# 🐍 Guía de Python 🐍

## *Niveles 0 a 5*

### 🧩 Nivel 0 – Tipos y funciones básicas

*Aprender a declarar variables de distintos tipos y usar las funciones print() e input().*

```
# Declaración de variables básicas

numero_entero = 10              # entero (int)
numero_decimal = 3.14           # decimal (float)
cadena = "Hola mundo"           # cadena de texto (str)
booleano = True                 # booleano (bool)
nulo = None                     # valor nulo (NoneType)

# Mostrar por pantalla
print("Entero:", numero_entero)
print("Decimal:", numero_decimal)
print("Cadena:", cadena)
print("Booleano:", booleano)
print("Nulo:", nulo)

# Input del usuario
nombre = input("¿Cómo te llamas? ")
print("Hola,", nombre)
```

💡 Tip: todo lo que se escribe con input() se guarda como texto (str).

### ➕ Nivel 1 – Operaciones matemáticas

*Usar operadores aritméticos con números.*

```
# Operaciones básicas
a = 8
b = 3

suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b  # división con decimales
modulo = a  % b # devuelve el resto de la división
```

```
print("Suma:", suma)
print("Resta:", resta)
print("Multiplicación:", multiplicacion)
print("División:", division)

# Operaciones adicionales
division_entera = a // b   # división entera
modulo = a % b             # resto de la división
potencia = a ** b          # potencia

print("División entera:", division_entera)
print("Módulo:", modulo)
print("Potencia:", potencia)
```

💡 Tip:
Puedes convertir cadenas a números con int() o float(), por ejemplo:
**x = int(input("Introduce un número entero: "))**

## 🔀 Nivel 2 – Condicionales (if, else, elif)

*Tomar decisiones con if, else y operadores lógicos (and, or, not).*

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

if edad >= 18:
    print("Eres mayor de edad.")
else:
    print("Eres menor de edad.")

# Condiciones combinadas
nota = float(input("Introduce tu nota: "))

if nota >= 9:
    print("Excelente")
elif nota >= 7:
    print("Bien")
elif nota >= 5:
    print("Aprobado")
else:
    print("Suspenso")

# Uso de and
```

```
# Para que and funcione, ambas se cumplen
temperatura = 25
if temperatura > 20 and temperatura < 30:
    print("El clima es agradable.")
# Uso de or
esPrimo = 2
if esPrimo % 2 == 0 or esPrimo // 2 = 1
    print('El numero es 2')
# Para que or funcione, una de las condiciones se debe cumplir
# Uso de not
# Solo se usa para negar el cumplimiento de algo. No lo usaras por ahora pero más adelante verás un caso con diccionarios.
```

## 🧭 Nivel 3.1 — switch y match

Python no tiene un switch tradicional como otros lenguajes (C, Java, etc.) hasta Python 3.10, que introdujo una estructura muy potente llamada **match / case** (pattern matching). Te muestro varias formas — desde la más clásica y compatible hasta la forma moderna y recomendada.

1) En versiones anteriores a 3.10: usar dict de funciones o valores

```
def a():
    return "Opción A"

def b():
    return "Opción B"

switch = {
    "A": a,
    "B": b,
}

op = "A"
resultado = switch.get(op, lambda: "Opción por defecto")()
print(resultado)  # "Opción A"
```

Tip: usar funciones como valores evita largos if/elif/....

**2) Python 3.10+: `match` / `case` (pattern matching)**

Muy expresivo; sirve para comparar valores, estructuras y hacer destructuring:

