Tema 5:

Fundamentos de la sintaxis en Python (4 h)

5. Fundamentos de la sintaxis en Python

5.1. Reglas de identación y estilo

5.2. Nombres de variables, constantes y convenciones (PEP 8)

5.3. Tipos de datos básicos: números, cadenas, booleanos

5.4. Conversión de tipos (casting)

5.5. Operadores aritméticos, lógicos y de comparación

5.6. Entrada y salida estándar (input(), print())

5.7. Comentarios y documentación en el código

5.8. Parte práctica (ejemplos y ejercicios)

5.9. Distribución sugerida del tiempo

Profesor: Salvador Martínez Bolinches

Centro: IES Font de Sant Lluís

Año: 2025

Objetivos de aprendizaje

- Comprender la sintaxis básica de Python.
- Identificar y usar correctamente variables, identificadores y palabras reservadas.
- Manejar la indentación como parte de la sintaxis.
- Diferenciar entre expresiones, sentencias e instrucciones simples.
- Reconocer y aplicar las convenciones de estilo (PEP 8).

5.1. Reglas de indentación y estilo

En Python, la **sangría (indentación)** no es decorativa, sino parte de la sintaxis. Sustituye a las llaves `{}` de C/Java o a las palabras clave como `begin/end` en Pascal. Cada bloque debe estar indentado con **4 espacios** o **un tabulador**, pero NUNCA se debe usar tabuladores mezclados con espacios).

Ejemplo válido:

```
if True:
    print("Esto está indentado")
    print("Pertenece al bloque")
```

Ejemplo con error:

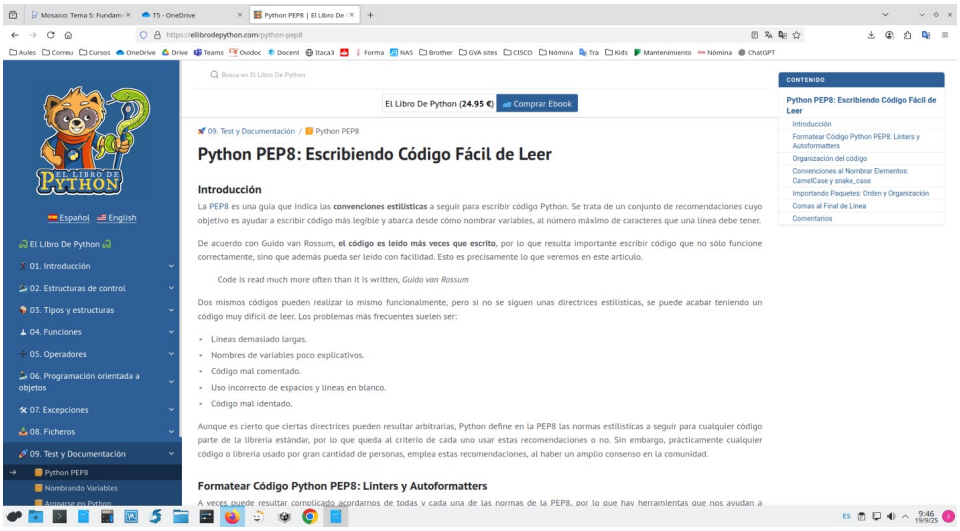
```
if True:
print("Error de indentación")    # SyntaxError
```

La **PEP8** es una guía oficial que indica las **convenciones estilísticas** a seguir para escribir código Python. Se trata de un conjunto de recomendaciones cuyo objetivo es ayudar a escribir código más legible y abarca desde cómo nombrar variables, al número máximo de caracteres que una línea debe tener.

De acuerdo con Guido van Rossum, **el código es leído más veces que escrito**, por lo que resulta importante escribir código que no sólo funcione correctamente, sino que además pueda ser leído con facilidad.

