

Tipos de datos y sentencias básicas en Python

En programación, comprender los **tipos de datos** disponibles y las **sentencias básicas** de control de flujo es fundamental para construir programas. En Python, un lenguaje de tipado dinámico y altamente legible, existen múltiples tipos de datos incorporados y estructuras de control que permiten dar forma a la lógica del código. A continuación, se presenta una explicación detallada de estos conceptos, incluyendo los principales tipos de datos de Python, el manejo de **variables** (y su alcance), los **operadores** disponibles (matemáticos, lógicos y de comparación), las conversiones de tipos, la creación y manipulación de **objetos**, y las sentencias básicas de control como condicionales y bucles.

Tipos de datos en Python

Python posee varios tipos de datos integrados para representar información de diversa naturaleza. Entre los tipos básicos se incluyen los numéricos, cadenas de texto, secuencias (como listas y tuplas), diccionarios, conjuntos y booleanos ¹. Cada tipo de dato tiene características y usos particulares. A continuación se describen los más importantes:

- **Números enteros** (`int`): Representan números enteros (sin parte decimal). Son utilizados para conteos, índices, identificadores, etc. Por ejemplo, `x = 5` crea una variable entera con valor 5. En Python 3 no hay distinción entre enteros normales y largos (de precisión arbitraria); el tipo `int` cubre ambos casos ².
- **Números decimales** (`float`): Son números de punto flotante, que incluyen una parte decimal. Se utilizan para representar valores con componente fraccionaria, como 3.14 o 0.001. Por ejemplo `y = 3.17` define un flotante con valor 3.17 ³. Python maneja automáticamente la precisión hasta un límite basado en la arquitectura del sistema.
- **Números complejos** (`complex`): Números con parte real y parte imaginaria. En Python se representan con la letra `j` para la parte imaginaria, por ej. `z = 4+2j` ². Son útiles en campos científicos o de cálculo avanzado que requieran números complejos.
- **Cadenas de caracteres** (`str`): Secuencias de texto (caracteres Unicode). Se definen entre comillas simples, dobles o triples (para texto multilínea). Por ejemplo: `nombre = "Ana"` es una cadena con la palabra "Ana". Las cadenas en Python son inmutables (no pueden modificarse una vez creadas) ⁴, aunque es posible operar con ellas para generar nuevas cadenas (concatenar, cortar subcadenas, buscar subtexto, etc.). Ejemplo: `mensaje = 'Hola, mundo'` crea una cadena, y `mensaje.upper()` retornaría `'HOLA, MUNDO'` (una nueva cadena en mayúsculas).
- **Listas** (`list`): Estructuras que almacenan **secuencias mutables** de elementos ordenados. Se definen con corchetes `[]`, separando los elementos por comas. Las listas pueden contener elementos de cualquier tipo y su tamaño puede crecer o disminuir dinámicamente. Son *mutables*, lo que significa que podemos modificar sus elementos una vez creada (agregar, quitar o cambiar elementos) ⁵. Ejemplo: `mi_lista = [1, 2, 3, "Hola"]` es una lista con cuatro elementos de distintos tipos. Podemos hacer `mi_lista[0] = 99` para cambiar el primer elemento, o `mi_lista.append('nuevo')` para añadir otro elemento al final.
- **Tuplas** (`tuple`): También son secuencias de elementos ordenados, **pero inmutables** (no se pueden modificar después de creadas). Se definen usualmente con paréntesis `()`. Las tuplas se usan para agrupar datos que no necesitan cambiar, ofreciendo protección frente a modificaciones accidentales. Ejemplo: `punto = (10, 20)` es una tupla de dos enteros. Si

intentáramos hacer `punto[0] = 5`, Python lanzaría un error porque las tuplas no admiten asignación a sus elementos tras la creación ⁶.

- **Diccionarios** (`dict`): Colecciones *no ordenadas* de pares clave-valor. Permiten almacenar valores etiquetados mediante una clave única (inmutable, típicamente cadenas o números) para luego acceder a ellos eficientemente mediante esa clave. Se definen con llaves `{}` usando la sintaxis `clave: valor` separando cada par con comas. Los diccionarios son mutables: se pueden agregar, modificar o eliminar pares. *Ejemplo:* `persona = {"nombre": "Ana", "edad": 30}` crea un diccionario con dos entradas, donde `"nombre"` es una clave que apunta al valor `"Ana"`, y `"edad"` es otra clave con valor `30`. Solo se puede acceder a los elementos a través de sus claves (por ej. `persona["nombre"]` devuelve `"Ana"`) ⁷ ⁸.
- **Conjuntos** (`set`): Colecciones *no ordenadas* de elementos únicos, es decir, no permiten duplicados. Se definen usando llaves `{}` con elementos separados por comas (similar a diccionarios pero sin pares clave-valor) o usando la función `set()`. Los sets son útiles para operaciones de conjuntos matemáticos (uniones, intersecciones, diferencias) y para eliminar duplicados de una colección. *Ejemplo:* `mi_set = {1, 2, 3, 3, 2}` produce un conjunto `{1, 2, 3}` donde los duplicados se eliminan automáticamente. Los conjuntos en Python son mutables (se les pueden añadir o quitar elementos), aunque existe la variante `frozenset` que es inmutable ⁹ ¹⁰.
- **Booleanos** (`bool`): Tipo de dato lógico que solo puede tomar dos valores: verdadero (`True`) o falso (`False`). Representan el resultado de evaluaciones lógicas y condiciones, y son esenciales en las sentencias de control (condicionales, bucles). En Python, `True` y `False` son en realidad constantes predefinidas de tipo `bool`, subclase de los enteros (internamente `True` vale 1 y `False` vale 0) ¹¹. Por ejemplo, `a = True` y `b = False` asignan valores booleanos. Podemos obtener valores booleanos al comparar otros datos (e.g. `5 > 3` devuelve `True`). Los booleanos permiten operaciones lógicas (`and`, `or`, `not`) que se detallarán más adelante ¹².

Nota: Además de los anteriores, Python incluye otros tipos integrados como el tipo `NoneType` (representado por el literal `None`, que indica ausencia de valor), tipos binarios (`bytes`, `bytearray`, `memoryview` para manejo de datos binarios) ¹³, y tipos de secuencia especiales como `range` (para generar secuencias de números enteros, muy usado en bucles) ¹⁴. Sin embargo, los listados arriba son los fundamentales que se encuentran con mayor frecuencia al iniciar en Python.

