Docente

Juan Ignacio Bonini

🎯 Objetivo de la guía

El propósito principal de esta guía es brindar una **introducción completa, detallada y progresiva** al uso de **Python**, abordando conceptos clave que van desde lo más básico hasta técnicas avanzadas. La guía está estructurada para facilitar el aprendizaje y desarrollo de habilidades prácticas a través de ejemplos claros, y explicaciones técnicas.

📝 ¿A quién está dirigida esta guía?

Esta guía está diseñada para:

- ✅ Estudiantes de programación que desean profundizar sus conocimientos en Python.
- ✅ Desarrolladores que buscan mejorar su comprensión de funciones clave, manejo de errores y formateo de datos.
- ✅ Profesionales que necesitan automatizar tareas repetitivas o generar informes automatizados.
- ✅ Personas interesadas en manipular datos, generar informes personalizados y optimizar procesos mediante scripts en Python.

🐍 Guía completa de python

✅ **Objetivo:** Esta guía está diseñada para programadores con conocimientos básicos que desean profundizar en **Python**. Aprenderás desde operaciones simples hasta manipulación avanzada de listas, manejo de errores, operadores bitwise y mucho más.

---

1. Instalación y entorno

Verificar instalación

```
$ python3 --version
```

Asegúrate de tener **Python 3.12+**.

Si no está instalado, descárgalo desde python.org.

Abrir el intérprete de Python

```
$ python3
```

Verás algo como:

```
Python 3.12.3 (main, Feb  4 2025, 14:48:35)
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

---

2. Primeros pasos

Imprimir texto

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

`print()` imprime el texto pasado como argumento.
Retorna `None`, ya que su propósito es mostrar información en la consola.

Asignación de variables

```
>>> a = 5
>>> b = "Hola"
>>> c = 3.14
```

Asignamos valores a las variables `a`, `b` y `c` usando el operador `=`.

Importante: La expresión a la **derecha** del `=` se **evalúa primero** y su resultado se almacena en la variable a la **izquierda**.

Ejemplo:

```
>>> x = 2 + 3

>>> x
5
```

En este caso:
Se evalúa `2 + 3` dando como resultado `5`.
El valor `5` se asigna a la variable `x`.

Nota: Si la parte derecha del `=` es una operación compleja, se resuelve antes de la asignación:

```
>>> y = (3 * 5) + (4 / 2)
>>> y
17.0
```

Primero se evalúan `3 * 5` y `4 / 2`, y después se suman para dar `17.0`.

Asignación múltiple

```
>>> x, y, z = 1, "mundo", 3.5
>>> x
1
>>> y
'mundo'
>>> z
3.5
```

Puedes asignar valores a varias variables en una sola línea.

Intercambio de variables

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Intercambia los valores de `a` y `b` sin necesidad de una variable temporal.

Desempaquetado de valores (unpacking)

```
>>> valores = (4, 5, 6)
>>> x, y, z = valores
>>> x
4
>>> y
5
>>> z
6
```

Unpacking: Extrae valores de una lista o tupla y los asigna a variables.
Si hay más valores de los que esperas:

```
>>> a, b = [10, 20, 30]
Traceback (most recent call last):
ValueError: too many values to unpack (expected 2)
```

Desempaquetado con * para valores restantes

```
>>> a, *resto, b = [1, 2, 3, 4, 5]
>>> a
1
>>> resto
[2, 3, 4]
>>> b
5
```

Usa * para capturar los valores intermedios o restantes durante el desempaquetado.

Asignación con el mismo valor

```
>>> x = y = z = 0
>>> x
0
>>> y
0
>>> z
0
```

Asigna el mismo valor a múltiples variables.

Identidad de objetos con `id()`

```
>>> a = 5
>>> b = a
>>> id(a) == id(b)

True
```

`id()` muestra la ubicación en memoria del objeto.

Si asignas una variable a otra (`b = a`), ambas apuntan al **mismo objeto**.

Cuidado con objetos mutables

```
>>> lista1 = [1, 2, 3]
>>> lista2 = lista1
>>> lista2.append(4)
>>> lista1
[1, 2, 3, 4]
```

Si asignas una lista a otra variable, ambas apuntan al **mismo objeto**, lo que puede causar resultados inesperados.

Copiar listas para evitar problemas

```
>>> lista1 = [1, 2, 3]
>>> lista2 = lista1[:]  # Copia superficial
>>> lista2.append(4)
>>> lista1
[1, 2, 3]
>>> lista2
[1, 2, 3, 4]
```

Usa `lista1[:]` para crear una **copia superficial** y evitar cambios inesperados.

Desempaquetado en bucles

```
>>> lista_pares = [(1, 2), (3, 4), (5, 6)]
>>> for a, b in lista_pares:
...     print(a, b)
1 2
3 4
5 6
```

Desempaquetado de tuplas dentro de un bucle para acceder directamente a los valores.

3. Importaciones

Importar un módulo

```
>>> import time
```

Importa el módulo `time` para funciones relacionadas con el tiempo.

Obtener el tiempo actual en segundos

