)`. Python lo inyecta solo.

3. Úsalo para guardar y leer los datos del objeto: `self.nombre = nombre` para guardar, `self.nombre` para leer. Si olvidas el `self` y escribes solo `nombre = nombre`, el dato se pierde en cuanto termina el método, porque era una variable local temporal, no un atributo del objeto.

¿Y por qué se llama `self` y no de otra forma? Por convención. Podrías llamarlo `yo`, `este` o `pepito` y el código funcionaría igual, porque lo que importa es que sea el primer parámetro. Pero no lo hagas: todo el mundo usa `self`, todo el código del planeta lo usa, y si usas otro nombre solo vas a confundir a quien lea tu código (incluido tú mismo dentro de tres meses).

Atributos: las características del objeto

Los atributos son las variables que viven dentro de un objeto. Describen cómo es: su estado, sus datos, sus características. En el ejemplo del perro, `nombre` y `edad` son atributos. Se crean asignándolos con `self` dentro de los métodos (normalmente en `__init__`) y se leen con la notación de punto desde fuera.

```
class Estudiante:
    def __init__(self, nombre, carrera, semestre):
        self.nombre = nombre
        self.carrera = carrera
        self.semestre = semestre

ana = Estudiante("Ana", "Ingeniería", 3)
print(ana.nombre)      # Ana
print(ana.carrera)     # Ingeniería

# También puedes modificarlos directamente:
ana.semestre = 4
print(ana.semestre)    # 4
```

Cada objeto guarda su propia copia de los atributos. `ana.nombre` y el nombre de otro estudiante son cajones distintos en muebles distintos. Cambiar el semestre de Ana no afecta a nadie más. Esta independencia es una de las grandes virtudes de los objetos: cada uno cuida sus propios datos y nadie los pisa por accidente.

Un detalle importante: los atributos no tienen que crearse todos en `__init__`. Puedes agregar atributos en cualquier método, e incluso desde fuera del objeto. Pero la buena práctica es declararlos todos en el constructor, para que cualquier persona que lea tu clase sepa de un vistazo qué datos maneja cada objeto. Piensa en `__init__` como la ficha de registro del objeto: ahí debe quedar claro qué información trae de fábrica.

Métodos: los comportamientos del objeto

Si los atributos describen cómo es el objeto, los métodos describen qué puede hacer. Son funciones definidas dentro de la clase, y la diferencia clave con las funciones normales es que trabajan con los datos del objeto a través de `self`.

```
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.titular = titular
        self.saldo = saldo_inicial

    def depositar(self, cantidad):
        self.saldo = self.saldo + cantidad
        print(f"Depósito de ${cantidad}. Nuevo saldo: ${self.saldo}")

    def retirar(self, cantidad):
        if cantidad <= self.saldo:
            self.saldo = self.saldo - cantidad
            print(f"Retiro de ${cantidad}. Nuevo saldo: ${self.saldo}")
        else:
            print("Fondos insuficientes. No se puede retirar esa cantidad.")

    def consultar(self):
        print(f"{self.titular}, tu saldo es: ${self.saldo}")
```

Observa cómo cada método usa `self.saldo` para leer y modificar el dinero de esa cuenta en particular. El método `retirar` incluso toma decisiones con un `if`, igual que cualquier código que ya sabes escribir: los métodos pueden contener condicionales, ciclos, llamadas a otros métodos, todo lo que aprendiste en el curso. La clase es solo una forma nueva de organizar lo que ya sabes.

Y aquí está la belleza del asunto: desde fuera, usar la cuenta es facilísimo.

```
cuenta = CuentaBancaria("Esteban", 1000)
cuenta.consultar()    # Esteban, tu saldo es: $1000
cuenta.depositar(500) # Depósito de $500. Nuevo saldo: $1500
cuenta.retirar(2000)  # Fondos insuficientes. No se puede retirar esa cantidad.
cuenta.retirar(300)   # Retiro de $300. Nuevo saldo: $1200
```

Quien usa la cuenta no necesita saber cómo se guarda el saldo ni qué validaciones hay por dentro. Solo llama a los métodos y la cuenta se encarga del resto. A esto se le llama encapsulamiento, una palabra elegante para una idea simple: cada objeto es responsable de

sus propios datos y ofrece una forma clara de interactuar con ellos. Es como el cajero automático: tú metes tu tarjeta y pides dinero; no necesitas saber cómo el banco mueve los billetes por dentro.

Varios objetos independientes: el verdadero poder del molde

Hasta aquí hemos visto clases con uno o dos objetos, pero el poder real aparece cuando creas muchos. Imagina que estás programando el sistema de una tienda y necesitas manejar 50 productos. Sin clases, tendrías que crear variables sueltas para cada uno: producto1_nombre, producto1_precio, producto2_nombre... una pesadilla de copiar y pegar donde un solo error de dedo rompe todo. Con una clase, defines el molde una vez y fabricas los 50 productos en un ciclo:

```
class Producto:
    def __init__(self, nombre, precio, existencia):
        self.nombre = nombre
        self.precio = precio
        self.existencia = existencia

    def vender(self, cantidad):
        if cantidad <= self.existencia:
            self.existencia = self.existencia - cantidad
            total = cantidad * self.precio
            print(f"Vendidos {cantidad} {self.nombre}. Total: ${total}")
        else:
            print(f"Solo hay {self.existencia} {self.nombre} disponibles.")

datos = [("Tacos", 15, 100), ("Refresco", 20, 50), ("Agua", 12, 80)]
tienda = []
for nombre, precio, existencia in datos:
    tienda.append(Producto(nombre, precio, existencia))

tienda[0].vender(3)
tienda[1].vender(60)
```

Tres productos creados en tres líneas de ciclo, cada uno con sus propios datos, cada uno respondiendo a vender con su propia información. Si mañana la tienda tiene 500 productos, el código no crece: solo cambia la lista de datos. El molde se escribe una vez y se usa mil veces. Esa es la economía de la POO.

Y algo crucial: los objetos son independientes de verdad. Mira:

```

cuenta1 = CuentaBancaria("Ana", 1000)
cuenta2 = CuentaBancaria("Luis", 500)

cuenta1.depositar(200)
print(cuenta2.saldo) # 500, no se movió ni un peso

```

Depositar en la cuenta de Ana no toca la cuenta de Luis. Cada objeto tiene su propio saldo, su propio titular, su propia vida. En programación esto elimina una familia entera de errores: esos bugs misteriosos donde cambiar una variable en una parte del programa rompía otra parte sin que entendieras por qué. Con objetos, los datos viven encerrados dentro de su dueño.

__str__: que tus objetos se presenten bonito

Hay un detalle molesto que ya habrás notado si probaste los ejemplos: cuando imprimes un objeto directamente, Python te muestra algo críptico.

```

firulais = Perro("Firulais", 3)
print(firulais)
# <__main__.Perro object at 0x7f8b2c4d3a90>

```

Ese galimatías es la representación interna del objeto: su tipo y su dirección en memoria. Útil para la computadora, inútil para ti. Para arreglarlo existe otro método especial: `__str__`. Python lo llama automáticamente cada vez que conviertes el objeto a texto, ya sea con `print` o con un f-string. Tú defines cómo quieres que se vea:

```

class Perro:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"Perro(nombre='{self.nombre}', edad={self.edad})"

firulais = Perro("Firulais", 3)
print(firulais)
# Perro(nombre='Firulais', edad=3)

```

Fíjate en dos cosas. Primera: `__str__` debe devolver un string con `return`, no imprimirlo con `print`. Python se encarga de mostrarlo. Segunda: igual que `__init__`, nunca lo llamas

directamente; Python lo invoca por ti cuando hace falta. Los métodos con doble guion bajo son como empleados invisibles: trabajan entre bambalinas cada vez que haces algo cotidiano como imprimir o crear un objeto. No necesitas memorizarlos todos ahora; con `__init__` y `__str__` tienes más que suficiente para escribir clases profesionales.

Ejemplo completo comentado: del molde simple al sistema útil

Vamos a juntar todo en un ejemplo más serio, comentado línea por línea, para que veas cómo se ve una clase bien escrita en la vida real. Usaremos un `Estudiante` que calcula su propio promedio, porque combina atributos, métodos, un ciclo y un condicional: todo tu curso en miniatura.


```

class Estudiante:
    # El constructor: recibe los datos iniciales del estudiante.
    # 'calificaciones' será una lista de números, puede venir vacía.
    def __init__(self, nombre, matricula, calificaciones):
        self.nombre = nombre
        self.matricula = matricula
        self.calificaciones = calificaciones

    # Método que agrega una calificación a la lista del estudiante.
    def agregar_calificacion(self, calificacion):
        self.calificaciones.append(calificacion)
        print(f"Calificación {calificacion} registrada para {self.nombre}.")

    # Método que calcula el promedio. Si no hay calificaciones,
    # evita la división entre cero devolviendo 0.
    def promedio(self):
        if len(self.calificaciones) == 0:
            return 0
        return sum(self.calificaciones) / len(self.calificaciones)

    # Método que decide si el estudiante aprueba (promedio >= 6).
    def esta_aprobado(self):
        return self.promedio() >= 6

    # Presentación bonita del objeto al imprimirlo.
    def __str__(self):
        estado = "aprobado" if self.esta_aprobado() else "reprobado"
        return f"{self.nombre} (mat. {self.matricula}): promedio {self.promedio():.1f}, {estado}."

# Creamos dos estudiantes independientes con el mismo molde.
ana = Estudiante("Ana", "A001", [9, 8, 10])
luis = Estudiante("Luis", "A002", [5, 4])

ana.agregar_calificacion(7)
print(ana)    # Ana (mat. A001): promedio 8.5, aprobado.
print(luis)   # Luis (mat. A002): promedio 4.5, reprobado.

```

Detente un momento a admirar lo que pasó aquí. La clase `Estudiante` encapsula todo lo que un estudiante sabe hacer: guardar calificaciones, calcular su promedio, decidir si aprueba y presentarse bonito. Desde fuera, el código que usa la clase es limpio y se lee casi como español: "ana, agrega calificación 7", "imprime a ana". Nada de listas sueltas, nada de

funciones que reciben diez parámetros, nada de variables globales. Cada estudiante cuida sus propios datos. Si la escuela crece a diez mil estudiantes, este mismo código sirve: solo creas más objetos.

Nota el detalle de `self.promedio()` dentro de `esta_aprobado`: un método puede llamar a otro método del mismo objeto usando `self`. Y en `__str__` usamos un f-string con formato `:.1f` para mostrar el promedio con un decimal, un truco elegante que aprendiste en la Unidad 8.