Variables y su uso en Python

Una **variable** es un nombre simbólico que apunta o se refiere a un valor almacenado en la memoria del programa. En Python, no es necesario declarar previamente una variable ni especificar su tipo; basta con asignarle un valor para crearla. Esto se debe a que Python es un lenguaje de **tipado dinámico**, donde el tipo de dato de la variable se determina automáticamente según el valor asignado ¹⁵. Por ejemplo, al ejecutar la instrucción `x = 5`, Python crea un objeto entero con valor 5 en memoria y establece que el nombre `x` se refiera a ese objeto (internamente, `x` es una referencia a un objeto de tipo entero) ¹⁶. Si luego hacemos `x = "Hola"`, el mismo nombre ahora referenciará un objeto de tipo cadena con ese contenido; es decir, la variable `x` habrá cambiado de tipo dinámicamente, sin necesidad de declaración explícita.

Algunos puntos importantes sobre las variables en Python:

- **Asignación:** Se realiza con el operador `=` (igual). A la izquierda va el nombre de la variable y a la derecha el valor o expresión cuyo resultado se asigna. Ejemplo: `nombre = "Ana"` asigna la

cadena "Ana" a la variable `nombre`. Si luego hacemos `edad = 30`, `edad` será un entero con valor 30.

- **Nombres de variables:** Deben comenzar con letra o guión bajo `_` (no pueden iniciar con dígito) y el resto de caracteres pueden ser letras, dígitos o guiones bajos. Además, Python distingue entre mayúsculas y minúsculas (es *case-sensitive*), por lo que `variable` y `Variable` son nombres distintos. Conviene usar nombres descriptivos y seguir convenciones (por ejemplo, usando `snake_case` para separar palabras con guiones bajos).
- **Reasignación y mutabilidad:** Podemos cambiar el valor referenciado por una variable en cualquier momento mediante asignaciones sucesivas. Si la variable apuntaba a un objeto inmutable (como una cadena o número) y se le asigna un nuevo valor, realmente **no se modifica** el objeto anterior sino que la variable pasa a referirse a otro objeto distinto en memoria. En cambio, para objetos mutables (como listas o diccionarios), es posible modificar su contenido interno sin cambiar de referencia (por ejemplo, agregar elementos a una lista existente). Esta distinción es importante para entender efectos colaterales: dos variables pueden referirse al mismo objeto mutable; en tal caso, un cambio a través de una variable se refleja al acceder por la otra, ya que es el mismo objeto en memoria.
- **Variables como referencias:** En Python, las variables funcionan como *referencias* a objetos. Si hacemos `a = 5` y luego `b = a`, **ambas** variables apuntan al *mismo* objeto entero `5` en la memoria, no se crea una copia independiente del valor. Esto significa que ciertos objetos mutables compartidos pueden afectar a varias variables. Por ejemplo, `lista1 = [1,2,3]` y luego `lista2 = lista1` hacen que `lista2` referencie la misma lista que `lista1`. Una operación como `lista2.append(4)` modificará ese objeto lista, y al inspeccionar `lista1` también veremos el `4` añadido, porque es el mismo objeto referenciado por dos nombres.

En resumen, una variable en Python es simplemente un rótulo para un objeto en memoria, que facilita almacenar y manipular datos en el código ¹⁶. La flexibilidad de no fijar el tipo de antemano le da mucha potencia al lenguaje, aunque implica que el programador debe ser consciente del tipo actual de los datos que maneja cada variable para evitar errores en tiempo de ejecución.

Ámbito (scope) de las variables

El **ámbito** de una variable se refiere a la región del programa donde dicha variable es *visible* o accesible. Python determina el alcance de las variables principalmente en dos niveles: **global** y **local** (también existe el alcance intermedio *enclosing* para funciones anidadas, y un ámbito *built-in* para nombres predefinidos del lenguaje, siguiendo el modelo LEGB: Local, Enclosing, Global, Built-in). En términos simples:

- Una **variable global** es la definida en el nivel más alto del código (fuera de cualquier función o bloque). Tiene alcance en todo el archivo o módulo, por lo que puede ser leída desde cualquier parte de ese código, incluyendo dentro de funciones (solo para lectura, a menos que se especifique lo contrario). Por ejemplo, si definimos `PI = 3.1416` fuera de cualquier función, `PI` es global.
- Una **variable local** es la que se define dentro de una función (o bloque con ámbito propio). Solo existe mientras la función se está ejecutando y es accesible únicamente dentro de esa función. Fuera de la función, esa variable no existe ni conserva valor alguno una vez la función termina. Por ejemplo, en la función `def area_circulo(radio): resultado = PI * radio**2; return resultado`, la variable `resultado` es local a `area_circulo` y no es accesible fuera de ella.

En resumen, “una variable local es aquella definida dentro de una función y solo puede ser accedida dentro de esa función. Por otro lado, una variable global es aquella definida fuera de una función y puede ser accedida desde cualquier lugar en el código” ¹⁷ .

Por defecto, cuando hacemos una asignación dentro de una función, Python trata ese nombre como una variable local a esa función (no afecta una global homónima). Como se mencionó, podemos leer variables globales dentro de funciones, pero **no modificarlas** directamente sin indicarlo: si intentamos asignar a una variable global dentro de una función, Python en realidad creará una nueva variable local con ese mismo nombre, a menos que le indiquemos explícitamente que nos referimos a la global. Para **modificar una variable global dentro de una función**, se usa la palabra clave `global`. Ejemplo:

```
x = 10      # variable global

def mi_funcion():
    global x
    x = 5    # sin 'global', esto crearía una variable local distinta
    print("Dentro de la función x =", x)

mi_funcion()
print("Fuera de la función x =", x)
```

Salida:

```
Dentro de la función x = 5
Fuera de la función x = 5
```

Con la declaración `global x`, indicamos que dentro de `mi_funcion` el nombre `x` hace referencia a la variable global ya existente, permitiendo asignarle un nuevo valor. Sin `global`, el intento de asignación habría creado un `x` local independiente, dejando intacta la `x` global.