```
>>> time.time()
1499517005.3954754
```

`time.time()`: Devuelve el tiempo actual en segundos desde el **epoch** (1 de enero de 1970).

Importar una función específica del módulo `time`

```
>>> from time import ctime
>>> ctime(1499517005.3954754)
'Sat Jul  8 09:30:10 2017'
```

`from time import ctime`: Importa solo la función `ctime` del módulo `time`.
`ctime(segundos)`: Convierte el valor de segundos en una cadena de fecha legible.

Importar `time` con un alias (`as`)

```
>>> import time as t
>>> t.time()
1499517005.3954754
```

`import time as t`: Importa el módulo `time` y lo renombra como `t`.

Ahora puedes usar `t.time()` en lugar de `time.time()`.

Importar varias funciones desde `time`

```
>>> from time import time, ctime
>>> time()
1499517005.3954754
>>> ctime(1499517005.3954754)
'Sat Jul  8 09:30:10 2017'
```

`from time import time, ctime`: Importa las funciones `time` y `ctime`.
`time()`: Retorna el tiempo actual en segundos.
`ctime(segundos)`: Convierte segundos a formato de fecha legible.

Importar todo el contenido de `time` (`*`)

```
>>> from time import *
>>> time()
1499517005.3954754
>>> ctime(1499517005.3954754)
'Sat Jul  8 09:30:10 2017'
```

`from time import *`: Importa **todas las funciones** del módulo `time`.

⚠️ **Nota:** No es recomendable importar todo con `*` porque puede generar conflictos si hay funciones con el mismo nombre.

Obtener ayuda sobre una función específica

```
>>> help(time.ctime)
```

`help(time.ctime)`: Muestra documentación y ejemplos de uso de la función `ctime`.

Importaciones absolutas

Una **importación absoluta** especifica la ruta completa desde el directorio raíz del proyecto.
Ejemplo: Si tienes esta estructura de carpetas:

```
/mi_proyecto/
├── main.py
└── modulos/
    └── reloj.py
```

Para importar `reloj.py` en `main.py`, podrías usar:

```
import modulos.reloj
```

Ventaja: Son más fáciles de entender y funcionan bien con proyectos grandes.

Importaciones relativas

```
from . import reloj
```

```
from ..modulos import reloj
```

Las **importaciones relativas** utilizan un punto `.` para indicar la ubicación del módulo relativo al archivo actual.

Ejemplo: Si `reloj.py` está en la misma carpeta que el archivo actual:

```
from . import reloj
```

Si `reloj.py` está en una carpeta superior:
```
from ..modulos import reloj
```

Diferencias clave:

Importación Absoluta: Se basa en la ruta desde la raíz del proyecto.
Importación Relativa: Se basa en la ubicación del archivo que ejecuta la importación.

¿Por qué es importante desde dónde se ejecuta el código?

El punto clave aquí es **el valor de `sys.path`** y el **directorio actual** desde donde se ejecuta el script.
Cuando ejecutas un script desde la terminal, Python añade automáticamente la carpeta actual al **`sys.path`**, lo que afecta cómo se resuelven los módulos.

Ejemplo Importancia del contexto de ejecución:

Estructura:
```
/mi_proyecto/
├── main.py
└── modulos/
    └── reloj.py
```

Si ejecutas desde la raíz del proyecto:

```
$ python3 main.py
```

Puedes usar:

```
import modulos.reloj
```

Si ejecutas `main.py` desde dentro de `modulos`:

```
$ cd modulos
$ python3 ../main.py
```

El comportamiento puede cambiar, y las **importaciones relativas** pueden fallar si no están bien configuradas.

Evita errores comunes:

Evita importar módulos con nombres que coincidan con módulos estándar.
Ejemplo: No llames a tu archivo `time.py` si importas `time`.

Usa `sys.path` si necesitas modificar la ruta de búsqueda.

```
import sys
sys.path.append('/ruta/a/mi_proyecto/modulos')
```

3. **Comprueba siempre el directorio actual antes de ejecutar.**

```
import os
print(os.getcwd())
```

Errores Comunes por importaciones incorrectas:

ImportError: Cuando el módulo no se encuentra.
ModuleNotFoundError: Si el módulo no está disponible en el `sys.path`.
ValueError: Ocurre si intentas usar importaciones relativas fuera de un paquete.

---

4. Operaciones básicas

Operaciones aritméticas

```
>>> 5 + 7
12
>>> 5 - 7
-2
>>> 5 * 6
30
>>> 5 / 2
2.5
>>> 5 // 2  # División entera
2
>>> 5 % 2   # Módulo (residuo)
1
>>> 2 ** 5  # Potencia
32
```

División y módulo juntos

```
>>> divmod(5, 2)
(2, 1)
```

`divmod(a, b)` devuelve `(cociente, residuo)`.

---

5. Manipulación de strings

Definición y slicing