Por qué la POO organiza los programas grandes

Ya tienes la intuición, pero vale la pena decirlo explícito, porque es la razón por la que la POO domina la industria: los programas grandes se vuelven inmanejables cuando los datos andan sueltos por todos lados. Imagina un programa de banco escrito sin clases: una lista de nombres, otra lista de saldos, otra de números de cuenta, y funciones que reciben índices para saber de qué cliente hablan. Agregar un cliente nuevo significa tocar tres listas al mismo tiempo y rezar por no equivocarte de índice. Encontrar un error significa rastrear qué función modificó qué lista y cuándo. A eso los programadores le dicen "código espagueti", y es la forma más rápida de odiar tu trabajo.

Con clases, cada cuenta es un paquete cerrado: sus datos y sus operaciones viajan juntos. Quieres saber todo lo que una cuenta puede hacer: lees la clase `CuentaBancaria`. Quieres cambiar cómo se calculan los intereses: modificas un método en un solo lugar y las diez mil cuentas del sistema heredan el cambio automáticamente, igual que el pastelero que retoca el molde una vez. Quieres agregar un tipo nuevo de cuenta: creas otra clase sin tocar la que ya funciona.

Esta organización tiene nombre y apellido en la industria: es la base del diseño de software serio. No necesitas aprender los términos elegantes todavía (herencia, polimorfismo y esas palabras que suenan a examen), pero quédate con la idea central: la POO te deja pensar en piezas, no en instrucciones sueltas. Y pensar en piezas es como piensan los arquitectos, los ingenieros y toda la gente que construye cosas grandes que no se caen.

Errores comunes de POO (y cómo esquivarlos)

Cada tema nuevo trae sus trampas clásicas. Estas son las de la POO, para que las reconozcas cuando aparezcan:

El error número uno es olvidar `self` en la definición de un método. Escribes `def ladrar():` en vez de `def ladrar(self):` y al llamar `firulais.ladrar()` Python te responde con un

`TypeError` quejándose de los argumentos. Recuerda: `self` siempre va primero en la definición, aunque el método no reciba nada más.

El error número dos es confundir la clase con el objeto. Intentar hacer `Perro.nombre` en vez de `firulais.nombre`, o llamar `Perro.ladrazar()` sin un objeto. La clase es el plano: no tiene nombre de perro ni puede ladrar. Solo los objetos, las casas construidas, tienen datos y hacen cosas. Si Python te dice que algo "no existe" en la clase, casi seguro estás hablando con el molde en vez de con la galleta.

El error número tres es escribir mal `__init__`: con un solo guion bajo (`_init_`), o con guiones de más o de menos, o llamándolo directamente como `firulais._init_()`. Si el constructor está mal escrito, Python no lo reconoce como constructor y tus objetos nacen vacíos, sin atributos, y todo lo demás falla en cascada con `AttributeError`. Son dos guiones bajos antes y dos después, y se invoca solo.

El error número cuatro es olvidar el `self` al guardar atributos: escribir `nombre = nombre` dentro de `__init__` en vez de `self.nombre = nombre`. El código corre sin errores, pero el dato se evapora cuando el método termina, porque era una variable local. Después, cuando otro método intente leer `self.nombre`, Python dirá que el atributo no existe. Regla de oro: todo dato que el objeto deba recordar lleva `self.` por delante.

Y el error número cinco es crear atributos fuera del constructor sin necesidad, regándolos por varios métodos. Funciona, pero tu clase se vuelve un misterio: nadie sabe qué datos maneja el objeto sin leerla completa. Declara todo en `__init__` y tu yo del futuro te lo va a agradecer.

Buenas prácticas con clases

Un nombre de clase claro vale oro: `CuentaBancaria` dice todo, `Cosa1` no dice nada. Usa `CapWords` (cada palabra con mayúscula inicial, sin guiones bajos) y sustantivos: las clases son cosas, no acciones. Los métodos, en cambio, son acciones: usa verbos en minúsculas con `snake_case`, como `depositar`, `calcular_promedio`, `agregar_calificacion`.

Mantén cada clase enfocada en una sola responsabilidad. Una clase `Estudiante` que maneja calificaciones está bien; una clase `Estudiante` que además imprime reportes en PDF, envía correos y cocina el desayuno es una señal de que necesitas más clases. Si un método no usa `self` para nada, pregúntate si de verdad pertenece a la clase o si debería ser una función normal.

Documenta con comentarios qué representa la clase y qué hace cada método no obvio, sobre todo el constructor: ¿qué parámetros recibe y qué significan? Y por último, prueba tus clases

en pequeño antes de usarlas en grande: crea un objeto, llama a cada método, imprime los atributos, rompe algo a propósito. Cinco minutos de pruebas te ahorran cinco horas de depuración.

El proyecto final: tu primer juego completo, "adivina el número"

Todo lo que aprendiste en nueve unidades fue entrenamiento. Esto es el partido. Vas a construir un juego real, funcional, que podrías presumirle a quien sea: la computadora piensa un número secreto entre 1 y 100, tú intentas adivinarlo, y después de cada intento ella te dice si tu número fue "muy alto" o "muy bajo", hasta que lo encuentras. Al final te dice en cuántos intentos lo lograste.

Antes de escribir código, todo buen programador diseña. No se trata de ser formal ni de hacer diagramas bonitos: se trata de responder tres preguntas antes de tocar el teclado. Primera: ¿qué necesita el juego? Un número secreto aleatorio (módulo `random`, Unidad 9), una forma de pedirle intentos al usuario (`input`, Unidad 3), convertir ese texto a número (`int`, Unidad 3), comparar el intento con el secreto (condicionales, Unidad 4), repetir hasta adivinar (ciclo `while`, Unidad 5), contar los intentos (un contador con `+=`, Unidad 2) y dar pistas de "más alto / más bajo". Segunda: ¿cuándo termina? Cuando el intento es igual al secreto. Tercera: ¿qué puede salir mal? Que el usuario escriba letras en vez de números, así que necesitamos manejo de errores (`try/except`, Unidad 8). ¿Ves? Cada pieza del diseño apunta a una unidad del curso. Nada de lo que aprendiste fue de adorno.

Aquí está el código completo, comentado línea por línea. Léelo despacio; cada comentario es una mini-lección.

```

import random # Traemos el módulo random (Unidad 9) para generar el secreto.

# El número secreto: un entero aleatorio entre 1 y 100, ambos incluidos.
secreto = random.randint(1, 100)

# Contador de intentos. Empieza en cero y crece con cada intento válido.
intentos = 0

print("¡Adivina el número! Estoy pensando en un número del 1 al 100.")

# El ciclo sigue mientras el jugador no adivine. La variable 'adivinado'
# es la bandera que controla la salida del ciclo.
adivinado = False
while not adivinado:
    # Pedimos el intento. Todo lo que entra por input llega como texto.
    entrada = input("Tu intento: ")

    # Validamos: si no es un número, avisamos y el ciclo vuelve a empezar
    # sin contar el intento (gracias al 'continue').
    try:
        numero = int(entrada)
    except ValueError:
        print("Eso no es un número válido. Intenta de nuevo.")
        continue

    # Solo los intentos válidos cuentan.
    intentos = intentos + 1

    # Comparamos y damos la pista correspondiente.
    if numero < secreto:
        print("Muy bajo. Intenta con un número más alto.")
    elif numero > secreto:
        print("Muy alto. Intenta con un número más bajo.")
    else:
        # ¡Adivinó! Levantamos la bandera y el ciclo termina.
        adivinado = True
        print(f"¡Correcto! El número era {secreto}.")
        print(f"Lo adivinaste en {intentos} intentos.")

```

Ejecútalo mentalmente una vez para sentir cómo fluye. La computadora elige, digamos, el 42. Tú escribes "50": el programa convierte el texto a número, cuenta el intento, compara $50 > 42$ e imprime "Muy alto". Escribes "30": $30 < 42$, "Muy bajo". Escribes "hola": el `try` falla, el `except` atrapa el error, imprime el aviso y el `continue` salta el resto del ciclo sin contar el

intento. Escribes "42": cae en el `else`, levanta la bandera, te felicita y te dice tus intentos. El ciclo verifica `while not adivinado`, ve que ya es `True` y termina. Juego completo, sin errores, con validación incluida.

Fíjate en las decisiones de diseño escondidas en ese código. El contador solo aumenta después de la validación, para no castigar al jugador por un error de dedo: eso es pensar en el usuario. La bandera `adivinado` hace que la condición de salida se lea en español casi literal ("mientras no adivinado"). El `try/except` protege exactamente la línea peligrosa (la conversión), no todo el programa a lo loco. Estos son los detalles que separan un programa que "funciona" de un programa bien hecho, y ya los estás aplicando.

Mejoras posibles: de juego sencillo a juego en serio

Un proyecto nunca está terminado, solo está en su versión actual. Aquí tienes tres mejoras naturales, ordenadas de fácil a avanzada, para que elijas tu propia aventura.

La primera mejora es el límite de intentos. Ahora mismo el jugador tiene intentos infinitos; con un límite, cada intento duele y el juego se pone emocionante. Solo necesitas una variable con el máximo y una condición extra en el ciclo:

```
MAX_INTENTOS = 7
intentos = 0
adivinado = False

while not adivinado and intentos < MAX_INTENTOS:
    # ... (pedir y validar el intento como antes) ...
    intentos = intentos + 1
    # ... (comparar y dar pistas como antes) ...

if not adivinado:
    print(f"¡Se acabaron los intentos! El número era {secreto}.")
```

La condición `intentos < MAX_INTENTOS` se evalúa en cada vuelta: si el jugador agota sus 7 intentos sin adivinar, el ciclo termina y el mensaje final revela el secreto. Siete intentos no son arbitrarios: con pistas de "más alto / más bajo" y búsqueda inteligente, cualquier número del 1 al 100 se encuentra en 7 intentos como máximo. Si no me crees, investiga la "búsqueda binaria": es tu primera probada de los algoritmos de verdad.

La segunda mejora son los niveles de dificultad. Fácil: del 1 al 50 con 10 intentos. Normal: del 1 al 100 con 7 intentos. Difícil: del 1 al 500 con 8 intentos. El jugador elige al inicio y el programa configura el rango y el límite según la elección. Esto te obliga a usar `if/elif/else`

para la configuración inicial, y de paso aprendes una lección de diseño: los buenos programas separan la configuración (qué valores usar) de la lógica (cómo jugar). Si mañana quieres agregar el nivel "legendario", solo tocas la configuración.

La tercera mejora conecta directo con la Unidad 9: guardar el puntaje en un archivo. Imagina que al ganar, el juego le pide su nombre al jugador y guarda su marca (nombre, dificultad, intentos) en un archivo de texto, y al iniciar muestra la tabla de mejores puntajes. De pronto tu juego tiene memoria: los récords sobreviven aunque cierres el programa. Necesitas abrir el archivo en modo lectura para cargar los puntajes, mostrarlos, y abrirlo en modo escritura (o agregado) para guardar el nuevo récord. Es el mismo `open`, `read`, `write` y `close` que practicaste, ahora al servicio de un juego de verdad. Y si quieres ponerte elegante, guarda los puntajes con el formato `nombre,dificultad,intentos` separados por comas, y al leerlos usa `split(",")` para reconstruir la tabla: estarás inventando tu propia mini-base de datos.