No existe ámbito de bloque en Python: a diferencia de lenguajes como C o Java, las estructuras condicionales (`if/else`) o los bucles (`for`, `while`) **no** crean un nuevo alcance de variables. Es decir, si definimos una variable dentro de un `if` o dentro de un `for`, seguirá existiendo fuera de él. El ámbito local solo lo introducen principalmente las definiciones de funciones, clases, y módulos. Por ejemplo:

```
if True:
    y = 3
# fuera del if, y sigue existiendo
print(y) # esto imprimirá 3, ya que y es visible globalmente
```

En este código, `y` es accesible tras el bloque `if` porque **no** se limitó el ámbito en el bloque condicional. Esto es importante para no asumir encapsulamiento de variables en cada bloque indentado.

Como referencia, Python aplica la regla LEGB mencionada: primero busca un nombre en el ámbito **Local** (dentro de la función actual), luego en el **Enclosing** (si la definición está dentro de funciones

anidadas), después en el **Global** del módulo, y finalmente en el entorno **Built-in** (nombres predefinidos como funciones internas, e.g. `len`, `print`). El primer lugar donde encuentre el nombre definirá a cuál variable se refiere la expresión. Entender el alcance es crucial para evitar *bugs* donde una variable oculta a otra del mismo nombre en otro nivel, o para decidir correctamente cuándo usar variables globales (lo cual se recomienda minimizar) frente a pasar valores a funciones mediante parámetros.

Operadores en Python

Los **operadores** son símbolos que permiten realizar operaciones sobre uno o más operandos (valores o variables). Python proporciona operadores similares a otros lenguajes en las categorías aritmética, comparación, lógicos, entre otros. A continuación se resumen los principales operadores matemáticos, lógicos y de comparación, así como consideraciones sobre cómo se comportan con distintos tipos de datos.

Operadores aritméticos (matemáticos)

Sirven para realizar **operaciones numéricas básicas**: suma, resta, multiplicación, división, etc. En Python, los operadores aritméticos fundamentales son los siguientes (suponiendo `x = 10` e `y = 3` como ejemplo):

- **+** **Suma**: Devuelve la suma de dos números. `x + y` da `13` ¹⁸. (Nota: este operador también se usa para concatenar cadenas de texto, y para concatenar listas, pero entonces realiza una operación distinta a la aritmética) ¹⁹.
- **-** **Resta**: Resta el segundo operando del primero. `x - y` da `7` ¹⁸. No está definido para cadenas ni listas (no se pueden "restar" strings o listas) ²⁰.
- ***** **Multiplicación**: Producto de dos números. `x * y` da `30` ¹⁸. (También, Python permite usar `*` para repetir cadenas cierto número de veces, o para replicar estructuras como listas, aunque eso es un uso especial del operador) ²¹.
- **/** **División**: División de punto flotante. `x / y` da aproximadamente `3.3333...` (si no es exacta, se obtiene un `float`) ²². En Python 3, la división entre enteros produce un `float` si el resultado no es entero (a diferencia de Python 2 que truncaba la división entera) ²³.
- **//** **División entera** (o cociente): División truncada que devuelve la parte entera del cociente (redondeo hacia abajo para números positivos). `x // y` con `x=10`, `y=3` da `3` (ya que 10 dividido por 3 es 3 con resto 1) ²⁴. Este operador descarta la parte fraccionaria del resultado.
- **%** **Módulo**: Resto de la división entera. `x % y` da `1` (el resto de dividir 10 entre 3 es 1) ²⁴. Útil para determinar divisibilidad o extraer residuos (por ejemplo `n % 2` es 0 si `n` es par, 1 si es impar).
- ****** **Exponenciación**: Potencia. `x ** y` calcula x elevado a y. Por ejemplo `2 ** 3` resulta `8`, y `10 ** 3` da `1000` ²⁴. Este operador tiene mayor precedencia que los anteriores. (Equivale a la función `pow(x, y)`.)

El orden de precedencia de los operadores aritméticos en Python sigue las reglas matemáticas habituales: primero exponentes, luego multiplicación/división/módulo, y por último suma/resta. Todos ellos pueden combinarse con paréntesis para forzar un orden específico de evaluación.

Operadores de comparación (relacionales)

Permiten **comparar valores** y producen un resultado booleano (`True` o `False`). En Python existen los operadores de comparación estándar:

- `==` **Igual a:** Verdadero si ambos operandos son iguales. Por ejemplo, `5 == 5` es `True`, mientras que `5 == 7` es `False`. **Importante:** Este es el operador de comparación de igualdad, distinto del `=` (asignación). Comparar cadenas realiza una comparación lexicográfica carácter por carácter; comparar números de tipos diferentes (entero vs flotante) es válido si son comparables numéricamente (e.g. `5 == 5.0` es `True` porque 5 y 5.0 representan el mismo valor numérico) ²⁵.
- `!=` **Distinto de:** Verdadero si los operandos *no* son iguales. Es simplemente la negación de `==`. Por ejemplo, `5 != 7` es `True`, `5 != 5` es `False`.
- `>` **Mayor que:** Verdadero si el operando de la izquierda es mayor que el de la derecha (numéricamente o lexicográficamente dependiendo del tipo). Ej: `7 > 3` es `True`. Comparaciones de orden requieren que los tipos sean compatibles (no tiene sentido, por ejemplo, `5 > "hola"`; Python lanzará un error de tipo en ese caso).
- `<` **Menor que:** Verdadero si el operando de la izquierda es menor que el de la derecha. Ej: `3 < 7` es `True`. (Igual restricción: los tipos deben ser comparables).
- `>=` **Mayor o igual que:** Verdadero si el primero es mayor *o igual* al segundo. Ej: `5 >= 5` es `True`; `5 >= 4` es `True`.
- `<=` **Menor o igual que:** Verdadero si el primero es menor *o igual* al segundo. Ej: `5 <= 5` es `True`; `4 <= 5` es `True`.

Estos operadores *devuelven un booleano* como resultado de la comparación, y se pueden encadenar múltiples comparaciones. Python permite sintaxis como `a < b < c` para chequear que `a` es menor que `b` y `b` es menor que `c`, evaluando de manera corta y clara (equivale a `(a < b) and (b < c)`), pero `b` se evalúa solo una vez ²⁶.