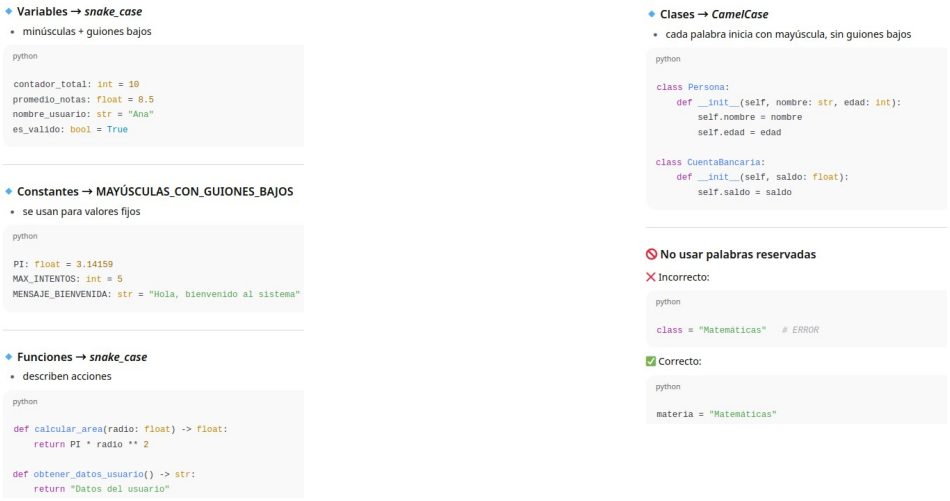
Dos mismos códigos pueden realizar lo mismo funcionalmente, pero si no se siguen unas directrices estilísticas, se puede acabar teniendo un código muy difícil de leer. Los problemas más frecuentes suelen ser:

- Líneas demasiado largas.
- Nombres de variables poco explicativos.
- Código mal comentado.
- Uso incorrecto de espacios y líneas en blanco.
- Código mal identado.



5.2. Nombres de variables, constantes y convenciones (PEP 8)

- Variables → `snake_case`
- Constantes → mayúsculas con guiones bajos: `PI = 3.14159`
- Funciones → `snake_case`
- Clases → `CamelCase`
- No usar palabras reservadas (`if`, `class`, `import`, etc.).
- Otros lenguajes, como C#, utilizan `pascalCase`, pero en Python no se usa.



En Python, la **guía de estilo PEP 8** propone una serie de convenciones para nombrar variables, constantes, funciones y clases, con el objetivo de mejorar la claridad y uniformidad del código.

Las **variables** deben escribirse en *snake_case*, es decir, con todas las letras en minúsculas y separando las palabras mediante guiones bajos, por ejemplo: `mi_variable`, `contador_total`. Esto facilita la lectura cuando los nombres son compuestos.

Las **constantes**, aunque Python no tiene constantes verdaderas, se definen siguiendo la convención de escribirlas en mayúsculas con guiones bajos, como `PI = 3.14159` o `MAX_INTENTOS = 5`. Esta forma indica al programador que esos valores no deberían modificarse durante la ejecución.

Las **funciones** también emplean el estilo *snake_case*. Se recomienda que sus nombres describan claramente la acción que realizan, como `calcular_area()` o `obtener_datos()`.

En cuanto a las **clases**, se utiliza el estilo *CamelCase*, donde cada palabra comienza con mayúscula y no se emplean guiones bajos. Ejemplos: `Persona`, `CuentaBancaria`.

Finalmente, es fundamental **evitar el uso de palabras reservadas** de Python —como `if`, `class`, `import`, `while`, entre otras— para nombrar variables, funciones o clases, ya que forman parte de la sintaxis del lenguaje y su uso incorrecto puede causar errores o confusión en el código.

En conjunto, seguir estas convenciones no solo hace el código más fácil de entender, sino que también favorece el trabajo en equipo y la coherencia en proyectos de mayor escala.

5.3. Tipos de datos básicos

Números

```
entero = 10
flotante = 3.14
complejo = 2 + 3j
```

Cadenas de texto

```
nombre = "Ana"
saludo = 'Hola'
frase = """Texto
en varias líneas"""
```

Booleanos

```
es_activo = True
es_admin = False
```

En Python, los **tipos básicos de datos** constituyen la base para la representación y manipulación de información dentro de un programa. Entre los más utilizados se encuentran:

