

PYTHON

DE PRINCIPIANTE A AVANZADO



LUIS MINGUILLON PASCUAL

Python

de principiante a avanzado

Luis Minguillón Pascual

COPYRIGHT

La licencia de este libro electrónico es para uso personal. Por lo tanto no puedes revenderlo a otras personas.

Gracias por respetar los derechos de autor.

©Luis Minguillón Pascual
06/08/2025

Prefacio

Vivimos en una época en la que la informática se ha convertido en parte esencial de nuestra vida cotidiana. Desde los dispositivos que usamos a diario hasta las herramientas que impulsan la ciencia, la medicina, el arte y la comunicación, la informática está presente en todos los ámbitos. Saber programar no solo abre puertas en el mundo laboral, donde cada vez se valoran más estas competencias, sino que también brinda un espacio para la creatividad, permitiendo a cada persona convertir sus ideas en proyectos reales. Programar es, en definitiva, aprender a construir el futuro.

He elegido Python como lenguaje de base para este libro porque considero que es una de las mejores opciones tanto para quienes se inician en la programación como para quienes desean profundizar sus conocimientos. Python es un lenguaje sencillo, claro y versátil, pero al mismo tiempo es poderoso y se utiliza en áreas tan diversas como el desarrollo web, la inteligencia artificial, la automatización, la ciencia de datos y la educación. Hoy en día, Python se ha convertido en uno de los lenguajes de programación más importantes y populares del mundo.

Este libro ha sido creado pensando en dos tipos de lectores: quienes no tienen ningún conocimiento de Python y quieren comenzar desde cero, y aquellos que ya tienen experiencia y buscan avanzar hacia un nivel más profundo. Mi intención es guiar al lector paso a paso, ofreciendo explicaciones claras y ejemplos prácticos, de modo que pueda comprender no solo la teoría, sino también cómo aplicarla en la práctica.

Antes de terminar, no quiero despedirme sin expresar mi más sincero agradecimiento a mi mujer, Marisa. Su apoyo incondicional, su paciencia y su confianza han sido fundamentales para que este libro pudiera hacerse realidad. Sin ella, este proyecto no hubiese sido posible.

Luis Minguillón Pascual

Índice :

Lección 1: Variables y Operadores en Python

- **Variables:** Concepto, Reglas para nombrar, Tipos
- **Operadores:** Aritméticos, Asignación, Comparación, Lógicos, Identidad y Pertenencia
- Buenas prácticas con variables
- Ejercicios (Inicio, Medio, Avanzado)

Lección 2: Estructuras Condicionales en Python

- **Concepto de Condición**
- **Estructuras:** `if`, `if-else`, `if-elif-else`
- **Operadores en Condiciones:** Comparación, Lógicos
- Condiciones Anidadas y Operador Ternario
- Buenas Prácticas en Condicionales y Ejemplos
- Ejercicios (Principiante, Medio, Avanzado)

Lección 3: Estructuras Repetitivas (Bucles)

- **Introducción** y Tipos de bucles (`for`, `while`, `do while` simulado)
- Componentes de un bucle y Bucles infinitos
- Instrucciones especiales (`break`, `continue`, `pass`)
- Anidación de bucles y Ejemplos prácticos
- Conclusión
- Ejercicios (Principiante, Medio, Avanzado)

Lección 4: Estructuras de Datos – Listas en Python

- **Introducción a las listas** y Creación de listas
- Acceso, Modificación y Recorrido de elementos
- Operaciones básicas y Métodos principales
- Slicing (rebanado), Listas anidadas (matrices)
- List Comprehension y Copias de listas
- Conclusiones
- Ejercicios (Principiante, Medio, Avanzado)

Lección 5: Estructuras de Datos en Python – Conjuntos

- **Introducción a los Conjuntos** y Creación de Conjuntos
- Operaciones básicas (añadir, eliminar, verificar, longitud)
- Operaciones matemáticas (Unión, Intersección, Diferencia, Simétrica)
- Métodos útiles y Conjuntos inmutables (`frozenset`)
- Aplicaciones prácticas y Buenas prácticas
- Ejercicios prácticos (Principiante, Medio, Avanzado)

Lección 6: Estructuras de Datos – Tuplas en Python

- **Introducción a las Tuplas** y Creación (vacía, un elemento, múltiples)
- Acceso a elementos y Slicing

- Operaciones básicas (concatenación, repetición, pertenencia)
- Funciones y métodos útiles (`len`, `max`, `min`, `sum`, `index`, `count`)
- Inmutabilidad y conversiones, Tuplas anidadas
- Tuplas vs Listas, Desempaquetado
- Aplicaciones prácticas y Buenas prácticas
- Resumen rápido
- Ejercicios (Principiante, Medio, Avanzado)

Lección 7: Diccionarios en Python

- **Introducción a los Diccionarios** y Creación (vacío, con elementos, con claves de diferentes tipos)
- Acceder a los valores (`[ ]`, `get()`)
- Modificar Diccionarios (agregar, modificar, eliminar)
- Métodos útiles (`keys`, `values`, `items`, `update`, `pop`, `popitem`, `copy`)
- Iterar sobre Diccionarios
- Diccionarios anidados y Comprensión de Diccionarios
- Buenas prácticas
- Ejercicios (Principiante, Medio, Avanzado)

Lección 8: Funciones en Python

- **Introducción a las funciones**, Definir y llamar funciones (sin parámetros, con parámetros, con retorno)
- **Tipos de parámetros:** Posición, Nombre, Por defecto, Argumentos variables (`*args`), Argumentos con palabras clave (`**kwargs`)
- Funciones anidadas, Funciones lambda (anónimas), Funciones recursivas
- Documentación de funciones (`docstring`)
- Buenas prácticas y Ejemplos prácticos
- Ejercicios (Principiante, Medio, Avanzado)

Lección 9: Programación Orientada a Objetos (POO) en Python

- **Introducción a la POO** y Conceptos Clave (Clase, Objeto, Atributo, Método, Encapsulamiento, Herencia, Polimorfismo, Abstracción)
- Creación de Clases y Objetos
- Encapsulamiento
- Herencia
- Métodos y Atributos Especiales (`__str__`, `__repr__`, `__del__`)
- Composición de Objetos
- Polimorfismo
- Clases y Métodos de Clase (`@classmethod`, `@staticmethod`)
- Buenas Prácticas en POO
- Ejercicio Completo de Aplicación (Sistema de Biblioteca)
- Ejercicios (Principiante, Medio, Avanzado)