Comparaciones entre distintos tipos: En Python 3, a diferencia de Python 2, no está definido un orden entre objetos de diferentes tipos arbitrarios. Esto quiere decir que salvo casos especiales (como comparar números entre sí, donde Python los convierte a un dominio común, o comparar cadenas entre sí), intentar usar `<` o `>` entre tipos no directamente comparables produce un `TypeError`. Por ejemplo, `5 < "7"` no es una comparación válida y genera un error, porque Python no sabe ordenar un entero con una cadena. Sin embargo, la **igualdad y desigualdad** (`==` y `!=`) **sí están definidas entre objetos de distintos tipos:** en la mayoría de casos, si los tipos son diferentes, Python las considera *no iguales* (por ejemplo, `5 == "5"` es `False` porque uno es `int` y otro `str`). Algunos tipos tienen reglas específicas: un entero y un flotante *sí* pueden compararse con `==` (como vimos, `5 == 5.0` es `True`), porque Python convierte internamente el `int` a `float` para compararlos numéricamente ²⁵. También, el valor booleano `True` se considera igual a `1` y `False` igual a `0` en comparaciones, dado que `bool` es una subclase de `int` en Python ¹¹.

Por otro lado, tipos como listas o tuplas se pueden comparar con `<` o `>` lexicográficamente si sus elementos se pueden comparar secuencialmente (Python va comparando elemento a elemento hasta determinar el resultado, similar a cómo se comparan palabras en un diccionario) ²⁷. Por ejemplo, `['a', 5] < ['b', 1]` es `True` porque compara 'a' con 'b' (y 'a' < 'b' al compararse por sus códigos de carácter). Pero intentar comparar una lista con un entero, o una tupla con una cadena, no está soportado y dará error de tipo.

En resumen, Python es un lenguaje **fuertemente tipado**: no hace conversiones implícitas arriesgadas entre tipos que no tengan una relación lógica definida para la operación. Debemos asegurarnos de comparar datos del mismo tipo o de tipos compatibles. Si necesitamos comparaciones especiales (por ejemplo, comparar representaciones numéricas en cadenas con números), tendremos que convertir explícitamente los tipos apropiados antes de la comparación.

Operadores lógicos (booleanos)

Los operadores lógicos permiten construir expresiones booleanas más complejas a partir de valores o comparaciones simples, combinándolos con las reglas de la lógica booleana. En Python los operadores lógicos principales son:

- **and (y lógico)**: El resultado es `True` si **ambas** expresiones u operandos son verdaderos; en caso contrario resulta `False`. Por ejemplo: `(x > 0) and (x < 10)` es `True` solo si `x` es mayor que 0 **y** además menor que 10 al mismo tiempo.
- **or (o lógico)**: El resultado es `True` si **al menos una** de las expresiones es verdadera. Solo es `False` cuando **todas** las condiciones son falsas. Ejemplo: `(x % 2 == 0) or (x % 3 == 0)` es verdadero si `x` es divisible por 2 **o** por 3 (o por ambos).
- **not (negación lógica)**: Es un operador unario (actúa sobre un solo valor) que devuelve `True` si la expresión es falsa, y `False` si la expresión es verdadera. Básicamente invierte el valor de verdad. Ej: `not (x == 0)` es `True` si `x == 0` es `False`, es decir, `not (x == 0)` comprueba que `x` **no** sea 0.

Estos operadores respetan las reglas habituales de cortocircuito: en una expresión `A and B`, Python evaluará `A` y si este resulta ser falso, *no evalúa* `B` (porque ya sabe que toda la conjunción será falsa sin necesidad de más cálculo) ²⁸. Análogamente, en `A or B`, si `A` es verdadero, Python da el resultado `True` inmediatamente sin evaluar `B` ²⁹. Esto puede ser útil para prevenir errores (por ejemplo, `(n != 0) and (m/n > 5)` no intentará calcular `m/n` si `n` es 0, evitando una división por cero) y para escribir condiciones eficientes.

Además, en Python, `and` y `or` devuelven uno de los operandos en lugar de estrictamente un booleano 0/1. Es decir, `expr1 and expr2` devolverá `expr1` si este es falso en contexto booleano, de lo contrario devuelve `expr2`. Del mismo modo, `expr1 or expr2` devuelve `expr1` si este es verdadero, sino devuelve `expr2` ³⁰ ³¹. Aunque a veces útil, esto suele transparentarse al usarlos en contextos lógicos (dentro de un `if`, Python internamente considera su valor `True/False`).

Ejemplo combinando operadores lógicos y de comparación:

```
edad = 25
ingresos = 5000
if edad >= 18 and ingresos >= 1000:
    print("Cumple los criterios")
else:
    print("No cumple los criterios")
```

En este ejemplo, la condición después del `if` usa `and` para exigir que ambas comparaciones sean verdaderas: la variable `edad` debe ser al menos 18 **y** al mismo tiempo `ingresos` debe ser al menos 1000. Solo si ambas lo son, se imprimirá "Cumple los criterios". Si cualquiera de las dos fallara (o ambas), el resultado del `and` sería `False` y ejecutaría el bloque del `else`.

Conversión implícita de tipos (coerción)

En las expresiones que involucran múltiples operandos de diferentes tipos, Python a veces realizará **conversiones implícitas** para poder llevar a cabo la operación manteniendo la consistencia en el tipo de resultado. Estas conversiones automáticas suelen ocurrir en contextos aritméticos con tipos numéricos y siguen una jerarquía conocida como *promoción de tipos*:

1. Si un operando es de tipo **booleano** (`bool`) y otro es numérico (`int`, `float` o `complex`), Python convierte el booleano a entero (recordando que `True` equivale a 1 y `False` equivale a 0) ¹¹ antes de proceder. Ejemplo: `True + 2` se interpreta como `1 + 2`, dando `3`.
2. Si hay mezcla de **enteros** (`int`) y **flotantes** (`float`) en una expresión, los enteros se convierten a flotantes para realizar la operación en coma flotante (evitando pérdida de la parte decimal). Ejemplo: `3 + 2.5` convierte el 3 en 3.0 implícitamente y el resultado será 5.5 (tipo `float`).
3. Si se mezclan **floats** y **complejos** (`complex`), el float se promociona a complejo (imaginaria 0) para la operación. Por ejemplo, `2.0 + (3+4j)` se convierte en `(2.0+0j) + (3+4j)` resultando `5+4j`.