```
def describe(value):
    match value:
        case 0:
            return "cero"
        case 1 | 2:
            return "uno o dos"
        case int() as n if n < 0:
            return "entero negativo"
        case [a, b]:
            return f"lista de dos elementos: {a}, {b}"
        case {"tipo": "persona", "nombre": name}:
            return f"persona: {name}"
        case _:
            return "otro"

print(describe([10, 20])) # lista de dos elementos: 10, 20
```

Tips:

- **match** es poderoso: puede hacer coincidir tipos, valores, patrones anidados y condiciones (if dentro del case).
- **No lo uses si tu problema es simple:** a veces dict o if son más legibles.
- **case _:** es el "default".

## 🔁 Nivel 3 – Bucles (while y for)

Repetir acciones usando while y for.

🔸 Bucle while

```
contador = 0
while contador < 5:
    print("Contador vale:", contador)
    contador += 1  # incrementa en 1 cada vez
```

💡 Cuidado:
El while necesita una condición que se vuelva falsa en algún momento, o tendrás un bucle infinito.

🔸 Bucle for

```
# Recorrer un rango de números
for i in range(5):  # del 0 al 4
    print("i =", i)
```

## 🧮 Nivel 4 – Funciones matemáticas en Python (math)

El módulo estándar math ofrece muchas funciones numéricas y constantes. Importa con:

```
#Importar libreria math
import math
```

### Constantes importantes

```
math.pi    # π
math.e     # e
math.tau    # 2π
```

### Funciones trigonométricas y de ángulos

```
math.sin(x)      # seno (x en radianes)
math.cos(x)
math.tan(x)
math.asin(x)     # arco seno (devuelve radianes)
math.acos(x)
math.atan(x)
math.degrees(r)  # radianes → grados
math.radians(d)  # grados → radianes
```

### Potencias, raíces y logaritmos

```
math.sqrt(x)     # raíz cuadrada
math.pow(x, y)   # potencia (aunque usar x ** y es idiomático)
math.exp(x)      # e**x
math.log(x)      # log natural
math.log10(x)    # log base 10
math.log2(x)     # log base 2
```

### Redondeo y truncado

```
math.ceil(2.3)   # 3  (arriba)
math.floor(2.7)  # 2  (abajo)
math.trunc(2.9)  # 2  (parte entera)
round(2.5)       # 2 o 3 según regla de round() de Python (round to
even)
```

Tip: round() es builtin y tiene comportamiento de "round half to even". Si necesitas control distinto, usa math o decimal.

### Factoriales, combinatoria y gcd

```
math.factorial(5)   # 120
math.gcd(12, 18)    # 6  (máximo común divisor)
```

Para combinatoria avanzada usa math.comb (Python 3.8+): math.comb(n, k).

### Operaciones con float y seguridad numérica

```
math.isfinite(x)
math.isinf(x)
math.isnan(x)
math.copysign(x, y)  # devuelve x con signo de y
math.fmod(x, y)      # resto flotante similar a C
math.hypot(x, y)     # sqrt(x*x + y*y) estable numericamente
```

**Tip: para cálculos con mucha precisión (decimales financieros) prefiere decimal.Decimal. Para grandes arrays, usa NumPy.**

```
import math

angulo_deg = 30
angulo_rad = math.radians(angulo_deg)
print(math.sin(angulo_rad))  # 0.5

print(math.hypot(3, 4))      # 5.0
print(math.log10(1000))      # 3.0
```

## ✂️Nivel 5 — Funciones con str (manipulación de cadenas)

Las cadenas en Python son muy ricas en métodos. Recordatorio: las str son inmutables.

Operaciones y métodos básicos

```
s = "  Hola Mundo  "
s.lower()        # "  hola mundo  "
s.upper()        # "  HOLA MUNDO  "
```

```
s.capitalize()   # "  hola mundo  " -> sólo la primera letra de la cadena (tras espacios sigue igual)
s.title()        # "  Hola Mundo  "
s.strip()        # "Hola Mundo"  (quita espacios al inicio y final)
s.lstrip(), s.rstrip()
```

Buscar y reemplazar

```
s.find("Mundo")     # índice de la primera aparición o -1
s.rfind("o")        # índice de la última
s.index("o")        # igual que find, pero lanza ValueError si no existe
s.replace("Mundo", "Python")  # devuelve nueva cadena
```

Split y join