Lección 10: Tratamiento de ficheros CSV en Python

- **Introducción a CSV** (formato, ventajas, limitaciones)

- **Módulo csv en Python:** Lectura (`reader`, `DictReader`)
- **Escritura de CSV:** Usando `writer` y `DictWriter`
- Configuración de delimitadores y formatos
- **Manipulación de datos CSV:** Filtrar, Modificar y escribir, Ordenar
- Buenas prácticas al tratar CSV
- CSV con `pandas` (opcional)
- Ejercicios (Principiante, Medio, Avanzado)

Lección 1.: Variables y Operadores en Python

Python es un lenguaje de programación muy flexible y fácil de aprender. Comprender **variables** y **operadores** es fundamental, porque son la base para almacenar y manipular información en cualquier programa.

1. Variables en Python

1.1 ¿Qué es una variable?

Una **variable** es un espacio en memoria que se utiliza para almacenar información. Puedes pensar en ella como una “caja” donde guardas un valor y le das un nombre.

```
edad = 25
nombre = "Ana"
es_estudiante = True
```

- `edad` almacena un número entero.
- `nombre` almacena un texto (cadena de caracteres o *string*).
- `es_estudiante` almacena un valor booleano (`True` o `False`).

1.2 Reglas para nombrar variables

1. Deben comenzar con una letra o un guion bajo `_`.
2. No pueden empezar con un número.
3. Solo pueden contener letras, números y guiones bajos.
4. No se pueden usar palabras reservadas de Python (`if`, `for`, `class`, etc.).
5. Es recomendable usar nombres descriptivos.

```
# Correcto
mi_edad = 25
nombre_completo = "Carlos"
```

```
# Incorrecto
2edad = 25
nombre-completo = "Carlos"
```

1.3 Tipos de variables

En Python, no es necesario declarar el tipo de variable; Python lo infiere automáticamente. Sin embargo, los tipos principales son:

- **Enteros (int):** números sin decimales.
- **Flotantes (float):** números con decimales.
- **Cadenas de texto (str):** texto.
- **Booleanos (bool):** True o False.

```
x = 10          # int
y = 3.14        # float
nombre = "Ana"  # str
activo = True   # bool
```

2. Operadores en Python

Los operadores se utilizan para realizar operaciones con los valores de las variables.

2.1 Operadores aritméticos

Sirven para operaciones matemáticas:

Operador	Descripción	Ejemplo
+	Suma	a + b
-	Resta	a - b
*	Multiplicación	a * b
/	División	a / b
//	División entera	a // b
%	Módulo (resto)	a % b
**	Potencia	a ** b

Ejemplos:

```
a = 10
b = 3

print(a + b) # 13
print(a - b) # 7
print(a * b) # 30
print(a / b) # 3.3333333333333335
print(a // b) # 3
print(a % b) # 1
print(a ** b) # 1000
```

2.2 Operadores de asignación

Se usan para asignar valores a variables:

Operador	Ejemplo	Descripción
=	x = 5	Asignación simple
+=	x += 2	Suma y asigna (x = x + 2)
-=	x -= 2	Resta y asigna
*=	x *= 2	Multiplica y asigna
/=	x /= 2	Divide y asigna
%=	x %= 2	Módulo y asigna
**=	x **= 2	Potencia y asigna
//=	x //= 2	División entera y asigna

Ejemplo:

```
x = 10
x += 5    # Equivale a x = x + 5
print(x) # 15
x *= 2
print(x) # 30
```

2.3 Operadores de comparación

Se utilizan para comparar valores, y devuelven True o False.

Operador	Descripción	Ejemplo
==	Igual a	a == b
!=	Distinto de	a != b
>	Mayor que	a > b
<	Menor que	a < b
>=	Mayor o igual que	a >= b
<=	Menor o igual que	a <= b

Ejemplo:

```
a = 5
b = 10

print(a == b) # False
print(a != b) # True
print(a < b)  # True
print(a >= b) # False
```

2.4 Operadores lógicos

Se utilizan para combinar condiciones:

Operador	Descripción	Ejemplo
and	Y lógico	<code>a > 0 and b > 0</code>
or	O lógico	<code>a > 0 or b > 0</code>
not	Negación	<code>not(a > 0)</code>

Ejemplo:

```
a = 5
b = 10

print(a > 0 and b > 0) # True
print(a > 0 or b < 0) # True
print(not(a > 0))      # False
```

2.5 Operadores de identidad y pertenencia

- **Identidad:** comprueba si dos variables son el mismo objeto.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y) # True
print(x is z) # False
print(x is not z) # True
```

- **Pertenencia:** comprueba si un valor está dentro de una secuencia (lista, tupla, string).

```
lista = [1, 2, 3, 4]

print(2 in lista) # True
print(5 not in lista) # True
```

3. Buenas prácticas con variables

1. Usa nombres descriptivos: `edad` es mejor que `x`.
 2. Evita nombres de una sola letra, salvo en bucles (`i`, `j`).
 3. Usa `snake_case` (`nombre_variable`) para nombres de variables.
 4. Inicializa variables antes de usarlas.
 5. Mantén un código limpio y legible.
-

4. Ejercicios

4.1 Ejercicios de Python (Nivel Inicio) – Variables y Operadores

Ejercicio 1: Suma de dos números

```
a = 5
b = 3
suma = a + b
print(suma)
```

Explicación: Creamos dos variables `a` y `b`, usamos el operador `+` para sumarlas y guardamos el resultado en `suma`.

Ejercicio 2: Resta de dos números

```
x = 10
y = 4
resta = x - y
print(resta)
```

Explicación: Usamos el operador `-` para restar `y` de `x`.

Ejercicio 3: Multiplicación

```
num1 = 6
num2 = 7
producto = num1 * num2
print(producto)
```

Explicación: Multiplicamos `num1` por `num2` usando `*`.

Ejercicio 4: División

```
a = 20
b = 5
division = a / b
print(division)
```

Explicación: La división `/` devuelve un número flotante.

Ejercicio 5: División entera y módulo