- **Enteros (*int*)**: representan números enteros, positivos o negativos, sin parte decimal (ejemplo: `10`, `-3`). A diferencia de otros lenguajes, en Python los enteros no tienen un límite fijo de tamaño; pueden crecer tanto como lo permita la memoria disponible.
- **Números de punto flotante (*float*)**: se emplean para valores numéricos con decimales o en notación científica (ejemplo: `3.14`, `2.5e3`). Los flotantes en Python siguen el estándar IEEE 754 de doble precisión, lo que puede producir pequeñas imprecisiones en operaciones aritméticas (por ejemplo, `0.1 + 0.2` puede dar `0.30000000000000004`).
- **Números irracionales o complejos (complex):** No se utilizan mucho. Constan de dos partes: 2 + 5j.
- **Cadenas de texto (*str*)**: corresponden a secuencias de caracteres delimitadas por comillas simples o dobles, utilizadas para almacenar y procesar información textual (ejemplo: `"Hola"`, `'Python'`). Las cadenas en Python son **inmutables**, lo que significa que no pueden modificarse directamente; cualquier operación sobre ellas genera una nueva cadena.
- **Booleanos (*bool*)**: expresan valores lógicos, con únicamente dos posibles estados: `True` o `False`, fundamentales en estructuras de control y lógica condicional. En operaciones numéricas, `True` se interpreta como `1` y `False` como `0, 0.X, {}, [], ()`. Existen dos funciones ***all(valores)*** y ***any(valores)***, que devuelven `True` si todos los valores son `True` o si alguno es `True`, respectivamente.

Estos tipos de datos son considerados **primitivos** y permiten al programador construir estructuras más complejas, definir comportamientos y resolver problemas mediante el uso del lenguaje.

En resumen, los tipos de datos básicos en Python no solo permiten almacenar información elemental, sino que también posibilitan la construcción de expresiones, estructuras y algoritmos más complejos. Su flexibilidad e integración con el sistema de tipos dinámicos del lenguaje convierten a Python en una herramienta especialmente versátil para la programación.

Existen multitud de fiunciones en Python para trabajar con números:

abs(num) → valor absoluto de num ***divmod(num1, num2) → tupla (cociente, resto)***
hex(num) → hexadecimal ***bin(num) → binario***
oct(num) → octal ***pow(num, potencia) → eleva num a la potencia***
round(num, decimales) → redondea

También hay operadores a nivel de bit:

| → or ***^ → orex*** ***& → and*** ***~ → invierte bits***
num»2 → desplaza 2 bits a la derecha ***num«2 → desplaza 2 bits a la izquierda***

5.4. Conversión de tipos (*casting*)

En Python, la **conversión de tipos de datos**, también denominada *type casting*, consiste en transformar un valor de un tipo determinado a otro compatible, con el fin de permitir su manipulación en contextos donde se requiera un formato específico. Este proceso resulta fundamental en un lenguaje de tipado dinámico como Python, ya que, si bien las variables no requieren declarar de antemano su tipo, en muchas ocasiones es necesario asegurar la coherencia entre los datos empleados en operaciones matemáticas, comparaciones o estructuras de control.

Existen dos modalidades principales de conversión: **explícita** e **implícita**.

La **conversión explícita** es aquella en la que el programador indica de manera intencional el tipo de dato al que desea transformar un valor, utilizando funciones predefinidas como `int()`, `float()`, `str()` o `bool()`.

Por ejemplo:

```
numero = int("25")       # convierte la cadena "25" en un entero
decimal = float(10)      # convierte el entero 10 en un número de punto flotante
texto = str(3.14)        # convierte el flotante 3.14 en la cadena "3.14"
```

En este caso, el éxito de la conversión depende de la compatibilidad entre los tipos; intentar convertir una cadena no numérica a entero, por ejemplo `int("hola")`, generará un error.

Por otro lado, la **conversión implícita** ocurre de manera automática cuando Python adapta los tipos involucrados en una operación para evitar pérdida de información. Por ejemplo, al sumar un número entero y uno flotante, el resultado será promovido a flotante:

Este mecanismo, conocido como *type coercion*, está diseñado para preservar la precisión de los cálculos y minimizar inconsistencias.

```
resultado = 5 + 3.0   # el resultado es 8.0 (float)
```

En síntesis, la conversión de tipos en Python es un recurso esencial para garantizar la correcta manipulación de datos y la compatibilidad entre operaciones. El dominio de estas técnicas permite al programador escribir código más robusto, claro y adaptable a diferentes contextos de aplicación.

```
x = "123"
y = int(x)    # 123
z = float(x)  # 123.0
```

5.5. Operadores aritméticos, lógicos y de comparación

- **Aritméticos**: `+`, `-`, `*`, `/`, `//` (división entera), %, `**` (potencia).
- **Comparación**: ==, `!=`, <, <=, >, >=.
- **Lógicos**: `and`, `or`, `not`.

```
print(5 // 2)   # 2
print(5 ** 2)   # 25
print(True and False)  # False
```

En Python, los **operadores** son símbolos especiales que permiten realizar operaciones sobre uno o más operandos. Entre los más relevantes en la práctica cotidiana se encuentran los **aritméticos**, los **de comparación** y los **lógicos**, cada uno con una finalidad distinta.