```
>>> t = "Hola mundo"
>>> t[0]   # Primer carácter
'H'
>>> t[-1]  # Último carácter
'o'
>>> t[2:6]  # Substring desde la posición 2 hasta 5
'la m'
>>> t[::-1]  # Inversión de cadena
'odnum aloH'
```

---

f-Strings: Formateo de cadenas

Los **f-strings** (formatted string literals) son una forma moderna y eficiente de formatear cadenas en Python. Introducidos en **Python 3.6**, permiten incrustar expresiones dentro de cadenas de texto usando llaves `{}`.

Ventajas de los f-strings:

Más rápidos y fáciles de leer que otros métodos.
Permiten evaluar expresiones directamente en el formato.
Se pueden usar para manipular y formatear datos dinámicos.

Sintaxis de un f-string

Un **f-string** se define precediendo la cadena con una **`f`** o **`F`** antes de las comillas:

```
f"texto {expresión}"
```

Ejemplo Básico:

```
nombre = "Juan"
edad = 25
```

```
print(f"Hola, {nombre}. Tienes {edad} años.")
# Salida: Hola, Juan. Tienes 25 años.
```

Métodos importantes de cadenas

```
>>> t.upper()     # Mayúsculas
'HOLA MUNDO'
>>> t.lower()     # Minúsculas
'hola mundo'
>>> t.capitalize()  # Primera letra mayúscula
'Hola mundo'
>>> t.title()     # Cada palabra con mayúscula
'Hola Mundo'
>>> t.replace("mundo", "Python")
'Hola Python'
```

---

6. Listas y Tuplas

En Python, tanto las **listas** como las **tuplas** son estructuras de datos que permiten almacenar colecciones de elementos. La diferencia principal radica en que las listas son **mutables** (sus valores pueden cambiar), mientras que las tuplas son **inmutables** (no se pueden modificar después de creadas).

Una **lista** es una secuencia ordenada de elementos que puede contener datos de diferentes tipos. Se define usando corchetes `[]` y los elementos están separados por comas.

Características de las listas:

Mutable: Los valores pueden modificarse después de la creación.
Heterogénea: Puede contener elementos de distintos tipos.
Indexada: Los elementos se acceden a través de índices, comenzando desde `0`.
Dinámica: Se pueden agregar, eliminar o modificar elementos en cualquier momento.

Definición de lista

```
>>> l = [56, "24", 5, [1, 2, 3]]
```

Acceso a Elementos en una lista:

```
>>> mi_lista = [10, 20, 30, 40, 50]
>>> mi_lista[0]
10
>>> mi_lista[-1]
50
```

`mi_lista[0]` → Accede al primer elemento (`10`).
`mi_lista[-1]` → Accede al último elemento (`50`).

Modificación de una lista:

```
>>> mi_lista[1] = 99
>>> mi_lista
[10, 99, 30, 40, 50]
```

Se cambia el valor del índice `1` de `20` a `99`.

Listas anidadas:

```
>>> lista_anidada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> lista_anidada[1][2]
6
```

Las listas pueden contener otras listas, formando matrices o estructuras complejas.

Listas por comprensión:

```
>>> cuadrados = [x ** 2 for x in range(5)]
>>> cuadrados
[0, 1, 4, 9, 16]
```

List comprehension: Crea listas de manera más concisa.

Ventajas de las listas

✅ Flexibilidad para modificar su contenido.
✅ Fácil manipulación con métodos incorporados.
✅ Permite elementos heterogéneos.

Desventajas de las listas

❗ Más lentas que las tuplas al iterar.
❗ Consumen más memoria.

Métodos importantes de listas

```
>>> l.append("hola")  # Añadir al final
```

```
>>> l.extend([7, 8])  # Extender con otra lista
>>> l.insert(2, "nuevo")  # Inserta en posición 2
>>> l.remove(5)  # Elimina el valor 5
>>> l.pop()  # Elimina el último elemento
>>> l.sort()  # Ordenar lista
```

Tuplas

Una **tupla** es una colección **ordenada** e **inmutable** de elementos. Se define usando **paréntesis** `()` y los elementos están separados por comas.

```
>>> t = (3, 4, 5, 6)
>>> t[1]
4
```

⚠️ **Nota:** Las tuplas son **inmutables**, no se pueden modificar.

Características de las tuplas:

Inmutable: Los valores no pueden cambiar después de la creación.
Heterogénea: Puede contener distintos tipos de elementos.
Indexada: Los elementos son accesibles mediante índices, como las listas.
Más rápida: Es más eficiente que las listas para iteraciones.

Acceder a elementos en una tupla:

```
>>> mi_tupla = (10, 20, 30, 40, 50)
>>> mi_tupla[0]
10
>>> mi_tupla[-1]
50
```

`mi_tupla[0]` → Accede al primer elemento (`10`).
`mi_tupla[-1]` → Accede al último elemento (`50`).

Intentando de modificar una tupla:

```
>>> mi_tupla[1] = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Error: No se puede modificar una tupla.

Conversión entre listas y tuplas:

```
>>> lista = [1, 2, 3]
>>> tupla = tuple(lista)
>>> tupla
(1, 2, 3)
>>> lista_nueva = list(tupla)
>>> lista_nueva
[1, 2, 3]
```

`tuple(lista)`: Convierte una lista a tupla.
`list(tupla)`: Convierte una tupla a lista.

Desempaquetado de tuplas:

```
>>> tupla = (1, 2, 3)
>>> a, b, c = tupla
>>> a
1
>>> b
2
>>> c
3
```

Unpacking: Asigna los valores de la tupla a variables individuales.