```
frases = "uno,dos,tres".split(",")  # ['uno','dos','tres']
", ".join(["a", "b", "c"])          # "a, b, c"
```

Verificaciones

```
"abc".isalpha()     # True (solo letras)
"123".isdigit()     # True (solo dígitos)
"abc123".isalnum()  # True (letras y números)
"   ".isspace()     # True
```

Slicing y acceso

```
t = "Python"
t[0]      # 'P'
t[-1]     # 'n'
t[1:4]    # 'yth'  (start inclusive, end exclusive)
t[:3]     # 'Pyt'
t[3:]     # 'hon'
```

Formateo (muy usado)

- f-strings (Python 3.6+): `f"{nombre} tiene {edad} años"`
- `str.format()`: `"{} tiene {}".format(nombre, edad)`
- `%` viejo: `"%s tiene %d años" % (nombre, edad)` (menos recomendable)

Ejemplo:

```
nombre = "Ana"
edad = 20
print(f"{nombre} tiene {edad} años")

#Mas métodos útiles:
s.startswith("Ho")
s.endswith("do")
s.count("o")          # cuenta ocurrencias
s.partition(" ")      # particiona en (before, sep, after)
s.splitlines()        # separa por saltos de línea
s.encode("utf-8")     # bytes
b = b"hola"
b.decode("utf-8")     # str
```

Buenas prácticas y tips

- Evita concatenar strings con + en bucles; usa join() para eficiencia.
- Para manipulación compleja (regex), usa el módulo re.
- Ten en cuenta la inmutabilidad: cada operación devuelve una nueva cadena.
- Usa f-strings para legibilidad y rendimiento.

## 🧩 Nivel 5.1 — Arrays (Listas), Tuplas y Diccionarios en Python

🎯 Objetivo del nivel

Aprender a usar las estructuras de datos básicas de Python:

- Listas (arrays dinámicos)
- Tuplas (listas inmutables)
- Diccionarios (mapas clave-valor)
- Sets (conjuntos sin duplicados)

Dominar estas estructuras te permitirá almacenar, organizar y manipular datos complejos de forma limpia y eficiente.

---

## 🔹 1. Listas (arrays dinámicos)

### 💡 Qué son

**Las listas son colecciones ordenadas y mutables.**
**Permiten almacenar valores de cualquier tipo, incluso mezclados.**

```
numeros = [1, 2, 3, 4, 5]
nombres = ["Ana", "Luis", "María"]
mixta = [1, "dos", 3.0, True]
```

### 📦 Operaciones básicas

```
# Acceso por índice
print(numeros[0])      # 1
print(numeros[-1])     # último elemento

# Modificar
numeros[2] = 10
print(numeros)         # [1, 2, 10, 4, 5]

# Añadir y eliminar
numeros.append(6)
numeros.insert(0, 0)    # inserta al principio
numeros.pop()           # elimina el último
numeros.remove(10)      # elimina el valor 10
```

### 🔁 Recorrer listas

```
for n in numeros:
    print(n)

# Enumerar índices
for i, n in enumerate(numeros):
    print(f"Índice {i}: valor {n}")
```

### 🧠 Operaciones útiles

```
print(len(numeros))        # longitud
print(sum(numeros))        # suma de todos
print(max(numeros))        # mayor valor
print(min(numeros))        # menor valor
```

### 🧩 Slicing (sublistas)

```
print(numeros[1:4])   # elementos del índice 1 al 3
```

```
print(numeros[:3])    # primeros 3
print(numeros[-2:])   # últimos 2
```

🧱 List comprehensions

Forma compacta de construir listas:

```
cuadrados = [x**2 for x in range(1, 6)]
pares = [x for x in numeros if x % 2 == 0]
```

💡 *Tip:* Usa append() en bucles si necesitas eficiencia y claridad; list comprehensions si buscas elegancia y velocidad.

🔸 2. Tuplas (listas inmutables)

💡 Qué son

Las tuplas son como listas, pero no se pueden modificar una vez creadas.