En resumen, la jerarquía típica de conversión implícita en Python es: `bool → int → float → complex` ³². Python convierte al "tipo más amplio" de la jerarquía presente en la operación, para asegurar que no se pierda información (por ejemplo, no convertiría `float` a `int` automáticamente porque podría perder la parte decimal, pero sí convierte `int` a `float` porque un entero puede representarse exactamente como `float` hasta cierto rango) ³³.

Ejemplos de conversiones implícitas:

- Expresión `5 + True`: aquí `True` (`bool`) se convierte a `1` (`int`), resultando `5 + 1 = 6`. El tipo del resultado será entero (`int`).
- Expresión `4 * 3.5`: el 4 (`int`) se convierte a 4.0 (`float`) y realiza la multiplicación en coma flotante, dando 14.0 (`float`).
- Expresión `7.0 - False`: `False` se convierte a 0 (`int`, luego a `float` 0.0) y el resultado es `7.0 - 0.0 = 7.0` (`float`).
- Expresión `2 + 3j + 4`: el 4 (`int`) se convierte primero a 4.0 (`float`) y luego a `4.0 + 0j` (`complex`) para sumarlo con `2+3j`, obteniendo `(6+3j)`.

Es importante destacar que Python **no hace conversiones implícitas entre tipos no numéricos de forma mágica**. Por ejemplo, no convierte automáticamente una cadena que contiene dígitos en un número en contexto aritmético. Intentar `"123" + 5` produce un error de tipo, ya que Python no sabe sumar directamente una cadena y un entero (no asume que quisiste convertir la cadena `"123"` al número 123; debes hacerlo explícitamente con `int("123") + 5` si ese es el objetivo). En operaciones mixtas que involucren secuencias (listas, tuplas, cadenas) y números, no hay coerción automática: esos tipos son incompatibles para aritmética a menos que se transformen manualmente.

En cuanto a comparaciones, como mencionamos antes, Python no convierte automáticamente tipos distintos para comparaciones de orden. Prefiere lanzar un error que dar un resultado potencialmente ambiguo. La excepción es de nuevo con números: Python considera `int` y `float` (y `bool`) comparables en valor mediante conversión implícita al tipo más general (`float`), y también permite comparar `int` con `complex` en igualdad (convirtiendo `int` a `complex`) sabiendo que la parte imaginaria del `int` convertido será 0. Fuera de eso, es responsabilidad del programador convertir datos para compararlos si es necesario (por ejemplo, comparar la longitud de una lista con un número, etc.,

no requiere conversión porque `len(lista)` ya produce un número; pero comparar un string con un número requiere convertir uno de los dos lados, según la lógica deseada).

En situaciones donde necesitemos forzar un cambio de tipo, Python ofrece funciones integradas para **conversión explícita** (casting): `int()`, `float()`, `str()`, `bool()`, etc., que toman un valor y devuelven una representación en el tipo deseado si es posible. Por ejemplo, `str(123)` produce `"123"`, `int("45")` produce el número `45` (si la cadena es un número válido), `float("3.14")` da `3.14`. También existen `list()` para convertir iterables a listas, `tuple()` para tuplas, etc. Estas conversiones explícitas son necesarias cuando Python no convierte por sí solo en el contexto dado.

Crear y manipular objetos en Python

Python es un lenguaje de programación orientado a objetos, lo que significa que bajo el capó *todo* en Python es un **objeto** (incluyendo los tipos básicos). Aquí nos enfocaremos en cómo definir nuestros propios tipos de objetos mediante **clases** y cómo instanciarlos (crearlos) y manipular sus **atributos**.

Una **clase** en Python es como un plano o plantilla que define la estructura y comportamiento (datos y funciones) que tendrán los objetos creados a partir de ella. Un **objeto** es una instancia concreta de una clase, con valores específicos en sus atributos. La programación orientada a objetos permite agrupar datos y funcionalidades relacionadas.

Crear un objeto (instancia) de una clase

Para crear nuestros propios objetos en Python, primero definimos una clase usando la sintaxis `class NombreDeClase:`, especificando dentro los atributos (normalmente mediante un método constructor `__init__`) y métodos que tendrán sus instancias. Luego, obtenemos un objeto *instanciando* esa clase, es decir, llamando al nombre de la clase como si fuera una función, lo que ejecuta el constructor y produce un nuevo objeto en memoria.

En resumen, “para crear un objeto Python, defines una clase y luego instancias la clase usando el constructor de la clase. Luego puedes llamar a métodos o acceder a atributos del objeto” ³⁴. Veamos un ejemplo simple de definición de clase e instanciación de un objeto:

```
class Persona:
    def __init__(self, nombre, edad):
        # método constructor, llamado al crear una nueva instancia
        self.nombre = nombre    # atributo de instancia 'nombre'
        self.edad = edad        # atributo de instancia 'edad'

    def saludar(self):
        # método de instancia que imprime un saludo usando los atributos
        print(f"Hola, me llamo {self.nombre} y tengo {self.edad} años.")

# Instanciar la clase Persona para crear objetos (instancias)
p1 = Persona("Ana", 30)
p2 = Persona("Luis", 22)
```

En este ejemplo, hemos definido la clase `Persona` con un constructor (`__init__`) que recibe nombre y edad, asignándolos a los atributos del objeto (`self.nombre` y `self.edad`). También definimos un método `saludar` que utiliza esos atributos.

Al hacer `p1 = Persona("Ana", 30)`, Python crea un **objeto** de tipo `Persona`. Durante este proceso, llama automáticamente al método `__init__` pasando `"Ana"` y `30` como argumentos, de modo que dentro de `__init__`, `self.nombre` se establece a `"Ana"` y `self.edad` a `30` para ese objeto. El resultado es que `p1` ahora referencia a un objeto `Persona` en memoria con esos atributos. De forma similar, `p2 = Persona("Luis", 22)` crea otro objeto `Persona` con nombre "Luis" y edad 22. Cada instancia tiene sus propios valores de atributos independientes.

Atributos de un objeto y cómo manipularlos

Los **atributos** de un objeto son básicamente variables que pertenecen a ese objeto (definidas normalmente vía `self` dentro de la clase). En el ejemplo anterior, `nombre` y `edad` son atributos de los objetos de la clase `Persona`. Podemos acceder y modificar estos atributos utilizando la notación de punto `objeto.atributo`.