Ventajas de las tuplas:

✅ Más rápidas que las listas.
✅ Seguras, ya que no pueden ser modificadas.
✅ Consumen menos memoria.

Desventajas de las tuplas:

❗ No se pueden modificar después de su creación.
❗ Menos flexibles que las listas.

---

7. Operadores bitwise

Los **operadores bitwise** o **operadores a nivel de bits** permiten manipular los **bits individuales** de un número entero. Estos operadores convierten los valores a su representación **binaria** y realizan operaciones **bit a bit**.

¿Qué son los Bits?

Un **bit** es la unidad más pequeña de información que una computadora puede manejar. Puede tener solo dos valores:

0 → Apagado / Falso
1 → Encendido / Verdadero
Los números enteros se representan en **binario**, lo que permite manipularlos con operaciones bitwise.

AND bitwise (&)

Realiza una **AND lógica** entre los bits de dos números. El resultado es 1 solo si ambos bits son 1.

Ejemplo:

```
>>> a = 5    # 0b0101
>>> b = 3    # 0b0011
>>> a & b
1 en binario → 0b0001
5 en binario → 0b0101
3 en binario → 0b0011
```

`0b0101 & 0b0011 = 0b0001` → **1**

OR bitwise (|)

Realiza una **OR lógica** entre los bits de dos números. El resultado es 1 si al menos uno de los bits es 1.

Ejemplo:

```
>>> a = 5    # 0b0101
>>> b = 3    # 0b0011
>>> a | b
7 en binario → 0b0111
5 en binario → 0b0101
3 en binario → 0b0011
```

`0b0101 | 0b0011 = 0b0111` → **7**

XOR bitwise (^)

El operador **XOR bitwise** (`^`) compara los **bits** de dos números binarios **posición por posición**.

Regla Principal:

Si los **bits son diferentes**, el resultado es **`1`**.
Si los **bits son iguales**, el resultado es **`0`**.

¿Qué es XOR (O Exclusivo)?

XOR (del inglés **eXclusive OR**) es una operación lógica que devuelve **`True`** si exactamente uno de los bits es `1` y el otro es `0`.

Si ambos bits son **`0`** o **`1`**, devuelve **`0`**.

Realiza una **XOR lógica** (o exclusivo) entre los bits. El resultado es `1` si los bits son diferentes, y `0` si son iguales.

Ejemplo:

```
>>> a = 5    # 0b0101
>>> b = 3    # 0b0011
>>> a ^ b
```

```
6 en binario → 0b0110
5 en binario → 0101
3 en binario → 0011
```

`0b0101 ^ 0b0011 = 0b0110` → **6**

NOT bitwise (~)

El operador **NOT** (~) invierte todos los bits del número. Cambia los `1` por `0` y los `0` por `1`. También convierte un número positivo en su complemento negativo.

Ejemplo:

```
>>> a = 5    # 0b0101
>>> ~a
-6  # complemento de dos
```

Cálculo:

`5 = 0b0101`
`~5 = -6` (Complemento de dos, `-(n + 1)`)

⚠️ **Nota:** El resultado es `-6` debido al sistema de representación **complemento a dos**.

Desplazamiento a la izquierda (`<<`)

Desplaza los bits de un número hacia la **izquierda**, agregando ceros (`0`) al final. Esto es equivalente a multiplicar el número por `2^n`.

Ejemplo:

```
>>> a = 5     # 0b0101
>>> a << 1
10  # 0b1010
```

`5 = 0b0101`
`5 << 1` → Desplaza los bits una posición a la izquierda, agregando un `0` al final:
`0b1010 = 10`

Desplazamiento a la derecha (`>>`)

Desplaza los bits de un número hacia la **derecha**, descartando los bits sobrantes. Esto es equivalente a dividir el número por `2^n`.

Ejemplo:

```
>>> a = 5     # 0b0101
>>> a >> 1
2  # 0b0010
```

`5 = 0b0101`
`5 >> 1` → Desplaza los bits una posición a la derecha:

`0b0010 = 2`

Ejemplos combinados:

```
>>> a = 12    # 0b1100
```

```
>>> b = 5    # 0b0101

# AND
>>> a & b
4   # 0b0100

# OR
>>> a | b
13  # 0b1101

# XOR
>>> a ^ b
9   # 0b1001

# NOT
>>> ~a
-13  # complemento de dos

# Desplazamiento a la izquierda
>>> a << 2
48  # 0b110000

# Desplazamiento a la derecha
>>> a >> 2
3   # 0b0011
```

Complemento a dos

El operador **NOT** (~) usa el **complemento a dos**, lo que significa que:

~x es igual a **`-(x + 1)`**.

Ejemplo:

```
>>> x = 5
>>> ~x
-6
```

Complemento de 5:

Invierte bits: `~0b0101 = 0b1010`

Resultado en decimal es `-(5 + 1) = -6`.