1. **Operadores aritméticos**
   Se emplean para efectuar cálculos matemáticos sobre valores numéricos. Incluyen la suma (+), resta (-), multiplicación (*), división (/), división entera (//), módulo (%) y exponenciación (**). Estos operadores permiten realizar desde operaciones simples hasta cálculos más avanzados.

2. **Operadores de comparación**
   Sirven para establecer relaciones entre dos valores, devolviendo un resultado booleano (`True` o `False`). Los más comunes son: igual a (==), distinto de (`!=`), mayor que (>), menor que (<), mayor o igual que (>=) y menor o igual que (<=). Son esenciales en estructuras de control, ya que determinan el flujo del programa.

3. **Operadores lógicos**
   Operan sobre valores booleanos y permiten construir expresiones lógicas más complejas. Python incluye tres: `and` (conjunción lógica, verdadero si ambas condiciones lo son), `or` (disyunción lógica, verdadero si al menos una condición lo es) y `not` (negación lógica, invierte el valor de verdad de la expresión). Estos operadores resultan imprescindibles en decisiones múltiples y validación de condiciones.

```
# Operadores aritméticos
a = 10
b = 3
print(a + b)   # 13
print(a - b)   # 7
print(a * b)   # 30
print(a / b)   # 3.333...
print(a // b)  # 3 (división entera)
print(a % b)   # 1 (resto de la división)
print(a ** b)  # 1000 (10 elevado a 3)

# Operadores de comparación
print(a == b)   # False
print(a != b)   # True
print(a > b)    # True
print(a < b)    # False
print(a >= 10)  # True
print(b <= 3)   # True

# Operadores lógicos
x = True
y = False
print(x and y)  # False (ambos deben ser True)
print(x or y)   # True (al menos uno True)
print(not x)    # False (invierte el valor de x)
```

En conjunto, estos tres grupos de operadores constituyen la base de la programación en Python, al facilitar el cálculo, la comparación y la toma de decisiones dentro de un programa.

5.6. Entrada y salida estándar

Salida

```
print("Hola", "mundo", sep="-", end="!!!\n")
```

Entrada

```
nombre = input("Introduce tu nombre: ")
print("Bienvenido,", nombre)
```

En Python, la **entrada y salida de datos** constituyen mecanismos fundamentales para la interacción entre el programa y el usuario. Entre las funciones integradas más empleadas destacan `print()` y `input()`.

1. **Salida de datos: `print()`**
   La función `print()` permite mostrar información en la consola o terminal. Acepta uno o varios argumentos y los convierte a cadenas de texto para su visualización. Además, admite parámetros opcionales como `sep`, que define el separador entre los elementos (por defecto un espacio), y `end`, que especifica el carácter final de la salida (por defecto un salto de línea). Esta función resulta esencial tanto para la comunicación con el usuario como para depuración de programas.
2. **Entrada de datos: `input()`**
   La función `input()` posibilita la captura de información escrita por el usuario en la consola. Devuelve siempre el valor ingresado como una cadena de texto (`str`), independientemente del tipo de dato esperado. Por esta razón, cuando se requiera trabajar con enteros, flotantes u otros tipos, será necesario realizar una conversión explícita (*casting*), por ejemplo mediante `int()`, `float()`, etc.

En conjunto, `print()` e `input()` proporcionan un canal básico pero indispensable para la interacción humano-máquina en programas desarrollados con Python.

```
# --- Salida de datos con print ---
print("Hola, mundo")
print("Nombre:", "Ana", "Edad:", 25)         # Uso de múltiples argumentos
print("Python", "Java", "C++", sep=" | ")    # Cambio de separador
print("Texto sin salto de línea", end=" ")
print("continuación en la misma línea")

# --- Entrada de datos con input ---
nombre = input("Ingrese su nombre: ")        # siempre devuelve str
print("Hola,", nombre)

edad = int(input("Ingrese su edad: "))       # conversión explícita a entero
print("El próximo año tendrás", edad + 1)

altura = float(input("Ingrese su altura en metros: "))  # conversión a float
print("Tu altura es:", altura, "m")
```

Formatear la salida

Hay dos estilos para formatear la salida. El **estilo viejo o tradicional**, y el estilo más moderno. El estilo tradicional utiliza el símbolo % seguido de un carácter para indicar el tipo de dato a mostrar:

%d → dec ***%f*** → float ***%s*** → str ***%o*** → oct ***%b*** → bin ***%x*** → hex **…**

Además, puede llevar otros caracteres:

%.2f → redondea el float a dos decimales

Podemos cambiar el carácter que separa los diferentes elementos mostrados y el carácter final.

print(1,2,3,4,5, sep="-", end=".") → 1-2-3-4-5.