Siguiendo con el ejemplo:

```
print(p1.nombre)    # Acceder al atributo 'nombre' del objeto p1 -> imprime
                    "Ana"
p2.edad = 23        # Modificar el atributo 'edad' de p2 (antes era 22, ahora
                    23)
p1.saludar()        # Llamar al método saludar() del objeto p1 -> "Hola, me
                    llamo Ana y tengo 30 años."
```

La primera línea imprime el nombre almacenado en `p1` (que es "Ana"). La segunda línea cambia la edad de `p2` a 23 (sobrescribiendo el valor anterior 22). La tercera línea demuestra cómo invocar un método de instancia: `p1.saludar()` hace que Python busque el método `saludar` dentro de la clase `Persona` y lo ejecute con `self` apuntando a `p1`, por lo que imprime un mensaje usando los datos de `p1`.

Manipular atributos puede significar tanto leer su valor como asignarles nuevos valores. También es posible crear atributos "sobre la marcha" en un objeto (Python lo permite dinámicamente), aunque no es siempre buena práctica hacerlo fuera del `__init__` ya que puede complicar la consistencia del objeto.

Cabe destacar que los atributos de instancia (como `self.nombre`) son distintos para cada objeto. Cambiar `p2.edad` no afecta a `p1.edad` y viceversa, porque cada instancia tiene su propio espacio de atributos. Sin embargo, Python también soporta **atributos de clase** (variables declaradas dentro del cuerpo de la clase pero fuera de los métodos) que son compartidos por todas las instancias de esa clase.

Por ejemplo:

```

class Punto:
    color_defecto = "rojo" # atributo de clase (común a todos)
    def __init__(self, x, y):
        self.x = x         # atributos de instancia
        self.y = y

p = Punto(1, 2)
q = Punto(3, 4)
print(p.color_defecto) # "rojo"
print(q.color_defecto) # "rojo"
Punto.color_defecto = "azul" # modificar el atributo de clase
print(p.color_defecto) # "azul" (afecta a ambas instancias)

```

Aquí `color_defecto` es un atributo asociado a la clase `Punto` en sí misma, no solo a cada objeto. Por eso al cambiarlo a "azul" en la clase, todas las instancias lo ven actualizado. En cambio `x` e `y` son atributos propios de cada objeto `Punto`.

En la mayoría de casos al empezar, nos enfocamos en atributos de instancia, manipulándolos mediante la notación `obj.atributo`. Python también proporciona funciones útiles como `getattr(obj, "nombre_atributo")` para obtener un atributo por nombre en forma de cadena, `hasattr(obj, "atrib")` para comprobar si el objeto tiene un cierto atributo, y `setattr(obj, "atrib", valor)` para asignar un atributo dinámicamente por nombre. Estas pueden ser útiles en metaprogramación, pero para la mayoría de usos normales, la notación de punto es suficiente y más clara.

Resumiendo, crear objetos en Python implica definir clases y usar sus constructores, y la manipulación de atributos se realiza accediendo con `obj.atributo` para obtener o establecer valores. Esto permite modelar entidades del mundo real o abstracciones lógicas, agrupando en un objeto tanto sus datos (atributos) como funciones asociadas (métodos). La orientación a objetos hace más fácil organizar programas grandes, evitar repetición de código y **encapsular** lógica relacionada.

Sentencias básicas de control de flujo

Finalmente, en la construcción de programas es esencial el uso de **sentencias de control** que permiten alterar el flujo secuencial normal del código. Las más importantes son las sentencias **condicionales** (que ejecutan código solo si se cumple una condición) y los **bucles** o ciclos (que repiten un bloque de código varias veces, ya sea iterando sobre una colección o mientras se cumpla una condición). Veremos a continuación las estructuras básicas: `if/elif/else`, `for`, `while`, así como las sentencias de control de bucle `break` y `continue` que permiten modificar la ejecución de los loops.

La instrucción `if` (condicional)

La sentencia compuesta `if` permite **ejecutar condicionalmente** un bloque de código en función de una expresión booleana. En castellano equivale a "si (condición) entonces...". La sintaxis básica es:

```

if condición:
    # bloque 1 (se ejecuta si la condición es True)
elif otra_condicion:
    # bloque 2 (se ejecuta si la primera no se cumplió, pero esta sí)

```

```
else:
    # bloque 3 (se ejecuta si ninguna de las condiciones anteriores fue True)
```

Solo se ejecuta uno de los bloques posibles: el primero cuyo condicional sea verdadero; si ninguno lo es, opcionalmente el bloque `else` final se ejecutará como caso por defecto ³⁵. Los bloques se determinan por la indentación (espacios o tabulaciones sangrando el código). Por convención se usan 4 espacios para la indentación en Python.

Ejemplo de uso del `if` con `elif` y `else`:

```
x = 0
if x > 0:
    print("x es positivo")
elif x == 0:
    print("x es cero")
else:
    print("x es negativo")
```

En este caso, el programa imprimirá `"x es cero"` porque la primera condición `x > 0` es falsa, la segunda condición `x == 0` es verdadera, entonces ejecuta ese bloque. El bloque `else` final solo se ejecutaría si todas las condiciones anteriores fueran falsas (es decir, si `x` no fuera `>0` ni `==0`, implicando que es negativo en este contexto).

Se pueden encadenar múltiples cláusulas `elif` (abreviatura de *else if*) para chequear varias condiciones en orden. La primera que resulte verdadera gana, y las demás se saltan. Si ninguna se cumple, va al `else` (si está presente).

Por ejemplo:

```
def clasificar_nota(nota):
    if nota < 0 or nota > 10:
        print("Nota inválida")
    elif nota >= 9:
        print("Sobresaliente")
    elif nota >= 7:
        print("Notable")
    elif nota >= 5:
        print("Aprobado")
    else:
        print("Suspenso")

clasificar_nota(8)    # Imprime "Notable"
clasificar_nota(3)    # Imprime "Suspenso"
```

En este ejemplo, según el valor de `nota` se elige un camino u otro. Obsérvese que las condiciones están ordenadas de más restrictiva a más amplia; en cuanto una se cumple, Python ejecuta ese bloque y luego salta al final del `if` (no evalúa las siguientes condiciones aunque también pudieran ser verdaderas, ya que la estructura `if/elif` es *exclusiva*).

Una cosa a notar es que Python no usa llaves `{}` ni palabras explícitas para delimitar bloques; la indentación es lo que define qué sentencias pertenecen a cada bloque del `if`. Todas las líneas sangradas bajo un `if` (hasta volver a la columna anterior) forman parte del bloque condicional. Lo mismo aplica a los bloques `elif` y `else`.