Tabla de operadores bitwise

Operador	Nombre	Símbolo	Ejemplo	Resultado
AND	Y lógico	`&`	`a & b`	1 si ambos bits son 1
OR	O lógico	`|`	`a | b`	`a
XOR	XOR lógico	`^`	`a ^ b`	1 si los bits son diferentes
NOT	Negación (~)	`~`	`~a`	Invierte los bits
Shift L	Desplazamiento izq.	`<<`	`a << n`	Desplaza bits a la izquierda
Shift R	Desplazamiento der.	`>>`	`a >> n`	Desplaza bits a la derecha

Conversión binaria

```
>>> bin(7)
'0b111'
```

---

8. Conversión de tipos

La **conversión de tipos** (también llamada **coerción de tipos** o **casting**) permite transformar un valor de un tipo de dato a otro. Python ofrece dos formas principales de conversión de tipos:

1. **Conversión implícita (automática):** Python convierte automáticamente un tipo de dato a otro cuando es necesario.

2. **Conversión explícita (manual):** El programador convierte un valor usando funciones de conversión como `int()`, `float()`, `str()`, etc.

Conversión implícita (Automática)

Python realiza **conversión implícita** cuando se mezclan diferentes tipos de datos en una operación, y convierte los valores de menor jerarquía a un tipo de mayor jerarquía para evitar pérdida de información.

Ejemplo:

```
>>> a = 5    # Entero
>>> b = 2.5  # Flotante
>>> c = a + b
>>> c
7.5
```

Explicación:

Python convierte automáticamente el entero (`5`) a flotante (`5.0`), antes de realizar la suma.

El resultado es un **flotante** (`7.5`).

Precaución en conversión implícita:

```
>>> x = True
>>> y = 5
>>> resultado = x + y
>>> resultado
6
```

Python convierte automáticamente `True` a `1` (porque `True == 1`) y realiza la suma.

El resultado es `6`.

Conversión de tipos en expresiones

Suma de enteros y flotantes

```
>>> resultado = 5 + 3.5
>>> resultado
8.5
```

Conversión implícita: `5` es convertido a flotante (`5.0`).

Conversión de booleanos en operaciones

```
>>> resultado = True + 5
>>> resultado
6
```

`True` es tratado como `1` en operaciones aritméticas.

Entrada desde el usuario

```
>>> edad = input("Ingresa tu edad: ")
>>> edad
'25'
>>> int(edad) + 5
30
```

`input()` siempre devuelve una **cadena**, por lo que es necesario convertirla antes de usarla como número.

Conversión explícita (Manual)

La **conversión explícita** o **casting** se realiza usando funciones específicas que transforman un valor de un tipo a otro.

Ejemplos de conversión manual:

Convertir a entero con `int()`

```
>>> int(3.9)
3
>>> int("45")
45
>>> int(True)
1
>>> int(False)
0
```

⚠️ **Nota:** Al convertir un flotante a entero, Python **trunca** los decimales (no redondea).

Convertir a flotante con `float()`

```
>>> float(5)
5.0
>>> float("3.14")
3.14
>>> float(True)
1.0
```

Convertir a cadena con `str()`

```
>>> str(123)
'123'
>>> str(3.14)
'3.14'
>>> str(True)
'True'
```

Convertir a Booleano con `bool()`

Cualquier valor que no sea **`0`**, **`None`**, **`False`** **o cadena vacía** **`''`** se considera **True**.

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool("")
False
>>> bool("Python")
True
```

Convertir a lista con `list()`

```
>>> list((1, 2, 3))
[1, 2, 3]
>>> list("Hola")
['H', 'o', 'l', 'a']
```

Convertir a tupla con `tuple()`

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple("Hola")
('H', 'o', 'l', 'a')
```

Convertir a conjunto con `set()`

```
>>> set([1, 2, 2, 3, 4])
{1, 2, 3, 4}
>>> set("banana")
{'b', 'a', 'n'}
```

Nota: Elimina duplicados automáticamente.

Convertir a diccionario con `dict()`

```
>>> dict([(1, 'a'), (2, 'b')])
{1: 'a', 2: 'b'}
>>> dict(a=1, b=2)
{'a': 1, 'b': 2}
```

Se necesita una estructura válida para convertir a diccionario.

Conversión entre tipos de colecciones

```
>>> lista = [1, 2, 3]
>>> tuple(lista)
(1, 2, 3)

>>> tupla = (4, 5, 6)
>>> list(tupla)
[4, 5, 6]

>>> conjunto = {7, 8, 9}
>>> list(conjunto)
```

```
[7, 8, 9]
```

Errores comunes en conversión

Error al convertir una cadena inválida

```
>>> int("Hola")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hola'
```

No se puede convertir `"Hola"` a entero.