Hay funciones que convierten datos a otro tipo de datos: ***bin(), hex(), oct(), int(), str(), …***

Por otro lado, el **estilo moderno** usa la función ***format*** para formatear los datos:

format(numero, "X")

Esta función formatea ***numero*** de modo que, según valga "X":

"o" → oct ***"b"*** → bin ***"x"*** → hex **…**

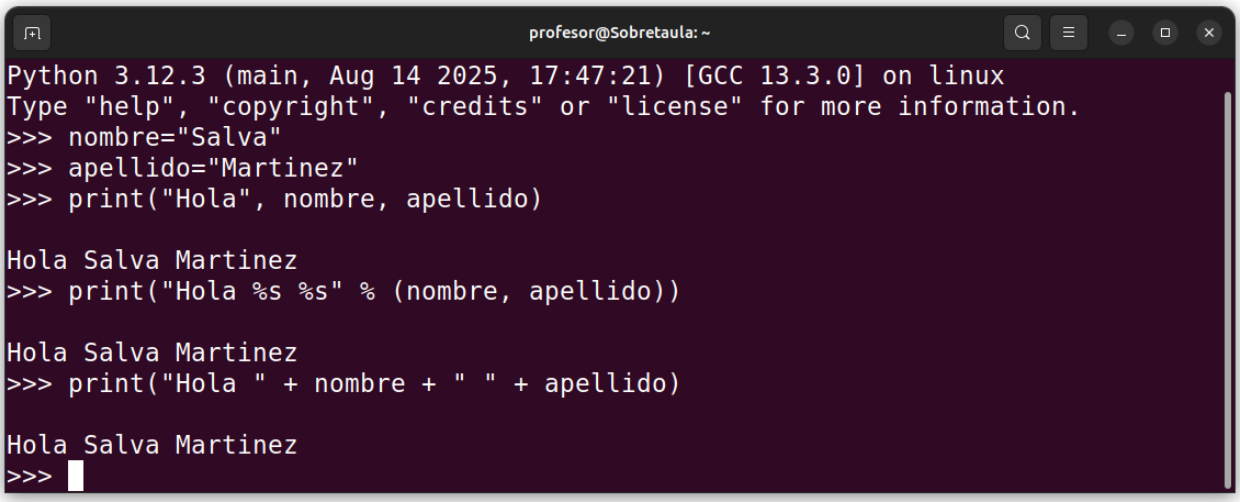
Recordemos que tenemos tres formas de mostrar cadenas de textos:

print("Hola", nombre, apellido)

print("Hola %s %s" % (nombre, apellido))

print("Hola " + nombre + " " + apellido)

Las dos primeras incluyen un espacio entre los elementos. En la tercera, los debes de indicar explícitamente.



5.7. Comentarios y documentación en el código

En Python, los **comentarios** constituyen anotaciones dentro del código fuente que no son interpretadas ni ejecutadas por el intérprete. Su función principal es **documentar** el programa, aumentando su legibilidad, facilitando la comprensión de la lógica empleada y promoviendo el mantenimiento colaborativo del software. Existen dos formas principales de comentarios: los **comentarios en línea** y los **docstrings**.

1. **Comentarios en línea**
   Se definen utilizando el símbolo de almohadilla #. Todo el texto que sigue a este carácter en la misma línea es ignorado por el intérprete. Se utilizan habitualmente para explicar fragmentos específicos de código, resaltar decisiones de diseño o desactivar temporalmente una instrucción.