Cada mejora es un mini-proyecto en sí misma. Mi recomendación: implementa primero el límite de intentos (es la más corta), luego la dificultad y al final el archivo de puntajes. En ese orden, cada versión del juego sigue funcionando mientras construyes la siguiente. Así trabajan los profesionales: por versiones, nunca rompiendo lo que ya sirve.

Repaso general: el mapa completo de lo que ya dominas

Antes de hablar del futuro, miremos el camino recorrido. Es fácil subestimar lo aprendido cuando lo ves por partes; míralo junto y vas a entender por qué puedes llamarte programador sin que te tiemble la voz.

En la Unidad 1 encendiste la estufa: qué es Python, cómo instalarlo y tu primer `print`. Parece poco, pero ahí venciste la barrera más grande de todas: pasar de "quiero aprender" a "ya escribí código que corre". En la Unidad 2 llenaste la alacena: variables, tipos de datos y operadores. Aprendiste que `=` asigna y no compara, que los tipos importan y que Python puede ser tu calculadora. En la Unidad 3 tus programas aprendieron a hablar: `input` para preguntar, conversiones para no romper nada y f-strings para responder con mensajes elegantes.

En la Unidad 4 les diste cerebro: condicionales `if/elif/else`, comparaciones y operadores lógicos. Tus programas dejaron de ser recetas lineales y empezaron a tomar decisiones. En la Unidad 5 les diste resistencia: ciclos `while` y `for`, y las listas, tu primera estructura para guardar colecciones de datos. En la Unidad 6 ampliaste el arsenal: métodos de listas, tuplas, diccionarios y sets. Aprendiste que no hay una sola forma de guardar datos, sino la forma correcta para cada caso.

En la Unidad 7 aprendiste a no repetirte: `sets`, funciones, parámetros y `return`. Descubriste que un buen programador escribe una función una vez y la usa cien veces. En la Unidad 8 puliste los detalles: `strings` y `f-strings` a fondo, métodos de texto y manejo de errores con `try/except`, la habilidad que separa los programas de juguete de los programas que aguantan usuarios reales. En la Unidad 9 conectaste tu código con el mundo: archivos para guardar información permanente, módulos con `import` para no reinventar la rueda y `list comprehensions` para escribir código compacto y elegante.

Y en esta Unidad 10 cerraste el círculo: clases y objetos para organizar programas grandes como piensa el mundo real, y un proyecto completo que usa todo lo anterior. Diez unidades, treinta clases, un lenguaje completo en tus manos. Cuando alguien te pregunte qué sabes hacer, ya no digas "estoy aprendiendo Python". Di: "programo en Python". Porque es verdad.

Tu siguiente paso: las rutas que se abren

Graduarte no es el final del camino; es el momento en que el camino se bifurca y tú eliges. Python es famoso por tener más salidas profesionales que casi cualquier otro lenguaje. Estas son las rutas principales, con qué sigue en cada una.

La ruta de los proyectos reales es la que yo te recomiendo primero, sin importar a dónde quieras llegar después. La teoría sin proyectos se oxida; los proyectos sin teoría se atoran. Necesitas construir cosas que te importen. Aquí van cinco ideas concretas, ordenadas de más accesible a más ambiciosa: primero, un administrador de gastos personales que guarde tus movimientos en un archivo y te muestre cuánto gastaste por categoría usando diccionarios y clases (junta las Unidades 6, 9 y 10 en algo útil para tu vida real). Segundo, un generador de contraseñas seguras con el módulo `random` y `string`, que te pida la longitud y los tipos de caracteres: corto, útil y perfecto para practicar validación. Tercero, una agenda de contactos completa con clases y archivo, como la que verás en los ejemplos de esta unidad, pero extendida por ti: búsqueda, edición y borrado. Cuarto, un scraper sencillo que descargue una página web y extraiga información con las librerías `requests` y `beautifulsoup4`: tu primer contacto con datos del internet real. Quinto, un bot de Telegram que responda mensajes automáticamente: combina Python con una API real y te enseña cómo hablan los programas entre sí por internet. Cada proyecto terminado vale más que diez tutoriales vistos, porque los proyectos te obligan a tomar decisiones, y decidir es lo que hacen los programadores.

La ruta de la inteligencia artificial es la más nombrada y con razón: Python es el lenguaje rey de la IA. Pero ojo, la IA no es magia ni es "aprender prompts": por debajo hay matemáticas y datos, y Python es la herramienta para trabajarlos. Tu mapa de aprendizaje sería: primero `numpy`, la librería para hacer cálculos numéricos rápidos con arreglos (es la base de todo lo

demás); después `pandas`, para manejar tablas de datos como un profesional, limpiarlos y analizarlos; luego `matplotlib`, para graficar esos datos y verlos con tus ojos; y finalmente `scikit-learn`, donde entrenarás tus primeros modelos de machine learning de verdad: predecir precios, clasificar mensajes, recomendar productos. No necesitas ser un genio de las matemáticas para empezar, pero sí necesitarás repasar estadística básica en el camino. Es una ruta larga y muy bien pagada.

La ruta del desarrollo web te convierte en constructor de sitios y aplicaciones de internet. Python tiene dos frameworks gigantes: `Flask`, ligero y perfecto para aprender cómo funciona un servidor web por dentro, y `Django`, completo y robusto, el que usan empresas grandes para aplicaciones serias. El camino natural es: aprende cómo funciona HTTP a grandes rasgos, construye tu primera página dinámica con `Flask`, aprende bases de datos con SQL (spoiler: en lucasegaacademy.com también hay curso de SQL, gratis como todo lo demás) y luego salta a `Django` para proyectos serios. Si además aprendes HTML, CSS y JavaScript, te conviertes en desarrollador full-stack, uno de los perfiles más buscados.

La ruta de la automatización es la más silenciosa y una de las más rentables: usar Python para que la computadora haga el trabajo aburrido por ti. Renombrar mil archivos de un jalón, llenar reportes de Excel automáticamente con `openpyxl`, descargar y organizar información, enviar correos masivos personalizados, monitorear precios de productos y avisarte cuando bajen. Las empresas pagan muy bien por esto porque cada script de automatización les ahorra horas de trabajo humano cada semana. Y es la ruta donde más rápido ves resultados: en una tarde puedes escribir algo que te ahorre una hora diaria para siempre.

No tienes que elegir una ruta para siempre. La mayoría de los programadores prueban varias antes de especializarse, y todas comparten la misma base que ya tienes. Mi consejo: dedica los próximos tres meses a proyectos reales mientras exploras una ruta en tus ratos libres. Los proyectos te dan portafolio, la ruta te da dirección, y juntos te dan algo que ningún curso puede darte: experiencia.

Consejos para seguir aprendiendo (los que nadie te dice al graduarte)

Primero: abre tu portafolio en GitHub hoy mismo, no "cuando estés listo". GitHub es la red social de los programadores y tu currículum vivo: ahí subes tus proyectos para que el mundo (y los reclutadores) vean lo que sabes hacer. No esperes a tener proyectos perfectos; sube el juego de adivina el número, súbelo con sus mejoras, documenta qué aprendiste en cada uno. Un GitHub con cinco proyectos honestos y bien explicados vale más que un certificado bonito. Aprende lo básico de `git` (guardar cambios, subirlos) en una tarde y úsalo desde tu próximo proyecto.

Segundo: programa todos los días, aunque sea veinte minutos. La programación es como un idioma o un músculo: se mantiene con uso diario y se oxida con pausas largas. No necesitas sesiones maratónicas; necesitas constancia. Veinte minutos diarios durante un año te ponen por delante del 90% de la gente que "alguna vez tomó un curso".

Tercero: lee código ajeno. Al principio cuesta, como leer en otro idioma, pero es la forma más rápida de subir de nivel. Entra a GitHub, busca proyectos pequeños en Python, lee cómo otros resolvieron problemas que tú ya resolviste y compara. Vas a descubrir trucos, estilos y formas de pensar que ningún libro te enseña. Cuando algo no entiendas, cópialo, ejecútalo, modifícalo y observa: el mismo método de "romper para aprender" que usaste en este curso.

Cuarto: enseña lo que sabes. Explícale a alguien qué es una clase, escribe un tutorial, responde preguntas en foros. Enseñar te obliga a ordenar tus ideas y revela los huecos en tu comprensión mejor que cualquier examen. Además, la comunidad de Python es famosa por ser amable con los principiantes: hoy preguntas, mañana respondes.

Y quinto: nunca dejes de construir. Los cursos te dan herramientas; los proyectos te hacen programador. Cada vez que aprendas algo nuevo, pregúntate "¿qué puedo construir con esto?" y constrúyelo, aunque sea feo, aunque sea pequeño. El programador que construye cien cosas pequeñas le gana siempre al que planea una cosa perfecta que nunca empieza.

Ejemplos paso a paso

Ejemplo 1: Tu primera clase, el Perro básico

Empecemos por lo más simple posible: una clase con un constructor y un método. Escríbelo, ejecútalo y juega con él antes de seguir.

```

# Definimos el molde: la clase Perro.
class Perro:
    # El constructor recibe el nombre y la raza al crear cada perro.
    def __init__(self, nombre, raza):
        # Guardamos los datos como atributos del objeto (con self).
        self.nombre = nombre
        self.raza = raza

    # Un método: el comportamiento de ladrar usando sus propios datos.
    def ladrar(self):
        print(f"{self.nombre} dice: ¡Guau guau!")

# Fabricamos dos objetos independientes con el mismo molde.
firulais = Perro("Firulais", "Mestizo")
rocky = Perro("Rocky", "Bulldog")

# Cada objeto usa sus propios atributos.
firulais.ladrar()    # Firulais dice: ¡Guau guau!
rocky.ladrar()       # Rocky dice: ¡Guau guau!
print(rocky.raza)    # Bulldog

```

Línea por línea: `class Perro:` abre la definición del molde. `def __init__(self, nombre, raza):` declara el constructor, que Python llamará solo al crear cada objeto. Las dos líneas con `self` guardan los datos dentro del objeto. `def ladrar(self):` define el comportamiento, que lee `self.nombre` para personalizar el mensaje. Al crear `firulais = Perro("Firulais", "Mestizo")`, Python ejecuta `__init__` con esos valores (el `self` lo pone Python). Los dos perros comparten el molde pero no los datos: cambia `firulais.nombre` y `rocky` ni se entera. Para practicar: agrega un atributo `edad` y un método `cumplir_anos` que sume 1 a la edad.