Error al convertir un valor no compatible

```
>>> dict([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

Para convertir a diccionario, los valores deben ser pares clave-valor. Ejemplo: {“a”: 1}

Funciones de conversión de tipos

Función	Convierte a	Ejemplo	Resultado
`int()`	Entero	`int(4.9)`	`4`
`float()`	Flotante	`float(5)`	`5.0`
`str()`	Cadena	`str(45)`	`'45'`
`bool()`	Booleano	`bool("")`	`False`
`list()`	Lista	`list((1, 2, 3))`	`[1, 2, 3]`
`tuple()`	Tupla	`tuple([1, 2, 3])`	`(1, 2, 3)`
`set()`	Conjunto	`set([1, 1, 2])`	`{1, 2}`
`dict()`	Diccionario	`dict([(1, 'a')])`	`{1: 'a'}`

Conclusión

La **conversión de tipos** es esencial para trabajar con diferentes tipos de datos en Python.
Usa **conversión implícita** cuando sea posible, pero **conversión explícita** si necesitas controlar los resultados.
Asegúrate de que los valores sean compatibles antes de convertirlos para evitar errores.

Conversión de números y cadenas

```
>>> int("2423")
2423
>>> int("24F2", 16)  # Hexadecimal a decimal
9458
>>> float("123.32")
123.32
>>> str(56)
'56'
```

---

9. Módulo `time`

El módulo **`time`** permite trabajar con funciones relacionadas con el tiempo en Python.
Ofrece herramientas para obtener el tiempo actual, formatearlo y medir la duración de procesos.

Obtener el tiempo actual

```
>>> import time
>>> time.time()
1499517005.3954754
```

`time.time()`: Retorna el tiempo actual en **segundos** desde el **epoch** (1 de enero de 1970, 00:00:00 UTC).
El valor devuelto es un **float** que representa el número de segundos transcurridos.

Convertir tiempo a formato legible

```
>>> time.ctime()
'Sat Jul  8 09:30:10 2017'
```

`time.ctime()`: Convierte el tiempo actual (o un valor de tiempo en segundos) a una **cadena legible**.

Si no se pasa un valor, utiliza el tiempo actual por defecto.

---

10. Manejo de errores

En Python, cuando ocurre un **error o excepción**, el programa se **detiene** y muestra un mensaje de error llamado **traceback**. Para evitar que el programa se detenga abruptamente, podemos **manejar los errores** utilizando bloques `try-except`.

¿Qué es una excepción?

Una **excepción** es un evento que interrumpe el flujo normal del programa cuando ocurre un error inesperado.

Ejemplos comunes de excepciones:

`ZeroDivisionError`: Ocurre cuando intentas dividir por cero.
`ValueError`: Ocurre cuando intentas convertir un valor no válido a un tipo específico.
`TypeError`: Se produce cuando un operador o función se aplica a un tipo de dato incorrecto.
`IndexError`: Aparece cuando intentas acceder a un índice fuera del rango de una lista o tupla.

División por Cero (`ZeroDivisionError`)

```
>>> 345 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Ocurre cuando intentas dividir un número por **cero**.
El programa se detiene y muestra un **traceback**.

Conversión fallida (`ValueError`)

```
>>> int("123.32")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```

Ocurre al intentar convertir una cadena que no es un número entero.
`int("123.32")` genera un **`ValueError`** porque el valor no es un entero válido.

¿Cómo evitar que el programa se detenga?

Para **manejar errores** y evitar que el programa termine abruptamente, usamos el bloque `try-except`.

Manejo de errores con `try-except`

El bloque **`try`** te permite probar un bloque de código para detectar errores.
Si ocurre una excepción, el control pasa al bloque **`except`**, donde puedes manejar el error.

Sintaxis básica

```
try:
    # Código que puede generar una excepción
except TipoDeError: # Acá de debe poner el tipo de error
    # Código que se ejecuta si ocurre ese error
```

Manejo de división por cero

```
try:
    resultado = 345 / 0
except ZeroDivisionError:
    print("Error: No puedes dividir entre cero.")
```

Salida:

```
Error: No puedes dividir entre cero.
```

Manejo de conversión fallida

```
try:
    valor = int("123.32")
except ValueError:
    print("Error: No es posible convertir a entero.")
```

Salida:

```
Error: No es posible convertir a entero.
```

Manejo de múltiples excepciones

Puedes manejar **múltiples excepciones** usando varios bloques `except`.

```
try:
    x = int("Hola")
    y = 5 / 0
except ValueError:
    print("Error: Conversión fallida.")
except ZeroDivisionError:
    print("Error: División por cero.")
```

Salida:

```
Error: Conversión fallida.
```

Bloques `else` y `finally`

Además de `try` y `except`, también puedes usar:

- **`else`**: Se ejecuta si **no** ocurre ninguna excepción.
- **`finally`**: Se ejecuta **siempre**, ocurra o no una excepción.

Sintaxis completa

```
try:
    # Código que puede generar excepciones
except TipoDeError:
    # Código si ocurre una excepción
else:
    # Código que se ejecuta si NO ocurre una excepción
finally:
    # Código que se ejecuta SIEMPRE (opcional)
```

Ejemplo completo

```
try:
    valor = int(input("Ingresa un número: "))
    resultado = 10 / valor
except ValueError:
    print("Error: Debes ingresar un número entero.")
except ZeroDivisionError:
    print("Error: No puedes dividir entre cero.")
else:
    print(f"Resultado: {resultado}")
finally:
    print("Fin del programa.")
```

Caso 1: Entrada Correcta

```
Ingresa un número: 5
Resultado: 2.0
Fin del programa.
```

Caso 2: Conversión Inválida