```
a = 17
b = 3
div_entera = a // b
resto = a % b
print(div_entera, resto)
```

Explicación: // devuelve la parte entera de la división y % devuelve el resto.

Ejercicio 6: Potencia

```
base = 2
exponente = 3
resultado = base ** exponente
print(resultado)
```

Explicación: ** eleva la base a la exponente.

Ejercicio 7: Asignación múltiple

```
x, y, z = 5, 10, 15
print(x, y, z)
```

Explicación: Podemos asignar valores a varias variables en una sola línea.

Ejercicio 8: Operadores de asignación

```
num = 10
num += 5
num -= 3
num *= 2
print(num)
```

Explicación: +=, -=, *= modifican el valor de num de forma acumulativa.

Ejercicio 9: Intercambio de valores

```
a = 5
b = 10
temp = a
a = b
b = temp
print(a, b)
```

Explicación: Usamos una variable temporal para intercambiar los valores.

Ejercicio 10: Concatenación de strings

```
nombre = "Ana"
apellido = "Pérez"
nombre_completo = nombre + " " + apellido
print(nombre_completo)
```

Explicación: + se usa para unir cadenas de texto.

Ejercicio 11: Comprobación de igualdad

```
a = 5
b = 5
resultado = a == b
print(resultado)
```

Explicación: == compara los valores y devuelve True si son iguales.

Ejercicio 12: Mayor o menor

```
x = 7
y = 10
print(x > y)  # False
print(x < y)  # True
```

Explicación: > y < comparan valores.

Ejercicio 13: Mayor o igual y menor o igual

```
a = 8
b = 8
print(a >= b)  # True
print(a <= b)  # True
```

Explicación: >= y <= incluyen la igualdad en la comparación.

Ejercicio 14: Negación booleana

```
activo = True  
print(not activo)
```

Explicación: `not` invierte el valor booleano.

Ejercicio 15: Operadores lógicos AND/OR

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

Explicación: `and` devuelve True solo si ambos son True; `or` devuelve True si alguno es True.

Ejercicio 16: Verificar pertenencia en lista

```
numeros = [1, 2, 3, 4, 5]  
print(3 in numeros)
```

Explicación: `in` devuelve True si el elemento está dentro de la lista.

Ejercicio 17: Verificar no pertenencia en lista

```
letras = ["a", "b", "c"]  
print("d" not in letras)
```

Explicación: `not in` devuelve True si el elemento no está en la lista.

Ejercicio 18: Identidad de variables

```
lista1 = [1, 2, 3]  
lista2 = lista1  
lista3 = [1, 2, 3]  
print(lista1 is lista2)  # True  
print(lista1 is lista3)  # False
```

Explicación: `is` verifica si las variables apuntan al mismo objeto en memoria.

Ejercicio 19: Área de un rectángulo

```
ancho = 5
alto = 10
area = ancho * alto
print(area)
```

Explicación: Multiplicamos ancho por alto para obtener el área.

Ejercicio 20: Perímetro de un rectángulo

```
ancho = 5
alto = 10
perimetro = 2 * (ancho + alto)
print(perimetro)
```

Explicación: El perímetro se calcula sumando los lados y multiplicando por 2.

4.2 Ejercicios de Python (Nivel Medio) – Variables y Operadores

Ejercicio 1: Promedio de tres números

```
a = 7
b = 8
c = 10
promedio = (a + b + c) / 3
print(promedio)
```

Explicación: Sumamos los tres números y dividimos entre 3 para obtener el promedio.

Ejercicio 2: Conversión de grados Celsius a Fahrenheit

```
celsius = 25
fahrenheit = (celsius * 9/5) + 32
print(fahrenheit)
```

Explicación: Usamos la fórmula de conversión: $F = C \times 9/5 + 32$.

Ejercicio 3: Conversión de metros a kilómetros

```
metros = 1500
kilometros = metros / 1000
print(kilometros)
```

Explicación: 1 km = 1000 metros, por lo que dividimos metros entre 1000.

Ejercicio 4: Suma de números pares en un rango

```
suma = 0
for i in range(1, 11):
    if i % 2 == 0:
        suma += i
print(suma)
```

Explicación: Usamos un bucle `for` y el operador `%` para detectar pares y sumarlos.

Ejercicio 5: Comprobar si un número es divisible por 3 y 5

```
num = 15
resultado = (num % 3 == 0) and (num % 5 == 0)
print(resultado)
```

Explicación: Usamos `and` para combinar condiciones de divisibilidad.

Ejercicio 6: Calcular el área de un círculo

```
import math
radio = 4
area = math.pi * radio ** 2
print(area)
```

Explicación: Área de círculo = $\pi \times \text{radio}^2$, usando `math.pi` para π .

Ejercicio 7: Intercambio de valores sin variable temporal

```
a = 5
b = 10
a, b = b, a
print(a, b)
```

Explicación: Python permite intercambiar valores directamente con la sintaxis `a, b = b, a`.

Ejercicio 8: Incrementar un valor según condición

```
x = 10
if x < 20:
    x += 5
print(x)
```

Explicación: Usamos `+=` solo si la condición se cumple.

Ejercicio 9: Calcular la diferencia absoluta

```
a = 7
b = 12
diferencia = abs(a - b)
print(diferencia)
```

Explicación: La función `abs()` devuelve el valor absoluto de la diferencia.

Ejercicio 10: Operación combinada de operadores

```
x = 5
y = 3
resultado = x**2 + y*2 - (x + y)
print(resultado)
```

Explicación: Python respeta la precedencia de operadores: primero **, luego *, luego + y -.

Ejercicio 11: Verificar si un número está en un rango usando operadores

```
num = 15
en_rango = (num >= 10) and (num <= 20)
print(en_rango)
```

Explicación: Comprobamos que el número esté entre 10 y 20 con and.

Ejercicio 12: Comparación múltiple de números

```
a = 5
b = 10
c = 5
resultado = (a == c) or (a > b)
print(resultado)
```

Explicación: or devuelve True si alguna de las condiciones es True.

Ejercicio 13: Calcular el perímetro de un triángulo

```
lado1 = 5
lado2 = 6
lado3 = 7
perimetro = lado1 + lado2 + lado3
print(perimetro)
```

Explicación: El perímetro es la suma de todos los lados.

Ejercicio 14: Conversión de segundos a horas, minutos y segundos