```
coordenada = (10, 20)
print(coordenada[0])  # 10
```

🧱 Cuándo usarlas

- Para datos constantes (no deben cambiar)
- Como claves en diccionarios
- Para devolver varios valores desde una función

```
def dividir(a, b):
    return (a // b, a % b)  # cociente, resto

resultado = dividir(9, 4)
print(resultado)  # (2, 1)
```

💡 *Tip:* Las tuplas se procesan más rápido que las listas → útiles en código de rendimiento crítico.

🔹 3. Diccionarios (mapas clave-valor)

💡 Qué son

Un diccionario almacena datos mediante pares “clave: valor”.
Las claves deben ser únicas e inmutables (como cadenas o tuplas)

```
persona = {
```

```
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Madrid"
}
```

### 📦 Operaciones básicas

```
# Acceder
print(persona["nombre"])

# Agregar o modificar
persona["email"] = "ana@example.com"
persona["edad"] = 26

# Eliminar
del persona["ciudad"]

# Obtener con valor por defecto (evita errores)
print(persona.get("pais", "Desconocido"))
```

### 🔁 Recorrer diccionarios

```
for clave, valor in persona.items():
    print(f"{clave}: {valor}")
```

### 🔧 Métodos útiles

```
persona.keys()     # devuelve solo las claves
persona.values()   # devuelve solo los valores
persona.items()    # devuelve pares clave-valor
```

💡 *Tip:* Usa `dict.get()` o el operador `in` para evitar errores:

```
if "email" in persona:
    print(persona["email"])
```

## 🔹 4. Sets (conjuntos)

### 💡 Qué son

**Colecciones sin duplicados y sin orden específico.**

```
colores = {"rojo", "verde", "azul"}
colores.add("amarillo")
colores.discard("verde")
```

### ⚙️ Operaciones de conjuntos

```
a = {1, 2, 3}
b = {3, 4, 5}
```

```
print(a | b)   # unión
print(a & b)   # intersección
print(a - b)   # diferencia
print(a ^ b)   # diferencia simétrica
```

💡 *Tip:* **Ideal para eliminar duplicados rápidamente:**

```
lista = [1, 2, 2, 3, 3, 3]
sin_repetidos = list(set(lista))
```

🧰 **Resumen rápido**

| Estructura | Ordenada | Mutable | Duplicados | Sintaxis |
|---|---|---|---|---|
| **List** | ✅ Sí | ✅ Sí | ✅ Sí | `[ ]` |
| **Tuple** | ✅ Sí | ❌ No | ✅ Sí | `( )` |
| **Dict** | ✅ (Python 3.7+) | ✅ Sí | ❌ No (claves) | `{clave: valor}` |
| **Set** | ❌ No | ✅ Sí | ❌ No | `{ }` o `set()` |

🧩 Ejemplo práctico combinando todo

Supón que queremos gestionar sesiones del Pomodoro (lista de diccionarios):

```
# Lista que guarda sesiones
sesiones = [
    {"fecha": "2025-11-13", "duracion": 25, "completada":
True},
    {"fecha": "2025-11-13", "duracion": 15, "completada":
False},
]