El bucle `for`

La sentencia `for` en Python se usa para **iterar** sobre una secuencia de elementos, ejecutando el bloque interno una vez por cada elemento de la colección. Su sintaxis es:

```
for variable in secuencia:
    # bloque de código a repetir
```

Donde *secuencia* puede ser cualquier objeto **iterable**: típicamente una lista, tupla, cadena de caracteres, rango de números, etc. En cada iteración, la *variable* toma el siguiente valor de la secuencia, y luego se ejecuta el bloque. Cuando se agotan los elementos, el bucle termina automáticamente.

Ejemplos básicos:

```
# Ejemplo 1: iterando sobre una lista
frutas = ["manzana", "banano", "cereza"]
for fruta in frutas:
    print("Me gusta la", fruta)
# Esto imprimirá:
# Me gusta la manzana
# Me gusta la banano
# Me gusta la cereza

# Ejemplo 2: usando range() para iterar 5 veces (0 a 4)
for i in range(5):
    print(i)
# Imprime: 0, 1, 2, 3, 4 (cada uno en una línea separada)
```

En el primer ejemplo, el bucle se repite 3 veces (tantas como elementos hay en la lista `frutas`). En cada iteración, la variable `fruta` toma un valor: "manzana" en la primera, "banano" en la segunda, "cereza" en la tercera, y se ejecuta el `print` con ese valor.

En el segundo ejemplo, `range(5)` genera la secuencia de enteros `[0, 1, 2, 3, 4]`. Así que el `for` asigna a `i` esos valores secuencialmente, imprimiéndolos. El uso de `range(n)` es muy común para iterar *n* veces, o para iterar por índices sobre una secuencia.

Python permite también iterar sobre estructuras más complejas (como diccionarios, conjuntos, archivos línea a línea, etc.), pero en esos casos la semántica exacta depende del iterador que el objeto proporciona. Por ejemplo, iterar sobre un diccionario recorre sus claves por defecto.

Una característica poderosa en Python es la posibilidad de **desempaquetar** valores en el `for`. Si la secuencia contiene elementos que son tuplas de varios valores, podemos escribir algo como:

```
puntos = [(1,2), (3,4), (5,6)]
for (x, y) in puntos:
    print(f"Coordenadas: x={x}, y={y}")
```

Esto imprimirá cada par de coordenadas, ya que en cada iteración Python desempaca la tupla en las variables `x` e `y`.

En resumen, el `for` de Python es equivalente al "for-each" de otros lenguajes: recorre directamente elementos de una colección en lugar de basarse en un contador (aunque internamente podemos usar un contador con `range` como vimos). Si se necesita acceso al índice del elemento durante la iteración, se suele usar la función `enumerate(secuencia)`, que produce pares (índice, valor).

El bucle `while`

El `while` es una estructura de repetición **basada en una condición lógica**. Funciona de la siguiente manera: *"mientras la condición sea verdadera, sigue ejecutando el bloque de código"*. Su sintaxis es:

```
while condición:
    # bloque a ejecutar repetidamente
    # (debería haber algo dentro que eventualmente haga falsa la condición,
    para no tener bucle infinito)
```

Antes de cada iteración, Python evalúa la condición; si es `True`, ejecuta el bloque y luego regresa a comprobar la condición, así sucesivamente hasta que la condición resulte `False`. Cuando esto ocurre, el bucle termina y la ejecución continúa con la siguiente instrucción después del `while`.

Ejemplo de un bucle `while` simple:

```
contador = 5
while contador > 0:
    print("Contando:", contador)
    contador -= 1    # reducimos el contador en 1 en cada iteración
print("Despegue!")
```

La salida será:

```
Contando: 5
Contando: 4
Contando: 3
Contando: 2
Contando: 1
Despegue!
```

Aquí la variable `contador` se inicializa en 5. La condición del `while` comprueba `contador > 0`. Mientras eso sea verdadero, imprime el valor y luego lo decrementa en 1. Cuando `contador` llega a 0,

la condición `contador > 0` pasa a ser falsa y el bucle termina, continuando con la impresión de "Despegue!".

Es **crucial** asegurarse de que la condición del `while` *eventualmente se vuelva falsa*, de lo contrario tendremos un **bucle infinito** (que nunca termina por sí solo). Por ejemplo, si olvidáramos actualizar `contador` dentro del bucle, la condición `contador > 0` sería siempre verdadera para el mismo valor inicial y el bucle no terminaría. Si accidentalmente se crea un bucle infinito, habrá que interrumpir manualmente la ejecución del programa (en muchos entornos, con Ctrl+C). Por tanto, al usar `while`, garantice que dentro del bloque se modifica alguna variable o estado involucrado en la condición, o la condición depende de algo externo que cambiará, para que eventualmente se corte la repetición.

El `while` es útil cuando no sabemos de antemano cuántas iteraciones se necesitarán, sino que depende de alguna condición dinámica. Por ejemplo, leer entradas de usuario hasta que ingrese cierto valor, esperar hasta que un recurso esté disponible, etc.

Un ejemplo más práctico: supongamos que queremos sumar números ingresados por teclado hasta que el usuario escriba "fin":

```
total = 0
print("Ingrese números para sumar. Escriba 'fin' para terminar.")
while True: # bucle "infinito" controlado internamente
    entrada = input("> ")
    if entrada.lower() == "fin":
        break # salimos del bucle
    try:
        numero = float(entrada)
    except ValueError:
        print("Entrada no válida, ingrese un número o 'fin'.")
        continue # pedir de nuevo sin añadir nada
    total += numero

print("La suma total es:", total)
```

En este código, usamos `while True` para crear un loop que en principio es infinito, pero dentro usamos una condición para romperlo (`break`) cuando el usuario ingresa "fin". Si la entrada no es "fin", intentamos convertirla a número; si falla (la cadena no representaba un número), usamos `continue` para saltar a la siguiente iteración pidiendo otra entrada, evitando ejecutar las líneas posteriores en esa iteración. De lo contrario, sumamos el número al total y el bucle se repite. Este ejemplo combina varios aspectos: bucle `while`, entrada de usuario, manejo de excepciones, y las sentencias `break` y `continue`.