```
segundos = 3670
horas = segundos // 3600
minutos = (segundos % 3600) // 60
seg_remanente = segundos % 60
print(horas, minutos, seg_remanente)
```

Explicación: Usamos // para división entera y % para restos.

Ejercicio 15: Comprobar si un número es par o impar

```
num = 17
es_par = num % 2 == 0
print(es_par)
```

Explicación: Si el resto al dividir por 2 es 0, el número es par.

Ejercicio 16: Calcular el promedio de notas y verificar aprobado

```
nota1 = 7
nota2 = 8
nota3 = 9
promedio = (nota1 + nota2 + nota3) / 3
aprobado = promedio >= 6
print(promedio, aprobado)
```

Explicación: Combinamos operadores aritméticos con operadores de comparación.

Ejercicio 17: Incrementar una variable según su valor

```
x = 12
x += 10 if x > 10 else 5
print(x)
```

Explicación: Usamos una expresión condicional para modificar x.

Ejercicio 18: Calcular el promedio de edad de un grupo

```
edades = [20, 25, 30, 22, 28]
promedio = sum(edades) / len(edades)
print(promedio)
```

Explicación: sum() suma la lista y len() devuelve el número de elementos.

Ejercicio 19: Verificar si un número es múltiplo de 4 o 6

```
num = 24  
resultado = (num % 4 == 0) or (num % 6 == 0)  
print(resultado)
```

Explicación: or devuelve True si el número es divisible por 4 o 6.

Ejercicio 20: Combinación de operadores aritméticos y lógicos

```
a = 5  
b = 10  
c = 15  
resultado = (a + b > 12) and (c - b == 5)  
print(resultado)
```

Explicación: Combinamos operaciones aritméticas con operadores lógicos and para evaluar condiciones complejas.

4.3 Ejercicios de Python Nivel Avanzado – Variables y Operadores

Ejercicio 1: Intercambio de variables sin usar una tercera variable

```
a = 12
b = 7
a = a + b
b = a - b
a = a - b
print(a, b)
```

Explicación: Se usa aritmética para intercambiar valores sin usar variable temporal.

Ejercicio 2: Comprobar si un número es divisible por 3, 5 y 7

```
num = 105
resultado = (num % 3 == 0) and (num % 5 == 0) and (num % 7 == 0)
print(resultado)
```

Explicación: and combina múltiples condiciones de divisibilidad.

Ejercicio 3: Operador ternario para asignación condicional

```
x = 10
y = 20
mayor = x if x > y else y
print(mayor)
```

Explicación: El operador ternario permite asignar un valor según una condición.

Ejercicio 4: Suma de cuadrados de números pares en una lista

```
numeros = [1, 2, 3, 4, 5, 6]
suma_cuadrados = sum(n**2 for n in numeros if n % 2 == 0)
print(suma_cuadrados)
```

Explicación: Se usan **comprensiones de listas** y operadores para filtrar y elevar al cuadrado.

Ejercicio 5: Verificar si un número está en un rango usando operador combinado

```
num = 15
resultado = 10 <= num <= 20
print(resultado)
```

Explicación: Python permite comparaciones encadenadas para verificar rangos.

Ejercicio 6: Concatenar y repetir strings con operadores

```
s1 = "Hola"
s2 = "Mundo"
resultado = (s1 + " ") * 3 + s2
print(resultado)
```

Explicación: + concatena y * repite strings.

Ejercicio 7: Modificar una lista usando operadores de asignación

```
lista = [1, 2, 3]
lista += [4, 5]
lista *= 2
print(lista)
```

Explicación: += añade elementos y *= repite la lista.

Ejercicio 8: Cálculo del factorial usando operadores

```
n = 5
factorial = 1
for i in range(1, n+1):
    factorial *= i
print(factorial)
```

Explicación: *= acumula el producto de los números del 1 a n.

Ejercicio 9: Comprobar si un número es múltiplo de 2, 3 o 5 usando operadores lógicos

```
num = 30
resultado = (num % 2 == 0) or (num % 3 == 0) or (num % 5 == 0)
print(resultado)
```

Explicación: or devuelve True si alguna de las condiciones se cumple.

Ejercicio 10: Operadores de identidad con listas

```
lista1 = [1, 2, 3]
lista2 = lista1
lista3 = [1, 2, 3]
print(lista1 is lista2) # True
print(lista1 is lista3) # False
```

Explicación: is verifica si dos variables apuntan al mismo objeto en memoria.

Ejercicio 11: Operaciones combinadas y precedencia

```
x = 5
y = 3
z = 2
resultado = x + y * z ** 2 - (x % z)
print(resultado)
```

Explicación: Python evalúa primero **, luego * y %, luego + y -.

Ejercicio 12: Calcular promedio de una lista y determinar si es alto

```
notas = [7, 9, 8, 10]
promedio = sum(notas)/len(notas)
alto = promedio > 8
print(promedio, alto)
```

Explicación: Combinamos operaciones aritméticas y de comparación.

Ejercicio 13: Uso de not y operadores lógicos en condiciones

```
a = True
b = False
resultado = not a or b
print(resultado)
```

Explicación: not a invierte el valor de a, y or combina los valores booleanos.

Ejercicio 14: Calcular número par más grande de una lista

```
numeros = [3, 7, 10, 14, 9]
par_mas_grande = max(n for n in numeros if n % 2 == 0)
print(par_mas_grande)
```

Explicación: Usamos comprensión de lista y max() para encontrar el número par más grande.

Ejercicio 15: Operador ternario para mensaje según edad

```
edad = 20
mensaje = "Mayor de edad" if edad >= 18 else "Menor de edad"
print(mensaje)
```

Explicación: Se usa el operador ternario para asignar un mensaje según una condición.

Ejercicio 16: Incrementar variable según condición compleja

```
x = 15
x += 10 if (x % 2 == 1 and x > 10) else 5
print(x)
```

Explicación: Condición compleja combinando módulo y comparación.

Ejercicio 17: Verificar si un string contiene vocales y su longitud