# Agregar nueva sesión
sesiones.append({"fecha": "2025-11-14", "duracion": 30,
"completada": True})

# Calcular tiempo total completado
total = sum(s["duracion"] for s in sesiones if
s["completada"])
print(f"Total de minutos productivos: {total}")
```

💡 *Tip real:* Esta estructura de datos es casi idéntica a la que usarás para guardar el historial en JSON de tu **Pomodoro Timer**.

🧠 Consejos finales para este nivel

1. **Listas → cuando necesites ordenar o contar elementos.**
2. **Tuplas → para datos constantes o retorno múltiple.**
3. **Diccionarios → para datos con identificadores o relaciones clave-valor.**
4. **Sets → para eliminar duplicados o comprobar pertenencia rápidamente.**
5. Aprende a **combinar** estructuras: listas de diccionarios, diccionarios de listas, etc.
6. En el Pomodoro y CLI, **usarás diccionarios como "objetos ligeros"** hasta que migres a clases (nivel 8).

# 🧩 Nivel 5.1 (ampliado) — Arrays, Diccionarios, Tuplas y el método `sorted()`

---

## 📘 Recordatorio: estructuras principales

Antes de aplicar `sorted()`, repasemos las tres estructuras clave:

| Estructura | Sintaxis | Mutable | Ordenada | Ejemplo |
|---|---|---|---|---|
| **Lista** | `[ ]` | ✅ Sí | ✅ Sí | `[1, 2, 3]` |
| **Tupla** | `( )` | ❌ No | ✅ Sí | `(1, 2, 3)` |
| **Diccionario** | `{clave: valor}` | ✅ Sí | ✅ (desde Python 3.7) | `{"a": 1, "b": 2}` |

## 🔹 1. Ordenar listas con `sorted()`

El método `sorted()` devuelve **una nueva lista ordenada**, sin modificar el original.

```
numeros = [5, 2, 9, 1, 7]
ordenados = sorted(numeros)

print(ordenados)   # [1, 2, 5, 7, 9]
print(numeros)     # [5, 2, 9, 1, 7] (no cambia)
```

🔄 Orden descendente

```
descendente = sorted(numeros, reverse=True)
print(descendente)  # [9, 7, 5, 2, 1]
```

💡 *Tip:* usa `reverse=True` cuando quieras mostrar los datos más recientes o grandes primero (por ejemplo, la sesión Pomodoro más larga).

🔸 **2. Ordenar cadenas de texto**

```
nombres = ["Ana", "carlos", "Bea", "david"]
orden = sorted(nombres)
print(orden)
📤 Resultado:
['Ana', 'Bea', 'carlos', 'david']
```

💡 ***Tip:*** **Python diferencia mayúsculas/minúsculas al ordenar.**
**Para ignorarlo:**
**orden = sorted(nombres, key=str.lower)**

## 🔹 3. Ordenar tuplas

**Las tuplas no son mutables, pero `sorted()` puede devolver una lista ordenada a partir de ellas.**

```
valores = (9, 3, 1, 7)
ordenados = sorted(valores)
print(ordenados)  # [1, 3, 7, 9]
```

💡 **Si necesitas mantenerlas como tuplas:**
**ordenados = tuple(sorted(valores))**

🔸 4. Ordenar diccionarios

Por defecto, `sorted()` aplicado a un diccionario ordena las claves.

```
persona = {"nombre": "Ana", "edad": 25, "altura": 1.70}
print(sorted(persona))  # ['altura', 'edad', 'nombre']
```

Pero generalmente queremos ordenar por valores, por ejemplo, de mayor a menor edad.

🔹 Ordenar por valores

```
datos = {"Ana": 25, "Luis": 19, "Carlos": 30}

# Ordenar por valor
ordenados = sorted(datos.items(), key=lambda x: x[1])
print(ordenados)
```

📤 Resultado:

```
[('Luis', 19), ('Ana', 25), ('Carlos', 30)]
```

💡 Para convertir de nuevo en diccionario:

```
ordenados_dict = dict(ordenados)
```

## 🔹 5. Ordenar listas de diccionarios (caso muy real)

Perfecto para tu proyecto Pomodoro o API JSON.