```
Ingresa un número: Hola
Error: Debes ingresar un número entero.
Fin del programa.
```

Caso 3: División por Cero

```
Ingresa un número: 0
Error: No puedes dividir entre cero.
Fin del programa.
```

Explicación:

1. El bloque `try` ejecuta el código que puede generar excepciones.
2. Si ocurre una excepción, el control pasa al bloque `except` correspondiente.
3. Si **no** ocurre ninguna excepción, el bloque `else` se ejecuta.
4. El bloque `finally` se ejecuta **siempre**, sin importar si hubo o no excepciones.

El manejo de errores con **`try-except`** evita que el programa termine abruptamente.
Usa **`else`** si deseas ejecutar código solo si **no** ocurre una excepción.
Usa **`finally`** para ejecutar código que debe ejecutarse **siempre**, incluso si ocurre un error.

---

11. Funciones y métodos importantes

Una **función** es un bloque de código reutilizable que realiza una tarea específica. Ejemplo: `print()`, `len()`, etc.

Un **método** es una función asociada a un objeto que puede modificar su comportamiento o devolver información. Los métodos son llamados usando la sintaxis.

Además existen **funciones integradas (built-in)** que permiten trabajar con diferentes tipos de datos, manipular estructuras de datos, y realizar operaciones básicas sin necesidad de importar módulos externos.
Salida:

¿Cómo ver métodos disponibles? - `dir(objeto)`

```
>>> dir("Hola")
```

Salida:

```
['__add__', '__class__', 'capitalize', 'center', 'count', 'encode',
'endswith', 'find', 'format', 'index', 'isalnum', 'isalpha', 'join',
'lower', 'replace', 'split', 'strip', 'upper', 'zfill']
```

Obtener ayuda de un método - `help(objeto)`

```
>>> help(str.capitalize)
```

Obtener el tipo de un objeto - `type(objeto)`

La función **`type(objeto)`** devuelve el tipo de dato del objeto pasado como argumento.

```
>>> type(5)
<class 'int'>

>>> type("Hola")
<class 'str'>

>>> type([1, 2, 3])
<class 'list'>
```

Usos comunes:
Identificar el tipo de un valor o variable:

```
>>> a = 3.14
>>> type(a)
<class 'float'>
```

Obtener la longitud de un objeto - `len(objeto)`

La función **`len(objeto)`** devuelve el número de elementos de una secuencia o colección.

```
>>> len("Hola mundo")
10
>>> len([1, 2, 3, 4])
4
>>> len({"a": 1, "b": 2, "c": 3})
3
```

Usos comunes:
Obtener la longitud de una cadena:

```
>>> texto = "Python"

>>> len(texto)
6
```

Obtener la cantidad de elementos en una lista o diccionario.

Obtener la identidad de un objeto - `id(objeto)`

La función **`id(objeto)`** devuelve la **dirección de memoria** donde está almacenado el objeto.

```
>>> a = 5
>>> id(a)
140722080798416
```

Usos comunes:
Verificar si dos variables apuntan al mismo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(a) == id(b)
True
```

Verificar tipo de dato - `isinstance()`

La función **`isinstance(objeto, tipo)`** devuelve `True` si el objeto es del tipo especificado y `False` si no lo es.

```
>>> isinstance(5, int)
True
>>> isinstance("Hola", str)
True
>>> isinstance([1, 2, 3], list)
True
>>> isinstance(3.14, int)
False
```

Usos comunes:
Validar tipos antes de realizar operaciones:

```
>>> if isinstance(5, int):
...     print("Es un entero.")
```

Entrada de datos desde la terminal / consola

La función **`input()`** permite al usuario ingresar datos desde la consola.

```
>>> nombre = input("¿Cómo te llamas? ")
¿Cómo te llamas? Juan
>>> print(f"Hola, {nombre}!")
Hola, Juan!
```

Nota: Los valores ingresados son **siempre cadenas**. Usa `int()` o `float()` si necesitas convertirlos.

Generar una secuencia de números

La función **`range()`** genera una secuencia de números enteros desde un valor inicial hasta uno final.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

```
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
```

Usos Comunes:
Crear bucles para iterar sobre un rango de valores:

```
for i in range(3):
    print(i)
# Salida:
# 0
# 1
# 2
```

Sumar los elementos de una secuencia

La función **`sum()`** suma todos los elementos de una lista, tupla o cualquier secuencia iterable.

```
>>> sum([1, 2, 3, 4, 5])
15
```

```
>>> sum((10, 20, 30))
60
```

Usos comunes:
Sumar valores de una lista:

```
numeros = [5, 10, 15]
total = sum(numeros)
print(total)  # 30
```

Obtener el valor máximo o mínimo

`max()` devuelve el valor más grande de una secuencia.
`min()` devuelve el valor más pequeño de una secuencia.

```
>>> max([5, 10, 2, 8])
10

>>> min([5, 10, 2, 8])
2
```

Usos comunes:
Encontrar el valor máximo/mínimo en listas o tuplas.

Ordenar una secuencia

La función **`sorted()`** devuelve una lista ordenada de los elementos de una secuencia.

```
>>> sorted([5, 2, 9, 1])
[1, 2, 5, 9]

>>> sorted("python")
['h', 'n', 'o', 'p', 't', 'y']
```

Usos comunes:
Ordenar listas, cadenas o cualquier objeto iterable.