```
texto = "Python"
resultado = ("a" in texto or "e" in texto or "i" in texto or "o" in texto or "u"
in texto) and len(texto) > 5
print(resultado)
```

Explicación: Combinamos operadores `in`, `or` y `and` para condiciones de strings.

Ejercicio 18: Cálculo de promedio ponderado

```
notas = [7, 8, 9]
pesos = [0.2, 0.3, 0.5]
promedio_ponderado = sum(n*p for n, p in zip(notas, pesos))
print(promedio_ponderado)
```

Explicación: `zip` combina listas y usamos operadores para multiplicar y sumar ponderaciones.

Ejercicio 19: Comprobar si un número es primo usando operadores

```
num = 13
es_primo = all(num % i != 0 for i in range(2, num))
print(es_primo)
```

Explicación: `all()` devuelve `True` si ningún número entre 2 y `num-1` divide al número, es decir, es primo.

Ejercicio 20: Combinación de listas y operaciones aritméticas

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
resultado = [x + y for x, y in zip(lista1, lista2)]
print(resultado)
```

Explicación: `zip` combina listas y se suman elemento a elemento usando comprensión de listas.

Lección 2.: Estructuras Condicionales en Python

Las **estructuras condicionales** permiten a un programa **tomar decisiones** según ciertas condiciones. Son fundamentales en la programación porque permiten que el flujo de ejecución cambie según los valores de las variables o el resultado de expresiones lógicas.

1. Concepto de Condición

Una **condición** es una expresión que devuelve un valor booleano: `True` (verdadero) o `False` (falso).

Ejemplo:

```
edad = 18
mayor_de_edad = edad >= 18
print(mayor_de_edad)  # True
```

En este ejemplo, la condición `edad >= 18` devuelve `True` porque 18 es igual a 18.

2. Estructura `if`

La forma más básica de una condición en Python es la estructura `if`.

Sintaxis:

```
if condición:
    # bloque de código que se ejecuta si la condición es True
```

Ejemplo:

```
edad = 20
if edad >= 18:
    print("Eres mayor de edad")
```

Explicación:

Si la condición `edad >= 18` es verdadera, se ejecuta el bloque indentado. Si es falsa, Python simplemente lo omite.

3. Estructura `if-else`

Si queremos ejecutar un bloque **alternativo** cuando la condición es falsa, usamos `else`.

Sintaxis:

```
if condición:
    # bloque si True
```

```
else:
    # bloque si False
```

Ejemplo:

```
edad = 16
if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

Explicación:

Si `edad >= 18` es falsa, se ejecuta el bloque dentro de `else`.

4. Estructura `if-elif-else`

Cuando tenemos **múltiples condiciones**, usamos `elif` (else if).

Sintaxis:

```
if condición1:
    # bloque 1
elif condición2:
    # bloque 2
elif condición3:
    # bloque 3
else:
    # bloque por defecto
```

Ejemplo:

```
nota = 85

if nota >= 90:
    print("Sobresaliente")
elif nota >= 75:
    print("Notable")
elif nota >= 60:
    print("Aprobado")
else:
    print("Suspenso")
```

Explicación:

Python evalúa cada condición en orden. Cuando encuentra una verdadera, ejecuta ese bloque y salta las demás.

5. Operadores de Comparación en Condiciones

Para evaluar condiciones, usamos **operadores de comparación**:

Operador	Significado	Ejemplo
<code>==</code>	Igual a	<code>x == 5</code>
<code>!=</code>	Distinto de	<code>x != 10</code>

Operador	Significado	Ejemplo
>	Mayor que	x > 3
<	Menor que	x < 8
>=	Mayor o igual	x >= 7
<=	Menor o igual	x <= 12

Ejemplo:

```
a = 10
b = 20
if a < b:
    print("a es menor que b")
```

6. Operadores Lógicos en Condiciones

Podemos combinar condiciones usando **operadores lógicos**:

Operador	Significado	Ejemplo
and	Y lógico	a > 5 and b < 10
or	O lógico	a < 5 or b > 10
not	Negación lógica	not(a == 5)

Ejemplo:

```
edad = 25
ingresos = 3000
if edad > 18 and ingresos > 2500:
    print("Cumple los requisitos")
```

Explicación:

El bloque se ejecuta solo si **ambas condiciones** son verdaderas.

7. Condiciones Anidadas

Podemos poner un `if` dentro de otro `if` para evaluar múltiples criterios.

Ejemplo:

```
edad = 20
ingresos = 1800

if edad >= 18:
    if ingresos >= 2000:
        print("Puedes solicitar el crédito")
    else:
        print("No tienes ingresos suficientes")
else:
    print("Eres menor de edad")
```

Explicación:

Primero se evalúa la edad, y dentro de ese bloque se evalúa el ingreso.

8. Operador Ternario (Condición en una línea)

Python permite una **sintaxis corta para condicionales simples**:

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

Explicación:

Si `edad >= 18` es verdadero, `resultado` recibe `"Mayor de edad"`; de lo contrario `"Menor de edad"`.

9. Buenas Prácticas en Condicionales

1. **Indentación correcta:** Python requiere indentación para identificar bloques.
 2. **Evitar condiciones complejas en una línea:** divide en varias líneas si es necesario.
 3. **Usar `elif` en lugar de múltiples `if` independientes** para condiciones exclusivas.
 4. **Evitar comparaciones redundantes:** usa rangos y operadores lógicos.
 5. **Nombrar variables de manera descriptiva** para que las condiciones sean legibles.
-

10. Ejemplos Combinados

Ejemplo 1: Clasificación de edades

```
edad = 30

if edad < 13:
    print("Niño")
elif edad < 20:
    print("Adolescente")
elif edad < 60:
    print("Adulto")
else:
    print("Adulto mayor")
```

Ejemplo 2: Descuento según cantidad

```
cantidad = 15

if cantidad > 20:
    descuento = 20
elif cantidad > 10:
    descuento = 10
else:
    descuento = 0

print(f"Descuento: {descuento}%")
```

11. Ejercicios

11.1 Ejercicios de Python – Estructuras Condicionales (Nivel Principiante)

Ejercicio 1: Verificar si un número es positivo

```
num = 5
if num > 0:
    print("El número es positivo")
```

Explicación: Si num es mayor que 0, la condición es True y se imprime el mensaje.

Ejercicio 2: Verificar si un número es negativo