Ejemplo 2: CuentaBancaria con métodos que cuidan el dinero

Ahora una clase útil de verdad, con lógica de negocio adentro: validaciones, decisiones y estado que cambia con el tiempo.

```

class CuentaBancaria:
    def __init__(self, titular, saldo_inicial=0):
        self.titular = titular
        self.saldo = saldo_inicial
        # Historial de movimientos: cada objeto guarda su propia lista.
        self.movimientos = []

    def depositar(self, cantidad):
        if cantidad <= 0:
            print("La cantidad debe ser positiva.")
            return
        self.saldo += cantidad
        self.movimientos.append(f"Depósito: +${cantidad}")
        print(f"{self.titular} depositó ${cantidad}. Saldo: ${self.saldo}")

    def retirar(self, cantidad):
        if cantidad <= 0:
            print("La cantidad debe ser positiva.")
            return
        if cantidad > self.saldo:
            print(f"Fondos insuficientes. Saldo disponible: ${self.saldo}")
            return
        self.saldo -= cantidad
        self.movimientos.append(f"Retiro: -${cantidad}")
        print(f"{self.titular} retiró ${cantidad}. Saldo: ${self.saldo}")

    def ver_historial(self):
        print(f"--- Historial de {self.titular} ---")
        if not self.movimientos:
            print("(sin movimientos)")
        for m in self.movimientos:
            print(m)

cuenta = CuentaBancaria("Esteban", 1000)
cuenta.depositar(500)
cuenta.retirar(200)
cuenta.retirar(5000)
cuenta.ver_historial()

```

Observa los detalles profesionales: el parámetro `saldo_inicial=0` tiene valor por defecto, así que puedes crear `CuentaBancaria("Ana")` sin saldo inicial. Cada método valida antes de actuar y usa `return` temprano para salir si algo está mal, un estilo limpio que evita

condicionales anidados. El atributo `movimientos` es una lista que vive dentro del objeto: cada cuenta lleva su propio historial, nadie lo comparte ni lo pisa. La salida del ejemplo muestra el depósito, el retiro, el rechazo del retiro imposible y el historial completo. Concepto clave: la clase no solo guarda datos, también impone reglas; el objeto se defiende solo de los malos usos.

Ejemplo 3: Estudiante que calcula su promedio

Una clase que combina listas, ciclos implícitos y decisiones: el estudiante gestiona sus calificaciones y sabe si aprueba.

```
class Estudiante:
    def __init__(self, nombre, matricula):
        self.nombre = nombre
        self.matricula = matricula
        # Cada estudiante nace con su lista vacía de calificaciones.
        self.calificaciones = []

    def agregar(self, calificacion):
        # Validamos el rango antes de guardar.
        if 0 <= calificacion <= 10:
            self.calificaciones.append(calificacion)
        else:
            print("Calificación inválida: debe estar entre 0 y 10.")

    def promedio(self):
        if not self.calificaciones:
            return 0.0
        return sum(self.calificaciones) / len(self.calificaciones)

    def __str__(self):
        return f"{self.nombre}: promedio {self.promedio():.2f}"

ana = Estudiante("Ana", "A001")
ana.agregar(9)
ana.agregar(8)
ana.agregar(15)    # Inválida: se rechaza y no entra a la lista.
ana.agregar(10)

print(ana)    # Ana: promedio 9.00
```

El constructor aquí no recibe calificaciones: cada estudiante nace con lista vacía y las va agregando con el método `agregar`, que valida el rango 0-10. El método `promedio` se protege de la división entre cero cuando la lista está vacía. Y `__str__` hace que `print(ana)` muestre un resumen bonito con el promedio a dos decimales (`: .2f`). Nota cómo `__str__` llama a `self.promedio()`: los métodos pueden usar otros métodos del mismo objeto. Para practicar: agrega un método `mayor_calificacion` que devuelva la más alta usando `max()`.

Ejemplo 4: El juego "adivina el número", versión completa y pulida

La versión del proyecto lista para ejecutar, con todo lo aprendido integrado: es el mismo diseño de la explicación, empaquetado como programa final.

```

import random

def jugar():
    secreto = random.randint(1, 100)
    intentos = 0
    print("Adivina el número del 1 al 100.")

    while True:
        entrada = input("Tu intento (o 'salir'): ")

        if entrada.lower() == "salir":
            print(f"Te rendiste. El número era {secreto}.")
            break

        try:
            numero = int(entrada)
        except ValueError:
            print("Escribe un número válido, por favor.")
            continue

        if numero < 1 or numero > 100:
            print("El número está entre 1 y 100. Intenta de nuevo.")
            continue

        intentos += 1

        if numero < secreto:
            print("Muy bajo, ve más alto.")
        elif numero > secreto:
            print("Muy alto, ve más bajo.")
        else:
            print(f"¡Exacto! Era el {secreto}.")
            print(f"Lo lograste en {intentos} intentos.")
            break

jugar()

```

Las novedades respecto a la versión de la explicación: el juego vive dentro de una función `jugar()`, porque envolver el programa en funciones lo hace reutilizable y ordenado. El jugador puede escribir "salir" para rendirse (comparamos con `entrada.lower()` para aceptar "SALIR" o "Salir"). Hay validación de rango: si el número no está entre 1 y 100, se avisa y no cuenta el intento. Y el ciclo usa `while True` con `break`, un estilo común en juegos donde hay varias

formas de salir. Cada `continue` salta lo que sigue sin contar intentos inválidos. Este es tu proyecto base: guárdalo, porque los siguientes ejemplos lo mejoran.

Ejemplo 5: El juego mejorado, con dificultad e intentos limitados

Tomamos el juego anterior y le agregamos las dos primeras mejoras: niveles de dificultad y límite de intentos. Fíjate cómo el diseño separa la configuración de la lógica.


```

import random

def elegir_dificultad():
    print("Elige dificultad: 1=Fácil (1-50, 10 intentos)")
    print("                2=Normal (1-100, 7 intentos)")
    print("                3=Difícil (1-500, 8 intentos)")
    while True:
        opcion = input("Tu elección (1/2/3): ")
        if opcion == "1":
            return 50, 10
        elif opcion == "2":
            return 100, 7
        elif opcion == "3":
            return 500, 8
        print("Opción inválida, elige 1, 2 o 3.")

def jugar():
    maximo, max_intentos = elegir_dificultad()
    secreto = random.randint(1, maximo)
    intentos = 0
    print(f"Adivina el número del 1 al {maximo}. Tienes {max_intentos} intentos.")

    while intentos < max_intentos:
        try:
            numero = int(input(f"Intento {intentos + 1}/{max_intentos}: "))
        except ValueError:
            print("Eso no es un número.")
            continue

        intentos += 1

        if numero < secreto:
            print("Muy bajo.")
        elif numero > secreto:
            print("Muy alto.")
        else:
            print(f"¡Ganaste en {intentos} intentos! El número era {secreto}.")
            return

    print(f"Perdiste. El número secreto era {secreto}.")

jugar()

```

La función `elegir_dificultad` devuelve dos valores en una tupla (`maximo`, `max_intentos`), y `jugar` los desempaqueta en una sola línea: `maximo, max_intentos = elegir_dificultad()`. El ciclo ahora tiene doble condición de salida: se gana adivinando (con `return`, que termina la función de golpe) o se pierde agotando los intentos, y el mensaje final solo aparece si el `return` nunca se ejecutó. El prompt muestra el progreso `Intento 3/7`, un detalle de experiencia de usuario que hace el juego más emocionante. Concepto clave: cuando un programa crece, dividirlo en funciones con responsabilidades claras (una configura, otra juega) evita que se vuelva un monstruo imposible de mantener.

Ejemplo 6: Mini-sistema de contactos con clases y archivo

El ejemplo más ambicioso: junta POO con archivos (Unidad 9) en un programa útil de verdad, una agenda que no olvida tus contactos aunque la cierres.

```
class Contacto:
    def __init__(self, nombre, telefono):
        self.nombre = nombre
        self.telefono = telefono

    def __str__(self):
        # Formato 'nombre,telefono': así se guarda en el archivo.
        return f"{self.nombre},{self.telefono}"

    @staticmethod
    def desde_linea(linea):
        # Reconstruye un Contacto desde una línea del archivo.
        nombre, telefono = linea.strip().split(",")
        return Contacto(nombre, telefono)

ARCHIVO = "contactos.txt"

def cargar():
    contactos = []
    try:
        with open(ARCHIVO, "r", encoding="utf-8") as f:
            for linea in f:
                if linea.strip():
                    contactos.append(Contacto.desde_linea(linea))
    except FileNotFoundError:
        # Si el archivo no existe, empezamos con lista vacía. No es error.
        pass
    return contactos
```

```

def guardar(contactos):
    with open(ARCHIVO, "w", encoding="utf-8") as f:
        for c in contactos:
            f.write(str(c) + "\n")

def main():
    contactos = cargar()
    while True:
        print("\n1=Agregar 2=Ver todos 3=Buscar 4=Salir")
        opcion = input("Elige: ")
        if opcion == "1":
            nombre = input("Nombre: ")
            telefono = input("Teléfono: ")
            contactos.append(Contacto(nombre, telefono))
            guardar(contactos)
            print("Contacto guardado.")
        elif opcion == "2":
            for c in contactos:
                print(f"- {c.nombre}: {c.telefono}")
        elif opcion == "3":
            busqueda = input("Nombre a buscar: ").lower()
            encontrados = [c for c in contactos if busqueda in c.nombre.lower()]
            for c in encontrados:
                print(f"- {c.nombre}: {c.telefono}")
            if not encontrados:
                print("No se encontró a nadie con ese nombre.")
        elif opcion == "4":
            print("¡Hasta luego!")
            break
        else:
            print("Opción inválida.")

main()

```

Hay mucho que admirar aquí. La clase `Contacto` sabe convertirse a texto (`__str__`) y reconstruirse desde texto (desde `_linea`): el objeto y el archivo hablan el mismo idioma, el formato `nombre,telefono`. El decorador `@staticmethod` marca un método que no necesita `self` porque no usa datos del objeto, solo fabrica uno nuevo desde una línea; es el primer decorador que ves y su trabajo es decirle a Python "no me pases el objeto, no lo necesito". `cargar` lee el archivo con `with open` (se cierra solo) y atrapa `FileNotFoundError` para la primera ejecución, cuando aún no hay archivo. `guardar` reescribe todo el archivo con la lista actual: simple y suficiente para una agenda pequeña. Y `main` es un menú con ciclo que usa

list comprehension para buscar. Este programa usa clases, archivos, excepciones, ciclos, condicionales, f-strings y comprehensions: literalmente todo tu curso en 70 líneas. Si entiendes este ejemplo completo, estás listo para construir lo que sea.

Práctica

Ejercicio 1

Crea una clase `Libro` con atributos `titulo`, `autor` y `paginas`. Agrega un método `descripcion` que imprima una frase como: "Cien años de soledad, de Gabriel García Márquez, tiene 471 páginas." Crea dos libros distintos e imprime su descripción.