Las sentencias `break` y `continue`

Estas son sentencias de control de bucles que permiten alterar el flujo normal de iteración desde dentro de un `for` o `while`. En particular:

- `break`: Termina *inmediatamente* el bucle que lo contiene. El flujo de ejecución sale del bucle por completo, incluso si quedaban iteraciones pendientes. (No rompe bucles externos anidados, solo el más interno en el que aparece).

- `continue`: Salta lo que resta de la iteración actual y pasa a la *siguiente iteración* del bucle. El bucle no se termina, pero cualquier código que quede después de `continue` en el cuerpo del loop se omite en esta vuelta, y se vuelve a evaluar la condición del `while` o a avanzar al siguiente elemento del `for`.

Usamos `break` cuando, bajo cierta condición, ya no tiene sentido continuar repitiendo el loop y queremos salir de él. Por ejemplo, buscar un elemento en una lista: en cuanto lo encontramos, podemos `break` para no seguir buscando. `continue` se usa cuando en algunas circunstancias particulares de la iteración actual queremos **saltarnos** el procesamiento restante y continuar con la siguiente repetición (p.ej., omitir ciertos valores).

Como mencionamos en el ejemplo del `while` anterior, es común ver `break` dentro de un `if` dentro del bucle, para decidir romper en base a algo, y similar con `continue`.

Ejemplo de uso de `break` en un `for`:

```
for n in range(1, 11):
    print("Revisando número", n)
    if n == 5:
        print("Número 5 encontrado, deteniendo búsqueda.")
        break
print("Bucle finalizado")
```

Salida:

```
Revisando número 1
Revisando número 2
Revisando número 3
Revisando número 4
Revisando número 5
Número 5 encontrado, deteniendo búsqueda.
Bucle finalizado
```

Aquí el bucle `for` iba a iterar de 1 a 10, pero al llegar a `n == 5` cumplimos la condición y ejecutamos el `break`, que rompe el bucle inmediatamente. Así, las iteraciones para 6,7,8,9,10 nunca ocurren. Tras el `break`, el flujo continúa *después* del `for` imprimiendo "Bucle finalizado". (Nótese que si hubiera otro bucle externo englobando este, ese externo no se rompería, solo salimos del bucle actual).

Ejemplo de uso de `continue`:

```
for i in range(1, 6):
    if i == 3:
        continue
    print("Número:", i)
```

Salida:

```
Número: 1
Número: 2
Número: 4
Número: 5
```

Aquí, cuando `i` vale 3, la sentencia `continue` se ejecuta, lo que hace que el bucle *salte* la impresión para ese valor y vaya a la siguiente iteración (con `i=4`). Por eso en la salida falta la línea "Número: 3". Las demás iteraciones se ejecutan normalmente.

En resumen, `break` y `continue` son herramientas para controlar la ejecución dentro de los loops: "`break` sale del ciclo por completo; `continue` salta al siguiente ciclo (iteración) inmediatamente" ³⁶ ³⁷. Usándolas con prudencia, se pueden hacer más eficientes ciertos algoritmos (evitar iteraciones innecesarias) o manejar condiciones especiales dentro de un bucle.

Con lo anterior, hemos cubierto los fundamentos solicitados: los tipos de datos básicos de Python y su uso, el concepto de variables y alcance, operadores matemáticos, lógicos y de comparación (incluyendo cómo Python maneja diferentes tipos en las expresiones), cómo crear y manipular objetos (orientación a objetos básica), y las sentencias de control esenciales (`if`, `for`, `while`, junto con `break` y `continue`). Esta base proporciona las herramientas necesarias para construir programas sencillos pero completos en Python, estructurando tanto los datos como la lógica de flujo de manera clara y efectiva. Con práctica, estos conceptos se afianzarán y permitirán abordar problemas de programación más complejos aprovechando las capacidades del lenguaje.

Referencias utilizadas:

- Documentación oficial de Python (en español) – *Tipos de datos integrados* ¹ ³ ³⁸ ⁵ ⁶ ⁷ ⁸ ³⁹ ⁴⁰, *Operaciones booleanas y comparaciones* ⁴¹ ¹¹, *Sentencias compuestas (if, while, for)* ³⁵.
- Tutoriales y artículos en español sobre Python – descripción de tipos de datos básicos ⁴ ¹⁰ ⁴², variables y tipado dinámico ¹⁵, alcance de variables globales y locales ¹⁷, operadores aritméticos y relacionales ¹⁸ ²⁵, comportamiento de `bool` como entero ¹¹, conversión implícita de tipos numéricos ³², y uso de `break/continue` en bucles ³⁶ ³⁷.

¹ ² ³ ⁴ ⁵ ⁶ ⁷ ⁸ ⁹ ¹⁰ ¹² ¹³ ¹⁴ ³⁸ ³⁹ ⁴⁰ ⁴² Python data types: numéricos y mucho más - IONOS España

<https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/python-data-types/>

¹¹ ²⁵ ²⁷ Operadores Relacionales | El Libro De Python

<https://ellibrodepython.com/operadores-relacionales>

¹⁵ ¹⁶ Variables de Python: guía para principiantes, para entenderlo todo

<https://datascientest.com/es/variables-de-python>

¹⁷ Alcance de variables en Python: global y local

<https://apuntes.de/python/alcance-de-variables-en-python-global-y-local/>

¹⁸ ¹⁹ ²⁰ ²¹ ²² ²³ ²⁴ Operadores Aritméticos | El Libro De Python

<https://ellibrodepython.com/operadores-aritmeticos>

26 28 29 30 31 41 Tipos integrados — documentación de Python - 3.15.0a2

<https://docs.python.org/es/dev/library/stdtypes.html>

32 33 Type Conversion. We know that each data value has a... | by code guest | Python Journey | Medium

<https://medium.com/python-journey/type-conversion-e280e2a43c5b>

34 OOP en Python: Cómo crear una clase, heredar propiedades y métodos

<https://diveintopython.org/es/learn/classes>

35 8. Sentencias compuestas — documentación de Python - 3.13.9

https://docs.python.org/es/3.13/reference/compound_stmts.html

36 Python break and continue (With Examples)

<https://www.programiz.com/python-programming/break-continue>

37 Sentencias IF y los bucles WHILE y FOR en Python - ▷ Cursos de Programación de 0 a Experto © Garantizados

<https://unipython.com/sentencias-if-los-bucles-while-for-python/>