```
num = -3
if num < 0:
    print("El número es negativo")
```

Explicación: Solo se imprime si la condición num < 0 se cumple.

Ejercicio 3: Verificar si un número es par o impar

```
num = 7
if num % 2 == 0:
    print("El número es par")
else:
    print("El número es impar")
```

Explicación: % devuelve el resto. Si el resto es 0, el número es par; si no, impar.

Ejercicio 4: Comparar dos números

```
a = 10
b = 20
if a > b:
    print("a es mayor que b")
else:
    print("b es mayor o igual que a")
```

Explicación: Usamos > para comparar y else para el caso contrario.

Ejercicio 5: Comprobar si una persona es mayor de edad

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

Explicación: `>=` verifica si la edad es igual o superior a 18.

Ejercicio 6: Verificar si un número es múltiplo de 5

```
num = 15
if num % 5 == 0:
    print("El número es múltiplo de 5")
```

Explicación: `%` calcula el resto; si es 0, es divisible.

Ejercicio 7: Comparar tres números y mostrar el mayor

```
a = 5
b = 10
c = 7

if a > b and a > c:
    print("a es el mayor")
elif b > a and b > c:
    print("b es el mayor")
else:
    print("c es el mayor")
```

Explicación: Combinamos `and` y `elif` para múltiples condiciones.

Ejercicio 8: Determinar si un número está en un rango

```
num = 15
if 10 <= num <= 20:
    print("El número está entre 10 y 20")
```

Explicación: Python permite comparaciones encadenadas para rangos.

Ejercicio 9: Verificar si un string contiene cierta palabra

```
texto = "Hola mundo"
if "Hola" in texto:
    print("Se encontró la palabra 'Hola'")
```

Explicación: `in` comprueba pertenencia de una subcadena en un string.

Ejercicio 10: Asignar mensaje según calificación

```
nota = 7
if nota >= 6:
    print("Aprobado")
else:
    print("Suspenso")
```

Explicación: Condición simple con `if-else` según la nota.

Ejercicio 11: Determinar si un número es cero, positivo o negativo

```
num = 0
if num > 0:
    print("Positivo")
elif num < 0:
    print("Negativo")
else:
    print("Cero")
```

Explicación: Usamos `elif` para la segunda condición y `else` para el caso restante.

Ejercicio 12: Verificar si una letra es vocal

```
letra = "a"
if letra in "aeiou":
    print("La letra es vocal")
else:
    print("La letra es consonante")
```

Explicación: Comprobamos si la letra pertenece a la cadena `"aeiou"`.

Ejercicio 13: Comprobar si un número es múltiplo de 3 y 5

```
num = 15  
if num % 3 == 0 and num % 5 == 0:  
    print("El número es múltiplo de 3 y 5")
```

Explicación: and asegura que ambas condiciones se cumplan.

Ejercicio 14: Verificar si un número es mayor que 10 o menor que 5

```
num = 3  
if num > 10 or num < 5:  
    print("El número es mayor que 10 o menor que 5")
```

Explicación: or devuelve True si al menos una condición es verdadera.

Ejercicio 15: Operador ternario para mensaje según edad

```
edad = 17  
mensaje = "Mayor de edad" if edad >= 18 else "Menor de edad"  
print(mensaje)
```

Explicación: El operador ternario permite una condición simple en una línea.

Ejercicio 16: Determinar si una persona puede conducir

```
edad = 20  
licencia = True  
if edad >= 18 and licencia:  
    print("Puede conducir")  
else:  
    print("No puede conducir")
```

Explicación: Combinamos condiciones con and para verificar ambos requisitos.

Ejercicio 17: Comprobar si un número es divisible entre 2, 3 o 5

```
num = 30  
if num % 2 == 0 or num % 3 == 0 or num % 5 == 0:  
    print("El número es divisible entre 2, 3 o 5")
```

Explicación: or devuelve True si alguna de las condiciones es verdadera.

Ejercicio 18: Comparar edades de dos personas

```
edad1 = 25
edad2 = 30
if edad1 > edad2:
    print("La primera persona es mayor")
elif edad2 > edad1:
    print("La segunda persona es mayor")
else:
    print("Ambas personas tienen la misma edad")
```

Explicación: `elif` y `else` permiten múltiples alternativas.

Ejercicio 19: Verificar si un número es par y mayor que 10

```
num = 12
if num % 2 == 0 and num > 10:
    print("El número es par y mayor que 10")
```

Explicación: Ambas condiciones deben cumplirse para imprimir el mensaje.

Ejercicio 20: Determinar si un año es bisiesto

```
anio = 2024
if (anio % 4 == 0 and anio % 100 != 0) or (anio % 400 == 0):
    print("El año es bisiesto")
else:
    print("El año no es bisiesto")
```

Explicación: Combinamos `and` y `or` para cumplir la regla de los años bisiestos: divisible entre 4 y no entre 100, o divisible entre 400.

11.2 Ejercicios de Python – Estructuras Condicionales (Nivel Medio)

Ejercicio 1: Verificar si un número es positivo, negativo o cero

```
num = -5
if num > 0:
    print("Positivo")
elif num < 0:
    print("Negativo")
else:
    print("Cero")
```

Explicación: Usamos `if`, `elif` y `else` para manejar los tres posibles casos.

Ejercicio 2: Determinar si un año es bisiesto

```
anio = 2024
if (anio % 4 == 0 and anio % 100 != 0) or (anio % 400 == 0):
    print("Bisiesto")
else:
    print("No es bisiesto")
```

Explicación: Combinamos `and` y `or` para cumplir las reglas de años bisiestos.

Ejercicio 3: Comparar tres números y mostrar el mayor

```
a, b, c = 10, 25, 15
if a > b and a > c:
    mayor = a
elif b > a and b > c:
    mayor = b
else:
    mayor = c
print("El mayor es:", mayor)
```

Explicación: Usamos `elif` para evaluar condiciones exclusivas y determinar el mayor.

Ejercicio 4: Determinar si un estudiante aprueba, notable o sobresaliente

```
nota = 85
if nota >= 90:
    print("Sobresaliente")
elif nota >= 75:
    print("Notable")
elif nota >= 60:
    print("Aprobado")
else:
    print("Suspenso")
```

Explicación: Se evalúan rangos de notas usando condiciones encadenadas.

Ejercicio 5: Verificar si un número es divisible por 2, 3 o 5