Solución paso a paso:

```
class Libro:
    def __init__(self, titulo, autor, paginas):
        self.titulo = titulo
        self.autor = autor
        self.paginas = paginas

    def descripcion(self):
        print(f"{self.titulo}, de {self.autor}, tiene {self.paginas} páginas.")

libro1 = Libro("Cien años de soledad", "Gabriel García Márquez", 471)
libro2 = Libro("El principito", "Antoine de Saint-Exupéry", 96)

libro1.descripcion()
libro2.descripcion()
```

El constructor recibe los tres datos y los guarda con `self`. El método `descripcion` arma la frase con un f-string leyendo los atributos del objeto. Al crear dos libros, cada uno guarda sus propios datos: `libro1.descripcion()` habla del primer libro y `libro2.descripcion()` del segundo, aunque el código del método es el mismo. Concepto clave: el molde se escribe una vez, los objetos se fabrican muchas veces, cada uno con su propia información.

Ejercicio 2

Crea una clase `Rectangulo` con atributos `base` y `altura`. Agrega métodos `area` (que devuelva $\text{base} \times \text{altura}$) y `perimetro` (que devuelva $2 \times (\text{base} + \text{altura})$). Crea un rectángulo de 5×3 e imprime su área y su perímetro.

Solución paso a paso:

```
class Rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def area(self):
        return self.base * self.altura

    def perimetro(self):
        return 2 * (self.base + self.altura)

r = Rectangulo(5, 3)
print(f"Área: {r.area()}")          # Área: 15
print(f"Perímetro: {r.perimetro()}") # Perímetro: 12
```

Los métodos `area` y `perimetro` usan `return` en vez de `print`: devuelven el valor para que quien llame decida qué hacer con él (imprimirlo, guardarlo, usarlo en otro cálculo). Esta es una buena práctica importante: los métodos que calculan devuelven, los métodos que informan imprimen. Mezclarlos a lo loco te ata las manos después. Nota que `r.area()` se llama con paréntesis vacíos porque aparte de `self` (que pone Python) no necesita más datos: todo lo que necesita ya vive en los atributos.

Ejercicio 3

Extiende la clase `CuentaBancaria` del Ejemplo 2 con un método `transferir(otra_cuenta, cantidad)` que mueva dinero de una cuenta a otra. Debe validar que la cantidad sea positiva y que haya fondos suficientes. Prueba transfiriendo \$300 de la cuenta de Ana (saldo \$1000) a la de Luis (saldo \$200).

Solución paso a paso:

```

class CuentaBancaria:
    def __init__(self, titular, saldo_inicial=0):
        self.titular = titular
        self.saldo = saldo_inicial

    def depositar(self, cantidad):
        if cantidad <= 0:
            print("La cantidad debe ser positiva.")
            return
        self.saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= 0 or cantidad > self.saldo:
            print("No se puede retirar esa cantidad.")
            return False
        self.saldo -= cantidad
        return True

    def transferir(self, otra_cuenta, cantidad):
        # Reutilizamos retirar: si falla, no se hace nada más.
        if self.retirar(cantidad):
            otra_cuenta.depositar(cantidad)
            print(f"{self.titular} transfirió ${cantidad} a {otra_cuenta.titular}.")

ana = CuentaBancaria("Ana", 1000)
luis = CuentaBancaria("Luis", 200)
ana.transferir(luis, 300)
print(f"Ana: ${ana.saldo}, Luis: ${luis.saldo}") # Ana: $700, Luis: $500

```

La clave del diseño: transferir no repite la lógica de validación, reutiliza los métodos retirar y depositar que ya existen. Para eso, retirar ahora devuelve True o False según haya funcionado, y transferir solo deposita en la otra cuenta si el retiro tuvo éxito. Fíjate que un método recibe otro objeto como parámetro (otra_cuenta) y lo usa llamando sus métodos con el punto: los objetos pueden colaborar entre sí. El resultado: Ana queda con \$700 y Luis con \$500. Concepto clave: los buenos métodos se combinan como piezas de Lego; si te descubres copiando y pegando lógica, es señal de que un método debería reutilizar a otro.

Ejercicio 4

Toma el juego "adivina el número" y extiéndelo con un sistema de puntaje: el jugador empieza con 1000 puntos y pierde 10 puntos por cada intento. Si adivina, muestra su puntaje final. Si el puntaje llega a 0, el juego termina con derrota.

Solución paso a paso:

```
import random

def jugar_con_puntaje():
    secreto = random.randint(1, 100)
    puntaje = 1000
    intentos = 0
    print("Adivina el número del 1 al 100. Empiezas con 1000 puntos, cada intento cuesta 10.")

    while puntaje > 0:
        try:
            numero = int(input(f"Puntaje: {puntaje} | Tu intento: "))
        except ValueError:
            print("Escribe un número válido (no te cuesta puntos).")
            continue

        intentos += 1
        puntaje -= 10

        if numero < secreto:
            print("Muy bajo.")
        elif numero > secreto:
            print("Muy alto.")
        else:
            print(f"¡Ganaste en {intentos} intentos con {puntaje} puntos!")
            return

    print(f"Te quedaste sin puntos. El número era {secreto}.")

jugar_con_puntaje()
```

El puntaje es una variable de estado más del juego, igual que `intentos`: nace en 1000 y cada intento válido resta 10. La condición del ciclo cambió de "hasta adivinar" a "mientras queden puntos", lo que crea la segunda forma de perder. Los intentos inválidos (letras) no restan puntos gracias al `continue` antes del descuento: una decisión de diseño justa con el jugador.

Si adivina, el `return` termina con victoria y muestra el puntaje restante; si el ciclo se agota, el mensaje final revela el secreto. Concepto clave: agregar una mecánica nueva a un juego casi siempre significa agregar una variable de estado, una forma de modificarla y una condición que la use; con esa receta puedes inventar tus propias reglas.

Ejercicio 5

Crea una clase `Termometro` con un atributo `celsius`. Agrega un método `a_fahrenheit` que devuelva la conversión $((\text{celsius} \times 9/5) + 32)$ y un método `descripcion` que devuelva "frío" si es menos de 15 grados, "agradable" si está entre 15 y 28, y "caluroso" si es más de 28. Prueba con 10, 22 y 35 grados.

Solución paso a paso:

```
class Termometro:
    def __init__(self, celsius):
        self.celsius = celsius

    def a_fahrenheit(self):
        return (self.celsius * 9 / 5) + 32

    def descripcion(self):
        if self.celsius < 15:
            return "frío"
        elif self.celsius <= 28:
            return "agradable"
        else:
            return "caluroso"

for grados in [10, 22, 35]:
    t = Termometro(grados)
    print(f"{grados}°C = {t.a_fahrenheit():.1f}°F, clima {t.descripcion()}")
# 10°C = 50.0°F, clima frío.
# 22°C = 71.6°F, clima agradable.
# 35°C = 95.0°F, clima caluroso.
```

Este ejercicio practica una clase con un solo atributo y métodos que lo transforman (`a_fahrenheit`) o lo interpretan (`descripcion` con sus tres rangos). El ciclo `for` crea tres termómetros distintos con el mismo molde para probar todos los casos, incluyendo los bordes: 15 cae en "agradable" por el `<= 28` combinado con el orden de los condicionales. Verifica

mentalmente: con 15 grados, la primera condición (menor a 15) es falsa, la segunda (menor o igual a 28) es verdadera. Concepto clave: el orden de los `if/elif` define los rangos; cuando trabajes con rangos, prueba siempre los valores justo en los bordes.

Ejercicio 6

Crea una clase `ListaCompras` que use POO para gestionar una lista de compras. Debe tener: un atributo `productos` (lista que nace vacía), método `agregar(producto)`, método `quitar(producto)` que avise si el producto no estaba, método `mostrar` que numere los productos, y `__str__` que devuelva algo como "Lista con 3 productos". Pruébala agregando 3 productos, quitando uno que sí existe y uno que no.

Solución paso a paso:

```

class ListaCompras:
    def __init__(self):
        # La lista nace vacía: no necesita datos iniciales.
        self.productos = []

    def agregar(self, producto):
        self.productos.append(producto)
        print(f"'{producto}' agregado.")

    def quitar(self, producto):
        if producto in self.productos:
            self.productos.remove(producto)
            print(f"'{producto}' eliminado.")
        else:
            print(f"'{producto}' no estaba en la lista.")

    def mostrar(self):
        if not self.productos:
            print("La lista está vacía.")
            return
        print("Tu lista de compras:")
        for i, p in enumerate(self.productos, start=1):
            print(f" {i}. {p}")

    def __str__(self):
        n = len(self.productos)
        return f"Lista con {n} producto{'s' if n != 1 else ''}"

lista = ListaCompras()
lista.agregar("Tortillas")
lista.agregar("Queso")
lista.agregar("Salsa")
lista.mostrar()
lista.quitar("Queso")
lista.quitar("Refresco") # No existe: avisa sin romperse.
print(lista) # Lista con 2 productos

```

El constructor no recibe parámetros: la lista siempre nace vacía y se llena con `agregar`. El método `quitar` verifica con `in` antes de usar `remove`, porque `remove` lanza un error si el elemento no existe; esa verificación previa es manejo defensivo sin necesidad de `try/except`. En `mostrar`, `enumerate(self.productos, start=1)` numera desde 1 como lo haría una persona. Y `__str__` incluye un detalle elegante: el plural condicional `'s' if n!= 1 else ''`

para que diga "1 producto" o "3 productos" correctamente. Concepto clave: una clase bien diseñada hace imposibles los errores tontos; aquí es imposible quitar algo que no existe sin enterarte, porque el objeto mismo te avisa.

Resumen de la unidad

En esta unidad cruzaste la puerta más importante del curso: la Programación Orientada a Objetos. Aprendiste que un objeto junta datos (atributos) con acciones (métodos), igual que las cosas del mundo real tienen características y comportamientos. Entendiste que la clase es el molde y el objeto es la instancia: el plano del arquitecto contra la casa construida, el molde de galletas contra cada galleta. Dominaste la sintaxis `class`, el constructor `__init__` que configura a cada objeto al nacer, y a `self`, el pronombre con el que el objeto se refiere a sí mismo, siempre primero en los métodos y siempre puesto por Python al llamar. Viste cómo crear muchos objetos independientes con el mismo molde, cómo presentarlos bonito con `__str__`, y por qué la POO mantiene ordenados los programas grandes: cada objeto cuida sus propios datos y nadie pisa los de nadie. Construiste tu primer proyecto completo, el juego "adivina el número", desde el diseño (qué necesita, cuándo termina, qué puede salir mal) hasta el código comentado línea por línea, con validación de errores incluida, y exploraste sus mejoras: límite de intentos, niveles de dificultad y puntaje guardado en archivo conectando con la Unidad 9. Repasaste el mapa completo de las diez unidades y viste las rutas que siguen: proyectos reales con cinco ideas concretas, inteligencia artificial (numpy, pandas, scikit-learn), desarrollo web (Flask, Django) y automatización. Y te llevaste los consejos que importan: portafolio en GitHub desde hoy, práctica diaria, leer código ajeno, enseñar lo que sabes y no dejar de construir nunca.