```
sesiones = [
    {"fecha": "2025-11-13", "duracion": 15, "completada":
False},
    {"fecha": "2025-11-13", "duracion": 25, "completada":
True},
    {"fecha": "2025-11-14", "duracion": 30, "completada":
True}
]

# Ordenar por duración
ordenadas = sorted(sesiones, key=lambda s: s["duracion"])
for s in ordenadas:
    print(s)
```

📤 Resultado:

```
{'fecha': '2025-11-13', 'duracion': 15, 'completada': False}
```

```
{'fecha': '2025-11-13', 'duracion': 25, 'completada': True}

{'fecha': '2025-11-14', 'duracion': 30, 'completada': True}
```

🔄 Orden descendente

```
ordenadas = sorted(sesiones, key=lambda s: s["duracion"], reverse=True)
```

💡 *Tip real:* en tu Pomodoro Timer, puedes mostrar las sesiones más largas primero o filtrar por completadas.

## 🔹 6. Ordenar por múltiples criterios

También puedes combinar condiciones.

Por ejemplo:
1️⃣ primero por fecha
2️⃣ luego por duración

```
sesiones_ordenadas = sorted(
    sesiones,
    key=lambda s: (s["fecha"], s["duracion"])
)
```

💡 Muy útil si tienes varios días y duraciones distintas.

### 🔸 7. Ordenar listas de tuplas

```
datos = [("Ana", 25), ("Luis", 30), ("Bea", 20)]
ordenados = sorted(datos, key=lambda x: x[1])
print(ordenados)  # [('Bea', 20), ('Ana', 25), ('Luis', 30)]
```

💡 *Tip:* muy común cuando trabajas con datos de APIs que devuelven tuplas.

## 🔹 8. Usar `sorted()` con `set`

Los conjuntos (`set`) no tienen orden, pero puedes convertirlos temporalmente:

```
colores = {"rojo", "verde", "azul"}
print(sorted(colores))  # ['azul', 'rojo', 'verde']
```

## ⚙️ 9. Comparativa entre `.sort()` y `sorted()`

| Característica | `list.sort()` | `sorted()` |
|---|---|---|
| Tipo de uso | método de lista | función global |
| Modifica la lista original | ✅ Sí | ❌ No |
| Devuelve una nueva lista | ❌ No (retorna None) | ✅ Sí |
| Funciona con otros iterables | ❌ Solo listas | ✅ Sí (tuplas, sets, dicts...) |

## 🧩 10. Ejemplo completo integrando todo

Supongamos que guardas tu historial Pomodoro en una lista de diccionarios (como en el nivel anterior):

```
sesiones = [
    {"fecha": "2025-11-12", "duracion": 15, "completada":
False},
    {"fecha": "2025-11-13", "duracion": 25, "completada":
True},
    {"fecha": "2025-11-11", "duracion": 30, "completada":
True}
]

# Ordenar por fecha
por_fecha = sorted(sesiones, key=lambda s: s["fecha"])

# Ordenar por duración descendente
por_duracion = sorted(sesiones, key=lambda s: s["duracion"],
reverse=True)

# Ordenar solo las completadas
completadas = [s for s in sesiones if s["completada"]]
ordenadas_completadas = sorted(completadas, key=lambda s:
s["duracion"])

print("🗓️ Ordenadas por fecha:")
for s in por_fecha:
    print(s)

print("\n🔥 Completadas ordenadas por duración:")
for s in ordenadas_completadas:
    print(s)
```

💾 Puedes luego guardar estos resultados en tu archivo JSON (`json.dump()`), o mostrarlos en tu CLI.

## 🧠 Consejos finales del nivel 5.1 (ampliado)

1. `sorted()` es universal — funciona con listas, tuplas, sets y diccionarios.
2. Usa `key=lambda` para definir el criterio de ordenación.
3. Usa `reverse=True` para mostrar los resultados más grandes o recientes primero.
4. Aprende a combinar `sorted()` con `filter()` o list comprehensions para ordenar solo los datos que te interesan.
5. Este método será clave tanto en tu Pomodoro (ordenar historial o estadísticas) como en tu proyecto API (ordenar resultados recibidos).