```
num = 15
if num % 2 == 0 or num % 3 == 0 or num % 5 == 0:
    print("Divisible entre 2, 3 o 5")
```

Explicación: OR permite que se cumpla cualquiera de las condiciones.

Ejercicio 6: Clasificar edades en niño, adolescente, adulto o adulto mayor

```
edad = 45
if edad < 13:
    print("Niño")
elif edad < 20:
    print("Adolescente")
elif edad < 60:
    print("Adulto")
else:
    print("Adulto mayor")
```

Explicación: Evaluamos rangos de edad de forma secuencial.

Ejercicio 7: Verificar si un número es múltiplo de 4 y 6

```
num = 24
if num % 4 == 0 and num % 6 == 0:
    print("Múltiplo de 4 y 6")
```

Explicación: and asegura que ambas condiciones se cumplan simultáneamente.

Ejercicio 8: Determinar si un string es largo o corto

```
texto = "Python"
if len(texto) > 5:
    print("String largo")
else:
    print("String corto")
```

Explicación: `len()` devuelve la longitud del string y se compara con 5.

Ejercicio 9: Validar contraseña simple

```
password = "abc123"
if len(password) >= 6:
    print("Contraseña válida")
else:
    print("Contraseña demasiado corta")
```

Explicación: Condición basada en longitud mínima.

Ejercicio 10: Determinar si un número es positivo y par

```
num = 8
if num > 0 and num % 2 == 0:
    print("Número positivo y par")
```

Explicación: Combinamos condiciones con `and`.

Ejercicio 11: Verificar si una letra es vocal o consonante

```
letra = "e"
if letra.lower() in "aeiou":
    print("Vocal")
else:
    print("Consonante")
```

Explicación: `.lower()` asegura que la comparación no sea sensible a mayúsculas.

Ejercicio 12: Clasificar número en rangos de 10 en 10

```
num = 37
if num < 10:
    print("Menor que 10")
elif num < 20:
    print("Entre 10 y 19")
elif num < 30:
    print("Entre 20 y 29")
elif num < 40:
    print("Entre 30 y 39")
else:
    print("40 o más")
```

Explicación: Condiciones encadenadas permiten clasificar en rangos.

Ejercicio 13: Determinar si un número es divisible por 2 o 3, pero no ambos

```
num = 6
if (num % 2 == 0 or num % 3 == 0) and not (num % 2 == 0 and num % 3 == 0):
    print("Divisible por 2 o 3, pero no ambos")
```

Explicación: Combinamos and, or y not para una condición más compleja.

Ejercicio 14: Verificar si una lista está vacía

```
lista = []
if lista:
    print("La lista tiene elementos")
else:
    print("La lista está vacía")
```

Explicación: En Python, una lista vacía se evalúa como False.

Ejercicio 15: Operador ternario para mensaje según número positivo o negativo

```
num = -8
mensaje = "Positivo" if num > 0 else "Negativo o cero"
print(mensaje)
```

Explicación: Operador ternario permite una condición simple en una línea.

Ejercicio 16: Verificar si un número está dentro de un rango de 10 a 50

```
num = 35
if 10 <= num <= 50:
    print("Número dentro del rango")
else:
    print("Número fuera del rango")
```

Explicación: Python permite comparaciones encadenadas para rangos.

Ejercicio 17: Determinar si un año es siglo

```
anio = 2000
if anio % 100 == 0:
    print("Año de siglo")
else:
    print("No es año de siglo")
```

Explicación: Un año es de siglo si es divisible entre 100.

Ejercicio 18: Comparar dos strings alfabéticamente

```
a = "Python"
b = "Java"
if a > b:
    print(f"{a} viene después de {b} alfabéticamente")
else:
    print(f"{a} viene antes de {b} alfabéticamente")
```

Explicación: Python compara strings según su orden lexicográfico.

Ejercicio 19: Clasificar notas en letras A, B, C, D, F

```
nota = 78
if nota >= 90:
    letra = "A"
elif nota >= 80:
    letra = "B"
elif nota >= 70:
    letra = "C"
elif nota >= 60:
    letra = "D"
else:
    letra = "F"
print("Calificación:", letra)
```

Explicación: Evaluamos rangos de notas y asignamos letra correspondiente.

Ejercicio 20: Determinar si un número es primo simple (nivel medio)

```
num = 17
if num > 1:
    for i in range(2, num):
        if num % i == 0:
            print("No es primo")
            break
    else:
        print("Es primo")
else:
    print("No es primo")
```

Explicación: Usamos un bucle con condicional para verificar divisores. El `else` del `for` se ejecuta solo si no se rompe el bucle.

11.3 Ejercicios de Python – Estructuras Condicionales (Nivel Avanzado)

Ejercicio 1: Verificar si un número es primo

```
num = 29
if num > 1:
    for i in range(2, int(num**0.5)+1):
        if num % i == 0:
            print("No es primo")
            break
    else:
        print("Es primo")
else:
    print("No es primo")
```

Explicación: Solo números mayores a 1 pueden ser primos. Comprobamos divisores hasta la raíz cuadrada del número. El `else` del `for` se ejecuta si no se rompe el bucle.

Ejercicio 2: Clasificar números en positivos, negativos o cero usando listas

```
numeros = [10, -5, 0, 8, -3]
for n in numeros:
    if n > 0:
        print(n, "positivo")
    elif n < 0:
        print(n, "negativo")
    else:
        print(n, "cero")
```

Explicación: Iteramos sobre la lista y aplicamos condicionales a cada elemento.

Ejercicio 3: Determinar si un string tiene más de 5 vocales

```
texto = "Python avanzado"
vocales = "aeiouAEIOU"
contador = sum(1 for c in texto if c in vocales)
if contador > 5:
    print("Más de 5 vocales")
else:
    print("5 o menos vocales")
```

Explicación: Usamos comprensión de listas para contar vocales y condicional para clasificar.

Ejercicio 4: Verificar si un número es divisible por 2, 3 y 5 simultáneamente

```
num = 30
if num % 2 == 0 and num % 3 == 0 and num % 5 == 0:
    print("Divisible por 2, 3 y 5")
```

Explicación: and requiere que todas las condiciones sean verdaderas.

Ejercicio 5: Determinar el máximo de una lista sin usar max()