Tu graduación

Hace diez unidades escribiste tu primer `print("Hola mundo")` y probablemente sentiste esa mezcla de emoción y de "¿y esto para qué sirve?". Hoy sabes exactamente para qué sirve: sirve para pedir datos, tomar decisiones, repetir tareas, guardar información, defenderse de errores, organizar programas con objetos y construir juegos completos. Sirve para todo lo que un programador hace.

Quiero que te quede clara una cosa: este curso te dio las herramientas, pero el programador lo hiciste tú, a punta de horas, errores y tercas ganas de que el código corriera. Nadie puede aprender a programar por ti. Tú lo hiciste. Cada vez que un error te frustró y seguiste, cada vez que rompiste algo a propósito para entenderlo, cada vez que un programa por fin funcionó y

sentiste ese corrientazo de orgullo, estabas construyendo algo más valioso que sintaxis: estabas construyendo la mentalidad de quien resuelve problemas con código. Esa mentalidad no se descarga, no se copia, no caduca. Es tuya para siempre.

Y ahora viene lo mejor, porque graduarte no es despedirte: es empezar en serio. Tienes tu juego, tienes tus clases, tienes tu portafolio por abrir y un mundo de proyectos esperando. Va a haber días en que nada funcione y quieras aventar la computadora por la ventana; en esos días acuérdate de hoy, del día en que terminaste tu primer curso completo de programación. Si pudiste con esto, puedes con lo que sigue.

Sigue construyendo. Sigue rompiendo cosas. Sigue aprendiendo. Los cursos gratis de lucasegaacademy.com te van a estar esperando cuando quieras tu siguiente reto, sea SQL, JavaScript o inteligencia artificial.

¡Felicidades, programador! Nos vemos en el siguiente curso. A seguir programando.

EXAMEN FINAL

Instrucciones

Este examen tiene 50 preguntas que cubren las 10 unidades del libro. Hay tres tipos: opción múltiple (elige la letra correcta), verdadero o falso (justifica mentalmente tu respuesta) y preguntas abiertas con código (escribe el programa completo). Hazlo sin ver las respuestas, con tu computadora a la mano para probar el código de las preguntas abiertas. No te califiques con dureza: el objetivo es detectar qué unidades necesitas repasar.

Puntaje sugerido: 90-100 excelente, 70-89 bien (repasa tus fallos), menos de 70 vuelve a leer las unidades donde fallaste y reintenta en una semana.

Preguntas de opción múltiple

PREGUNTA 1. ¿Qué muestra en pantalla este programa?

```
print("Hola" + "mundo")
```

A) Hola mundo (con espacio) B) Holamundo (sin espacio) C) Error de sintaxis D) Hola+mundo

PREGUNTA 2. ¿Cuál es el resultado de `7 // 2` en Python?

A) 3.5 B) 3 C) 4 D) 1

PREGUNTA 3. ¿Qué valor tiene `x` después de ejecutar este código?

```
x = "5"  
x = int(x) + 10
```

A) "510" B) 15 C) "15" D) Error

PREGUNTA 4. ¿Cuál de estos nombres de variable es VÁLIDO en Python?

A) 2nombre B) nombre-usuario C) nombre_usuario D) nombre usuario

PREGUNTA 5. ¿Qué muestra este código?

```
edad = int(input("Edad: "))
```

Si el usuario escribe 20: A) Guarda el texto "20" B) Guarda el número 20 C) Muestra 20 en pantalla D) Lanza un error

PREGUNTA 6. Dado `nombre = "ana"`, ¿qué muestra `print(f"Hola {nombre.upper()}")`?

A) Hola ana B) Hola ANA C) Hola {nombre.upper()} D) Error: no se puede usar métodos en f-strings

PREGUNTA 7. ¿Qué muestra este código?

```
x = 5
if x > 10:
    print("grande")
elif x > 3:
    print("mediano")
else:
    print("pequeño")
```

A) grande B) mediano C) pequeño D) No muestra nada

PREGUNTA 8. ¿Cuál expresión es True si `edad = 20` y `tiene_ine = False`?

A) `edad >= 18 and tiene_ine` B) `edad >= 18 or tiene_ine` C) `not edad >= 18` D) `edad < 18 and not tiene_ine`

PREGUNTA 9. ¿Cuántas veces se ejecuta el `print` en este ciclo?

```
for i in range(3, 10, 2):
    print(i)
```

A) 7 veces B) 4 veces C) 3 veces D) 5 veces

PREGUNTA 10. ¿Qué valor tiene `contador` al terminar?

```
contador = 0
while contador < 5:
    contador += 2
```

A) 4 B) 5 C) 6 D) Ciclo infinito

PREGUNTA 11. Dada la lista `frutas = ["manzana", "uva", "mango"]`, ¿qué muestra `frutas[-1]`?

A) manzana B) uva C) mango D) Error de índice

PREGUNTA 12. ¿Qué muestra este código?

```
numeros = [3, 1, 2]
resultado = numeros.sort()
print(resultado)
```

A) [1, 2, 3] B) [3, 1, 2] C) None D) Error

PREGUNTA 13. ¿Qué es punto = (10, 20)?

A) Una lista B) Una tupla C) Un diccionario D) Un set

PREGUNTA 14. Dado persona = {"nombre": "Ana", "edad": 25}, ¿qué muestra persona.get("ciudad", "desconocida")?

A) None B) Error: KeyError C) desconocida D) ""

PREGUNTA 15. ¿Qué muestra este código?

```
unicos = {1, 2, 2, 3, 3, 3}
print(len(unicos))
```

A) 6 B) 3 C) 4 D) Error

PREGUNTA 16. ¿Qué muestra print(" hola ".strip())?

A) "hola" B) "hola" C) "HOLA" D) "hola"(con espacio al final)

PREGUNTA 17. ¿Qué devuelve esta función al llamarla como suma(3, 4)?

```
def suma(a, b):
    return a + b
```

A) None B) 7 C) "34" D) Error: faltan los tipos

PREGUNTA 18. ¿Qué muestra este código?

```
def saludar(nombre="amigo"):
    print(f"Hola {nombre}")
saludar()
```

A) Hola B) Hola amigo C) Error: falta el argumento D) Hola None

PREGUNTA 19. ¿Qué lista produce `[x**2 for x in range(4)]`?

A) [0, 1, 4, 9] B) [1, 4, 9, 16] C) [0, 2, 4, 6] D) Error de sintaxis

PREGUNTA 20. ¿Qué muestra este código?

```
try:
    n = int("hola")
except ValueError:
    n = 0
print(n)
```

A) Error: ValueError B) 0 C) "hola" D) None

PREGUNTA 21. ¿Qué hace este código?

```
with open("datos.txt", "w") as f:
    f.write("Hola")
```

A) Lee el archivo datos.txt B) Escribe "Hola" en datos.txt (borrando lo anterior si existía) C) Agrega "Hola" al final del archivo D) Error: falta cerrar el archivo

PREGUNTA 22. ¿Cuál importación es correcta para obtener un número aleatorio entre 1 y 10?

A) `import random` y luego `random.randint(1, 10)` B) `import randint` y luego `randint(1, 10)` C) `from random import` y luego `randint(1, 10)` D) `random.import()` y luego `randint(1, 10)`

PREGUNTA 23. En POO, ¿qué es `self` dentro de un método?

A) Una palabra reservada que no hace nada B) La referencia al objeto que está usando el método C) El nombre de la clase D) Un parámetro obligatorio que siempre vale None

PREGUNTA 24. ¿Qué muestra este código?

```
class Perro:
    def __init__(self, nombre):
        self.nombre = nombre
p = Perro("Firulais")
print(p.nombre)
```

A) Error: falta self B) Firulais C) None D) Perro

PREGUNTA 25. ¿Cuál afirmación sobre `None` es correcta?

A) Es lo mismo que 0 B) Es lo mismo que `False` C) Es un valor especial que significa "sin valor" D) Es lo mismo que un string vacío

Preguntas de verdadero o falso

PREGUNTA 26. En Python, la indentación (los espacios al inicio de la línea) es obligatoria para definir bloques de código como el interior de un `if`.

PREGUNTA 27. `print(4 / 2)` muestra 2 (entero) en Python 3.

PREGUNTA 28. Los métodos de strings como `upper()` modifican el string original directamente.

PREGUNTA 29. En un `if/elif/else`, si la primera condición es verdadera, las demás ya no se evalúan.

PREGUNTA 30. `range(5)` genera los números del 1 al 5.

PREGUNTA 31. Las tuplas son mutables: se pueden modificar después de crearlas.

PREGUNTA 32. `{}` (llaves vacías) crea un diccionario vacío, no un set vacío.

PREGUNTA 33. Una función sin `return` devuelve `None` automáticamente.

PREGUNTA 34. El modo `"w"` de `open()` agrega texto al final del archivo sin borrarlo.

PREGUNTA 35. `break` termina el ciclo completo y `continue` salta a la siguiente iteración.

Preguntas abiertas con código

PREGUNTA 36. Escribe un programa que pida al usuario su nombre y dos números enteros, y muestre la suma, la resta, la multiplicación y la división de esos números en un mensaje personalizado con f-strings.

PREGUNTA 37. Escribe un programa que pida una edad y muestre: "Eres menor de edad" si es menor de 18, "Eres adulto joven" si tiene entre 18 y 30, "Eres adulto" si tiene entre 31 y 59, y "Eres adulto mayor" si tiene 60 o más. Valida que la edad no sea negativa.

PREGUNTA 38. Escribe un programa que muestre la tabla de multiplicar del número que pida el usuario (del 1 al 10), usando un ciclo `for`.

PREGUNTA 39. Escribe una función `es_par(n)` que devuelva `True` si el número es par y `False` si es impar. Luego úsala para mostrar todos los números pares del 1 al 20.

PREGUNTA 40. Escribe un programa que pida al usuario 5 calificaciones (convirtiendo a `float` y validando con `try/except` que sean números), las guarde en una lista y muestre: el promedio con 2 decimales, la calificación más alta y la más baja.

PREGUNTA 41. Dado el diccionario `alumno = {"nombre": "Luis", "materias": {"mate": 9, "historia": 8, "python": 10}}`, escribe código que muestre el nombre del alumno y su promedio de materias con 1 decimal.

PREGUNTA 42. Escribe un programa que pida una frase al usuario y cuente cuántas vocales tiene (mayúsculas y minúsculas cuentan igual). Usa un `set` o una lista de vocales.

PREGUNTA 43. Escribe una función `factorial(n)` que calcule el factorial de un número usando un ciclo (no recursión). Pruébala con 5 (debe dar 120).

PREGUNTA 44. Escribe un programa "lista de compras persistente": que cargue las compras desde `compras.txt` si el archivo existe, permita agregar productos con `input` hasta que el usuario escriba "fin", y guarde la lista final en el archivo.

PREGUNTA 45. Escribe una clase `Rectangulo` con `__init__` que reciba base y altura, y métodos `area()` y `perimetro()`. Crea un objeto de 5x3 y muestra su área y perímetro.