2. **Cadenas de documentación (*docstrings*)**

   Los *docstrings* (abreviatura de *documentation strings*) constituyen un mecanismo formal para documentar módulos, clases, métodos y funciones. Se delimitan con comillas triples (`""" ... """` o `''' ... '''`) y suelen colocarse inmediatamente después de la definición de un objeto. A diferencia de los comentarios en línea, los docstrings pueden recuperarse en tiempo de ejecución mediante la función integrada `help()` o el atributo `.__doc__`, lo cual los convierte en parte de la documentación oficial del programa o librería.

```python
# --- Comentarios en línea ---
# Este es un comentario en línea
x = 10  # Se asigna el valor 10 a la variable x

# print("Esto no se ejecuta")  # Comentario que desactiva código

# --- Docstrings ---
def suma(a, b):
    """
    Función que calcula la suma de dos números.

    Parámetros:
        a (int o float): primer número
        b (int o float): segundo número

    Retorna:
        int o float: la suma de a y b
    """
    return a + b

class Persona:
    """
    Clase que representa a una persona.

    Atributos:
        nombre (str): nombre de la persona
        edad (int): edad de la persona
    """
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

En conjunto, los comentarios en línea y los docstrings son herramientas esenciales para garantizar la calidad del software, no solo desde una perspectiva técnica, sino también desde el punto de vista de la comunicación y la colaboración entre programadores.

5.8. Parte práctica (ejemplos y ejercicios)

Ejemplo 1: Conversión de tipos

```
edad = input("Introduce tu edad: ")
edad = int(edad)
print("El año que viene tendrás", edad + 1)
```

```
# Solicitar al usuario dos números y sumarlos
num1 = int(input("Ingrese el primer número: "))
num2 = float(input("Ingrese el segundo número: "))

suma = num1 + num2
print("La suma es:", suma)
```

Ejemplo 2: Uso de operadores

```
a, b = 7, 3
print("División entera:", a // b)
print("Módulo:", a % b)
print("Potencia:", a ** b)
```

```
a = 15
b = 4

# Operadores aritméticos
print("Suma:", a + b)          # 19
print("División entera:", a // b)  # 3
print("Módulo:", a % b)        # 3

# Operadores de comparación
print("¿a es mayor que b?", a > b)   # True
print("¿a es igual a b?", a == b)   # False

# Operadores lógicos
print("Resultado lógico:", (a > 10) and (b < 5))  # True
```

Ejemplo 3: Documentación de funciones

```
def area_circulo(radio):
    """Calcula el área de un círculo dado su radio."""
    import math
    return math.pi * radio**2

print(area_circulo(5))
```

```
def calcular_area_triangulo(base: float, altura: float) -> float:
    """
    Calcula el área de un triángulo a partir de su base y altura.

    Parámetros:
        base (float): longitud de la base del triángulo
        altura (float): longitud de la altura correspondiente

    Retorna:
        float: área del triángulo calculada con la fórmula (base * altura) / 2
    """
    return (base * altura) / 2

# Uso de la función
print("Área del triángulo:", calcular_area_triangulo(10, 5))

# Consultar la documentación en tiempo de ejecución
help(calcular_area_triangulo)
```

Ejercicios propuestos

1. Crea un script que lea dos números introducidos por el usuario y muestre su suma, resta, producto y división real.
2. Pregunta al usuario si tiene carnet de conducir (`s/n`) y si tiene coche (`s/n`). Muestra si puede conducir o no, usando `and` y `or`.
3. Escribe una función `fahrenheit_a_celsius` documentada con docstring.
4. Escribe un programa que:
    - Pida el nombre y edad del usuario.
    - Compruebe si puede entrar a un bar (edad ≥ 18).
    - Muestre un saludo personalizado en mayúsculas.
    - Documente todas las funciones usadas.
5. ¿Se puede usar `class` como nombre de variable en Python? Explica por qué.
6. Marca cuáles de los siguientes nombres son identificadores válidos en Python:
    - `usuario_1`
    - `1usuario`
    - `usuario-nuevo`
    - `_usuario`
7. Escribe un `print` que muestre a la vez un número y un texto.
8. ¿Qué diferencia hay entre una expresión (`2 + 3`) y una sentencia (`x = 2 + 3`)?
9. ¿Qué PEP establece la guía de estilo oficial de Python?
10. Explica con un ejemplo en código la diferencia entre el operador de asignación `=` y el operador de comparación `==`.

5.9. Distribución sugerida del tiempo (4 h)

- **Teoría (1 h)** → reglas sintácticas, tipos de datos, operadores, PEP 8.
- **Ejemplos guiados (0,5 h)** → conversión de tipos, entrada/salida, docstrings.
- **Ejercicios propuestos (2,5 h)** → trabajo individual y en parejas, ejercicios integradores.