Aplicar una función a una secuencia

La función **`map(función, secuencia)`** aplica una función a cada elemento de una secuencia y devuelve un **iterador**.

```
>>> numeros = [1, 2, 3, 4]
>>> cuadrados = map(lambda x: x ** 2, numeros)
>>> list(cuadrados)
[1, 4, 9, 16]
```

Usos comunes:
Transformar elementos de una lista sin usar bucles.

Filtrar elementos de una secuencia

La función **`filter(función, secuencia)`** devuelve un **iterador** que contiene solo los elementos que cumplen con la condición dada por la función.

```
>>> numeros = [1, 2, 3, 4, 5, 6]
>>> pares = filter(lambda x: x % 2 == 0, numeros)
>>> list(pares)
[2, 4, 6]
```

Usos comunes:
Filtrar valores específicos de una lista o secuencia.

¿Qué es `lambda`?

Una **`lambda`** es una **función anónima** que puede tener cualquier número de argumentos, pero solo **una expresión**. Se usa cuando necesitas definir funciones rápidas y de una sola línea.

Sintaxis:

```
lambda argumentos: expresión
```

Ejemplo básico:

```
>>> suma = lambda x, y: x + y
>>> suma(5, 3)
8
```

`suma` es una función que toma dos argumentos y devuelve su suma.

Aplicar una función a una secuencia - `map()`

La función **`map(función, secuencia)`** aplica una función a cada elemento de una secuencia y devuelve un **iterador**.

Ejemplo: Elevar al cuadrado con `lambda` y `map()`

```
>>> numeros = [1, 2, 3, 4, 5]
>>> cuadrados = map(lambda x: x ** 2, numeros)
>>> list(cuadrados)
[1, 4, 9, 16, 25]
```

Explicación:
`lambda x: x ** 2` → Eleva `x` al cuadrado.
`map()` aplica esta función a cada elemento de la lista `numeros`.

Ejemplo: Convertir números a cadena

```
>>> numeros = [1, 2, 3, 4]
>>> str_numeros = map(lambda x: str(x), numeros)
>>> list(str_numeros)
['1', '2', '3', '4']
```

Explicación:
`lambda x: str(x)` → Convierte cada número en cadena.
`map()` aplica esta función a cada número de la lista.

Filtrar Elementos que cumplen una condición

La función **`filter(función, secuencia)`** devuelve un **iterador** que contiene solo los elementos que cumplen la condición definida en la función.

Ejemplo: Filtrar números pares

```
>>> numeros = [1, 2, 3, 4, 5, 6, 7, 8]
>>> pares = filter(lambda x: x % 2 == 0, numeros)
>>> list(pares)
[2, 4, 6, 8]
```

Explicación:
`lambda x: x % 2 == 0` → Retorna `True` si el número es par.
`filter()` incluye solo los elementos que cumplen esta condición.

Ejemplo: Filtrar números mayores a 5

```
>>> numeros = [1, 4, 6, 9, 3, 10]
>>> mayores = filter(lambda x: x > 5, numeros)
>>> list(mayores)
[6, 9, 10]
```

Explicación:
`lambda x: x > 5` → Devuelve `True` si el número es mayor que 5.
`filter()` conserva solo los números que cumplen esta condición.

Ordenar secuencias personalizadas

La función **`sorted()`** ordena una lista o secuencia. Si se pasa una función como argumento, permite personalizar el criterio de ordenación.

Ejemplo: Ordenar números descendentes

```
>>> numeros = [5, 1, 8, 3, 9]
>>> orden_desc = sorted(numeros, key=lambda x: -x)
>>> orden_desc
[9, 8, 5, 3, 1]
```

Explicación:
`lambda x: -x` → Invierte el valor para ordenar de mayor a menor.
`sorted()` ordena los elementos según este criterio.

Ejemplo: Ordenar por longitud de palabras

```
>>> palabras = ["Python", "es", "genial", "y", "potente"]
>>> ordenadas = sorted(palabras, key=lambda x: len(x))
>>> ordenadas
['y', 'es', 'Python', 'genial', 'potente']
```

Explicación:
`lambda x: len(x)` → Usa la longitud de cada palabra como criterio de ordenación.
`sorted()` ordena las palabras por longitud.

Ejemplo: Ordenar diccionarios por valor

```
>>> diccionario = {'a': 3, 'b': 1, 'c': 5}
>>> orden_dicc = sorted(diccionario.items(), key=lambda x: x[1])
>>> orden_dicc
[('b', 1), ('a', 3), ('c', 5)]
```

Explicación:
`lambda x: x[1]` → Usa el valor como criterio de ordenación.
`sorted()` ordena los pares clave-valor en función de los valores.

Conclusión

Las **funciones integradas** de Python son herramientas poderosas que facilitan el desarrollo de programas eficientes.
Usa `dir()` para explorar métodos y atributos.
Usa `help()` para obtener documentación detallada.
Aprende y domina las funciones más comunes para mejorar tus habilidades en Python.