PREGUNTA 46. Escribe el juego "adivina el número" completo: la computadora elige un número del 1 al 50, el usuario intenta adivinarlo con pistas de "más alto" o "más bajo", cuenta los intentos y al final felicita mostrando cuántos intentos tomó. Valida que la entrada sea un número.

PREGUNTA 47. Escribe un programa que genere una contraseña aleatoria de 12 caracteres usando el módulo `random` y letras, dígitos y símbolos. Muestra la contraseña generada.

PREGUNTA 48. Escribe una función `palindromo(texto)` que devuelva `True` si el texto es un palíndromo (se lee igual al derecho y al revés, ignorando espacios y mayúsculas). Pruébala con "Anita lava la tina".

PREGUNTA 49. Escribe un programa que pida al usuario una lista de números separados por comas (ej: "5,2,8,1"), los convierta a enteros usando list comprehension, y muestre la lista ordenada, el mayor, el menor y la suma.

PREGUNTA 50. Proyecto integrador: escribe un programa "registro de estudiantes" que use un diccionario donde la clave es el nombre y el valor es una lista de 3 calificaciones. El programa debe: 1) pedir 3 estudiantes con sus 3 calificaciones cada uno (validando números), 2) guardar todo en `estudiantes.txt`, 3) leer el archivo y mostrar el promedio de cada estudiante con f-strings. Usa funciones para organizar el código.

Respuestas y soluciones

Respuestas de opción múltiple

RESPUESTA 1. B) Holamundo. El operador `+` entre strings los concatena sin agregar espacios. Si quieres el espacio, debes incluirlo: `"Ho la" + " " + "mundo"` o mejor, usa f-strings o comas en print.

RESPUESTA 2. B) 3. La doble diagonal `//` es división entera: divide y se queda con la parte entera, truncando los decimales. $7 / 2 = 3.5$, así que $7 // 2 = 3$.

RESPUESTA 3. B) 15. Primero `int("5")` convierte el texto en el número 5, luego $5 + 10 = 15$. Sin la conversión, `"5" + 10` habría lanzado `TypeError`.

RESPUESTA 4. C) `nombre_usuario`. Las reglas: no empezar con número (A inválida), no usar guiones medios (B inválida, el guion se interpreta como resta), no usar espacios (D inválida). El guion bajo sí está permitido y es la convención (`snake_case`).

RESPUESTA 5. B) Guarda el número 20. El patrón `int(input(...))` pide el dato como texto y lo convierte a entero en la misma línea. No muestra nada en pantalla por sí solo; solo guarda.

RESPUESTA 6. B) Hola ANA. Dentro de un f-string puedes poner expresiones completas entre llaves, incluyendo llamadas a métodos como `nombre.upper()`. Python lo evalúa y sustituye el resultado.

RESPUESTA 7. B) mediano. $x = 5$: la primera condición ($5 > 10$) es falsa, la segunda ($5 > 3$) es verdadera, así que ejecuta ese bloque y ya no evalúa el else.

RESPUESTA 8. B) `edad >= 18 or tiene_ine`. Con `or` basta que una sea verdadera: `20 >= 18` es `True`, así que toda la expresión es `True`. La A falla porque `tiene_ine` es `False` y `and` exige ambas. La C: `not (20 >= 18) = not True = False`. La D: `20 < 18` es `False`.

RESPUESTA 9. B) 4 veces. `range(3, 10, 2)` genera 3, 5, 7, 9 (empieza en 3, pasos de 2, sin llegar a 10). Son 4 valores, así que el `print` se ejecuta 4 veces.

RESPUESTA 10. C) 6. El ciclo suma 2 cada vez: `0 -> 2 -> 4 -> 6`. Cuando contador vale 6, la condición `6 < 5` es falsa y el ciclo termina. No es infinito porque el contador sí cambia dentro del ciclo.

RESPUESTA 11. C) `mango`. Los índices negativos cuentan desde el final: `-1` es el último elemento. Es una forma elegante de obtener el último sin saber cuántos hay.

RESPUESTA 12. C) `None`. El método `sort()` ordena la lista en su lugar (la modifica) y devuelve `None`. Si quieres la lista ordenada en una variable nueva, usa `resultado = sorted(numeros)`.

RESPUESTA 13. B) Una tupla. Los paréntesis crean tuplas, que son como listas pero inmutables: una vez creadas no se pueden modificar.

RESPUESTA 14. C) desconocida. El método `get()` devuelve el valor de la clave si existe, y si no existe devuelve el valor por defecto que le pases (aquí "desconocida") en vez de lanzar `KeyError`.

RESPUESTA 15. B) 3. Los sets eliminan duplicados automáticamente: `{1, 2, 2, 3, 3, 3}` se convierte en `{1, 2, 3}`, que tiene 3 elementos.

RESPUESTA 16. B) `"hola"`. `strip()` quita los espacios al inicio y al final del string. No toca los espacios del medio ni cambia mayúsculas.

RESPUESTA 17. B) 7. La función recibe 3 y 4 como parámetros `a` y `b`, y `return a + b` devuelve 7. Python no exige declarar tipos de parámetros.

RESPUESTA 18. B) Hola amigo. El parámetro tiene un valor por defecto (`nombre="amigo"`), así que si llamas a la función sin argumentos, usa ese valor.

RESPUESTA 19. A) `[0, 1, 4, 9]`. La comprensión eleva al cuadrado cada `x` en `range(4)`, que genera 0, 1, 2, 3. Sus cuadrados: 0, 1, 4, 9.

RESPUESTA 20. B) 0. `int("hola")` lanza `ValueError` porque "hola" no es un número, el `except ValueError` lo atrapa y asigna 0 a `n`. El programa no se rompe: muestra 0.

RESPUESTA 21. B) Escribe "Hola" en datos.txt, borrando el contenido anterior si el archivo ya existía. El modo "w" (write) crea o sobrescribe. El `with` cierra el archivo automáticamente al terminar el bloque.

RESPUESTA 22. A) `import random` y luego `random.randint(1, 10)`. Primero importas el módulo, luego usas sus funciones con la sintaxis `módulo.función`. `randint` incluye ambos extremos: puede dar del 1 al 10.

RESPUESTA 23. B) La referencia al objeto que está usando el método. Cuando llamas `perro.ladRAR()`, Python pasa automáticamente el objeto como primer argumento (`self`), para que el método sepa sobre qué objeto trabajar.

RESPUESTA 24. B) `Firulais`. Al crear `Perro("Firulais")`, Python llama a `__init__` pasando el objeto nuevo como `self` y "Firulais" como nombre; el constructor guarda `self.nombre = "Firulais"`, y luego `p.nombre` lo recupera.

RESPUESTA 25. C) Es un valor especial que significa "sin valor". No es 0, ni `False`, ni string vacío: es su propio tipo (`NoneType`) y representa ausencia de valor.

Respuestas de verdadero o falso

RESPUESTA 26. Verdadero. En Python no hay llaves para bloques: la indentación (típicamente 4 espacios) es lo que define qué líneas pertenecen al `if`, al `for`, a la función, etc. Sin indentación correcta hay `IndentationError` o el código hace otra cosa.

RESPUESTA 27. Falso. En Python 3, la división `/` siempre devuelve flotante: `4 / 2` es `2.0`. Si quieres el entero, usa `//`: `4 // 2` es `2`.

RESPUESTA 28. Falso. Los strings son inmutables: `upper()` devuelve un string NUEVO en mayúsculas y deja el original intacto. Para conservar el cambio debes asignar: `texto = texto.upper()`.

RESPUESTA 29. Verdadero. El `if/elif/else` evalúa en orden y ejecuta solo el primer bloque cuya condición sea verdadera; los demás se saltan por completo.

RESPUESTA 30. Falso. `range(5)` genera 0, 1, 2, 3, 4: empieza en cero y no incluye el 5. Es el famoso "off by one" que debes tener siempre presente.

RESPUESTA 31. Falso. Las tuplas son inmutables: una vez creadas no se pueden modificar. Esa es justo su gracia frente a las listas: sirven para datos fijos que nadie debe cambiar por accidente.

RESPUESTA 32. Verdadero. `{}` crea un diccionario vacío. Para un set vacío debes usar `set()`, porque las llaves vacías ya estaban apartadas para diccionarios.

RESPUESTA 33. Verdadero. Si una función termina sin ejecutar un `return` con valor, Python devuelve `None` automáticamente. Por eso es tan común el bug de olvidar el `return`.

RESPUESTA 34. Falso. El modo `"w"` borra todo el contenido anterior del archivo. Para agregar al final sin borrar, usa el modo `"a"` (append).

RESPUESTA 35. Verdadero. `break` rompe el ciclo por completo y sigue después de él; `continue` abandona solo la iteración actual y salta a la siguiente.

Soluciones de preguntas abiertas

RESPUESTA 36. Programa con f-strings y operaciones básicas:

```
nombre = input("¿Cómo te llamas? ").strip()
a = int(input("Primer número: "))
b = int(input("Segundo número: "))
print(f"\n{nombre}, estos son tus resultados:")
print(f"Suma: {a} + {b} = {a + b}")
print(f"Resta: {a} - {b} = {a - b}")
print(f"Multiplicación: {a} * {b} = {a * b}")
print(f"División: {a} / {b} = {a / b:.2f}")
```

Explicación: se pide el nombre (limpiando espacios) y los dos números con conversión inmediata a `int`. Cada resultado se muestra con f-string, calculando dentro de las llaves; la división se formatea con 2 decimales. Nota: si el usuario escribe 0 como segundo número, la división fallará con `ZeroDivisionError`; en un programa blindado lo validarías (Unidad 9).

RESPUESTA 37. Clasificación de edades con `elif` y validación:

```

edad = int(input("¿Cuántos años tienes? "))
if edad < 0:
    print("Esa edad no es válida.")
elif edad < 18:
    print("Eres menor de edad.")
elif edad <= 30:
    print("Eres adulto joven.")
elif edad <= 59:
    print("Eres adulto.")
else:
    print("Eres adulto mayor.")

```

Explicación: primero se descarta lo imposible (edad negativa). Luego los rangos se evalúan en orden ascendente: como cada `elif` solo se evalúa si los anteriores fallaron, no hace falta escribir `edad >= 18 and edad <= 30`; basta `edad <= 30` porque ya sabemos que es mayor o igual a 18. Este es el patrón estándar de clasificación por rangos.

RESPUESTA 38. Tabla de multiplicar con for:

```

n = int(input("¿De qué número quieres la tabla? "))
print(f"\nTabla del {n}:")
for i in range(1, 11):
    print(f"{n} x {i} = {n * i}")

```

Explicación: `range(1, 11)` genera del 1 al 10 (el 11 no se incluye). En cada iteración, `i` toma un valor y el f-string muestra la multiplicación calculada dentro de las llaves. El ciclo `for` es ideal aquí porque sabemos exactamente cuántas iteraciones queremos: 10.

RESPUESTA 39. Función `es_par` y filtrado:

```

def es_par(n):
    return n % 2 == 0

print("Números pares del 1 al 20:")
for i in range(1, 21):
    if es_par(i):
        print(i, end=" ")

```

Explicación: `n % 2 == 0` es `True` para pares y `False` para impares; la función devuelve directamente el resultado de la comparación, sin necesidad de un `if`. Luego recorremos del 1 al 20 y mostramos solo los que pasan la prueba. El `end=""` en `print` evita el salto de línea para

mostrarlos en una fila. Concepto clave: las funciones encapsulan lógica que reutilizas; el `return` devuelve un booleano que el `if` usa directo.

RESPUESTA 40. Calificaciones con validación:

```
califs = []
for i in range(5):
    while True:
        try:
            c = float(input(f"Calificación {i+1}: "))
            break
        except ValueError:
            print("Eso no es un número, intenta de nuevo.")
    califs.append(c)

promedio = sum(califs) / len(califs)
print(f"\nPromedio: {promedio:.2f}")
print(f"Más alta: {max(califs)}")
print(f"Más baja: {min(califs)}")
```

Explicación: el `while True` con `try/except` es el patrón profesional de validación: intenta convertir, y si falla muestra un mensaje amable y repite la pregunta en vez de romperse. `break` sale del `while` solo cuando la conversión tuvo éxito. Luego `sum`, `max` y `min` hacen los cálculos en una línea cada uno. Observa cómo las funciones integradas evitan ciclos manuales.

RESPUESTA 41. Promedio desde diccionario anidado:

```
alumno = {"nombre": "Luis", "materias": {"mate": 9, "historia": 8, "python": 10}}
materias = alumno["materias"]
promedio = sum(materias.values()) / len(materias)
print(f"Alumno: {alumno['nombre']}")
print(f"Promedio: {promedio:.1f}")
```

Explicación: `alumno["materias"]` accede al diccionario interno; `.values()` obtiene solo las calificaciones (9, 8, 10); `sum` entre `len` da el promedio 9.0. Dentro del f-string usamos comillas simples para la clave porque el f-string ya usa dobles. Los diccionarios anidados son exactamente así como se representan datos reales (como el JSON de internet).

RESPUESTA 42. Contador de vocales:


```

frase = input("Escribe una frase: ").lower()
vocales = {"a", "e", "i", "o", "u"}
contador = 0
for letra in frase:
    if letra in vocales:
        contador += 1
print(f"La frase tiene {contador} vocales.")

```

Explicación: convertimos todo a minúsculas para contar A y a por igual. El set de vocales permite la prueba `letra in vocales`, que es rápida y legible. Recorremos la frase letra por letra con `for` (los strings son iterables) y contamos. Alternativa elegante con `sum`: `contador = sum(1 for letra in frase if letra in vocales)`.

RESPUESTA 43. Factorial con ciclo:

```

def factorial(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado

print(factorial(5))

```

Explicación: el factorial de 5 es $5 \times 4 \times 3 \times 2 \times 1 = 120$. Empezamos con `resultado = 1` (el neutro de la multiplicación; si empezaras en 0 todo daría 0) y multiplicamos por cada número del 1 al `n`. `range(1, n + 1)` incluye a `n`. Traza mental con `n=5`: $1 \times 1 = 1$, $1 \times 2 = 2$, $2 \times 3 = 6$, $6 \times 4 = 24$, $24 \times 5 = 120$. Correcto.

RESPUESTA 44. Lista de compras persistente:

```

compras = []
try:
    with open("compras.txt", "r", encoding="utf-8") as f:
        compras = [linea.strip() for linea in f if linea.strip()]
except FileNotFoundError:
    print("(Empezando lista nueva)")

print("Escribe productos (o 'fin' para terminar):")
while True:
    p = input("> ").strip()
    if p.lower() == "fin":
        break
    if p:
        compras.append(p)

with open("compras.txt", "w", encoding="utf-8") as f:
    for item in compras:
        f.write(item + "\n")

print(f"\nLista guardada con {len(compras)} productos:")
for item in compras:
    print(f"- {item}")

```

Explicación: primero intentamos cargar la lista anterior; si el archivo no existe (FileNotFoundError), empezamos vacía sin romperse: ese es el manejo elegante de "primera vez". El while agrega productos hasta que escriben "fin" (comparando en minúsculas). Al final se sobrescribe el archivo con la lista completa (modo "w") y se muestra el resultado. La persistencia hace que la lista sobreviva al apagar el programa: tus datos ya son permanentes.

RESPUESTA 45. Clase Rectangulo:

```
class Rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def area(self):
        return self.base * self.altura

    def perimetro(self):
        return 2 * (self.base + self.altura)

r = Rectangulo(5, 3)
print(f"Área: {r.area()}")
print(f"Perímetro: {r.perimetro()}")
```

Explicación: `__init__` recibe base y altura y las guarda como atributos del objeto (`self.base`, `self.altura`). Los métodos usan esos atributos para calcular. Al crear `Rectangulo(5, 3)`, Python llama a `__init__` automáticamente. Muestra Área: 15 y Perímetro: 16. Cada objeto guarda sus propios datos: podrías crear otro rectángulo de 10×2 sin que interfiera.

RESPUESTA 46. Juego adivina el número completo:

```

import random

secreto = random.randint(1, 50)
intentos = 0
print("Adivina el número del 1 al 50.")

while True:
    entrada = input("Tu intento: ").strip()
    try:
        numero = int(entrada)
    except ValueError:
        print("Escribe un número válido.")
        continue
    intentos += 1
    if numero < secreto:
        print("Más alto...")
    elif numero > secreto:
        print("Más bajo...")
    else:
        print(f"¡Correcto! Era el {secreto}.")
        print(f"Lo lograste en {intentos} intentos.")
        break

```

Explicación: `random.randint(1, 50)` elige el secreto. El `while True` con `break` al acertar es el ciclo de juego clásico. La validación con `try/except` evita que una letra rompa el juego: `continue` salta a la siguiente iteración sin contar el intento (los intentos solo cuentan entradas válidas). Las pistas guían al jugador. Este programa combina módulos, ciclos, condicionales, validación y contadores: todo el curso en 20 líneas.

RESPUESTA 47. Generador de contraseñas:

```

import random
import string

caracteres = string.ascii_letters + string.digits + "!@#$$%&*"
password = "".join(random.choice(caracteres) for _ in range(12))
print(f"Tu contraseña: {password}")

```

Explicación: `string.ascii_letters` trae todas las letras (mayúsculas y minúsculas), `string.digits` los dígitos, y agregamos símbolos. `random.choice` elige uno al azar, lo repetimos 12 veces con un generador y `"".join` los pega sin separador. Cada ejecución da una contraseña distinta. (Para contraseñas reales de alta seguridad los profesionales usan el módulo `secrets`, pero para aprender, `random` está bien.)

RESPUESTA 48. Detector de palíndromos:

```
def palindromo(texto):
    limpio = texto.lower().replace(" ", "")
    return limpio == limpio[::-1]

print(palindromo("Anita lava la tina"))
print(palindromo("Python"))
```

Explicación: normalizamos pasando a minúsculas y quitando espacios ("anitalavalatina"), y comparamos con su reverso. El slicing `[::-1]` invierte el string: paso -1 recorre hacia atrás. La primera prueba da `True`, la segunda `False`. Los palíndromos son un ejercicio clásico de entrevistas de trabajo: ahora ya sabes resolverlo.

RESPUESTA 49. Procesar números separados por comas:

```
entrada = input("Escribe números separados por comas: ")
numeros = [int(x.strip()) for x in entrada.split(",")]
numeros.sort()
print(f"Ordenados: {numeros}")
print(f"Mayor: {max(numeros)}")
print(f"Menor: {min(numeros)}")
print(f"Suma: {sum(numeros)}")
```

Explicación: `split(",")` rompe el texto, la list comprehension convierte cada parte a `int` (limpiando espacios con `strip` por si el usuario escribió "5, 2, 8"). Luego ordenamos y mostramos estadísticas con funciones integradas. Si el usuario escribe "5,2,8,1": ordenados [1, 2, 5, 8], mayor 8, menor 1, suma 16.

RESPUESTA 50. Registro de estudiantes integrador:

```

def pedir_calificaciones(nombre):
    califs = []
    for i in range(3):
        while True:
            try:
                c = float(input(f"Calificación {i+1} de {nombre}: "))
                break
            except ValueError:
                print("Debe ser un número, intenta de nuevo.")
        califs.append(c)
    return califs

def guardar(estudiantes):
    with open("estudiantes.txt", "w", encoding="utf-8") as f:
        for nombre, califs in estudiantes.items():
            f.write(f"{nombre}:{','.join(map(str, califs))}\n")

def mostrar():
    with open("estudiantes.txt", "r", encoding="utf-8") as f:
        for linea in f:
            nombre, datos = linea.strip().split(":")
            califs = [float(x) for x in datos.split(",")]
            prom = sum(califs) / len(califs)
            print(f"{nombre}: promedio {prom:.2f}")

# Programa principal
estudiantes = {}
for i in range(3):
    nombre = input(f"\nNombre del estudiante {i+1}: ").strip()
    estudiantes[nombre] = pedir_calificaciones(nombre)

guardar(estudiantes)
print("\n--- REPORTE ---")
mostrar()

```

Explicación: este proyecto integra todo el curso. `pedir_calificaciones` es una función reutilizable con validación `try/except` (Unidades 8 y 9). El diccionario `estudiantes` mapea nombre -> lista de calificaciones (Unidad 7). `guardar` persiste en archivo con formato `nombre:9.0,8.5,10.0` usando `join` y `map` (Unidades 7 y 9). `mostrar` lee el archivo, reconstruye los datos con `split` y conversión, y calcula promedios con f-strings formateados (Unidades 3 y 9). El programa principal orquesta las tres funciones. Si lograste entender este programa completo línea por línea, felicidades: ya programas en Python de verdad. Ve por tu certificado en lucasegaacademy.com.

Despedida del examen

Si llegaste hasta aquí y resolviste el examen con honestidad, ya tienes un diagnóstico claro de tu nivel. Repasa las unidades donde fallaste, vuelve a intentarlo en unos días y cuando te sientas listo, sigue con el siguiente curso en lucasegaacademy.com. Recuerda: equivocarse en un examen es barato; equivocarse sin haber practicado es caro. ¡A seguir programando!

Colibrí Editorial EGA

Curso Completo de Python

Curso Completo de Python está organizado en 10 unidades que agrupan las clases del curso en video de Lucas EGA Academy. Cada unidad trae introducción, explicación detallada con analogías y buenas prácticas, ejemplos paso a paso, práctica con soluciones explicadas y resumen. Cubre desde la sintaxis básica hasta funciones, estructuras de datos y programación orientada a objetos, más un examen final de 50 preguntas con respuestas.

lucasegaacademy.com