```
numeros = [5, 12, 8, 20, 3]
mayor = numeros[0]
for n in numeros:
    if n > mayor:
        mayor = n
print("Mayor:", mayor)
```

Explicación: Iteramos comparando cada elemento con el mayor actual.

Ejercicio 6: Asignar categoría según edad usando rangos complejos

```
edad = 35
if 0 <= edad <= 12:
    categoria = "Niño"
elif 13 <= edad <= 19:
    categoria = "Adolescente"
elif 20 <= edad <= 59:
    categoria = "Adulto"
else:
    categoria = "Adulto mayor"
print(categoria)
```

Explicación: Condiciones encadenadas permiten clasificar edades en rangos.

Ejercicio 7: Validar contraseña con varias condiciones

```
password = "Abc123!"
if len(password) >= 6 and any(c.isdigit() for c in password) and any(c.isupper()
for c in password):
    print("Contraseña segura")
else:
    print("Contraseña insegura")
```

Explicación: Combinamos longitud y comprobaciones con `and`.

Ejercicio 8: Determinar si un número es perfecto

```
num = 28
suma = sum(i for i in range(1, num) if num % i == 0)
if suma == num:
    print("Número perfecto")
else:
    print("No es perfecto")
```

Explicación: Un número perfecto es igual a la suma de sus divisores propios.

Ejercicio 9: Clasificar números en pares e impares dentro de una lista

```
numeros = [1, 2, 3, 4, 5, 6]
pares = []
impares = []
for n in numeros:
    if n % 2 == 0:
        pares.append(n)
    else:
        impares.append(n)
print("Pares:", pares)
print("Impares:", impares)
```

Explicación: Condicional dentro de un bucle para clasificar cada número.

Ejercicio 10: Verificar si una persona puede votar y si tiene carnet de identidad

```
edad = 20
tiene_carnet = True
if edad >= 18 and tiene_carnet:
    print("Puede votar")
else:
    print("No puede votar")
```

Explicación: and asegura que ambas condiciones se cumplan.

Ejercicio 11: Determinar la estación del año según mes

```
mes = 7
if mes in [12, 1, 2]:
    estacion = "Invierno"
elif mes in [3, 4, 5]:
    estacion = "Primavera"
elif mes in [6, 7, 8]:
    estacion = "Verano"
else:
    estacion = "Otoño"
print(estacion)
```

Explicación: Usamos listas dentro de in para verificar rangos de meses.

Ejercicio 12: Verificar si un número es divisible por uno de varios valores

```
num = 14
if any(num % x == 0 for x in [2, 3, 5, 7]):
    print("Divisible por 2, 3, 5 o 7")
```

Explicación: any() devuelve True si alguna condición se cumple.

Ejercicio 13: Determinar si un triángulo es equilátero, isósceles o escaleno

```
a, b, c = 5, 5, 7
if a == b == c:
    print("Equilátero")
elif a == b or a == c or b == c:
    print("Isósceles")
else:
    print("Escaleno")
```

Explicación: Comparamos los lados con operadores lógicos para clasificar el triángulo.

Ejercicio 14: Comprobar si un año es bisiesto usando condicional anidado

```
anio = 2000
if anio % 4 == 0:
    if anio % 100 != 0 or anio % 400 == 0:
        print("Bisiesto")
    else:
        print("No bisiesto")
else:
    print("No bisiesto")
```

Explicación: Condicional anidado para reflejar la regla exacta de años bisiestos.

Ejercicio 15: Validar entrada de usuario para número positivo

```
num = int(input("Ingrese un número: "))
if num > 0:
    print("Número positivo")
else:
    print("Número no positivo")
```

Explicación: Combinación de entrada y condicional.

Ejercicio 16: Determinar máximo y mínimo en una lista sin funciones predefinidas

```
numeros = [3, 7, 1, 9, 5]
maximo = numeros[0]
minimo = numeros[0]
for n in numeros:
    if n > maximo:
        maximo = n
    if n < minimo:
        minimo = n
print("Máximo:", maximo, "Mínimo:", minimo)
```

Explicación: Condicional dentro del bucle para actualizar máximo y mínimo.

Ejercicio 17: Clasificar edad con condicional anidado y operador ternario

```
edad = 17
categoria = "Mayor de edad" if edad >= 18 else "Menor de edad"
print(categoria)
```

Explicación: Operador ternario para decisión simple.

Ejercicio 18: Verificar si un string tiene letras y números

```
texto = "Python3"
if any(c.isalpha() for c in texto) and any(c.isdigit() for c in texto):
    print("Contiene letras y números")
```

Explicación: any() y condicional combinados permiten verificar ambas condiciones.

Ejercicio 19: Determinar si un número es capicúa

```
num = 12321
if str(num) == str(num)[::-1]:
    print("Capicúa")
else:
    print("No capicúa")
```

Explicación: Convertimos a string y comparamos con su reverso usando slicing.

Ejercicio 20: Clasificar días de la semana usando diccionario y condicional

```
dia = 3

dias_semana = {1: "Lunes", 2: "Martes", 3: "Miércoles", 4: "Jueves", 5:
"Viernes", 6: "Sábado", 7: "Domingo"}

if dia in dias_semana:
    print("Hoy es", dias_semana[dia])
else:
    print("Día inválido")
```

Explicación: Verificamos existencia de clave en diccionario antes de acceder al valor.

Lección 3.: Estructuras Repetitivas (Bucles)

1. Introducción

En programación, muchas veces necesitamos repetir una misma operación varias veces. Escribir manualmente las mismas instrucciones resultaría ineficiente, poco legible y difícil de mantener. Para resolver este problema, los lenguajes de programación ofrecen **estructuras repetitivas**, conocidas comúnmente como **bucles**.

Un bucle permite ejecutar un bloque de instrucciones varias veces, ya sea un número determinado de repeticiones o mientras se cumpla una condición.

Existen tres formas principales de repetición:

1. Repetición controlada por contador.
 2. Repetición controlada por condición.
 3. Repetición indefinida hasta que ocurra un evento.
-

2. Tipos de bucles más comunes

2.1. Bucle **for** (controlado por contador)

Se utiliza cuando sabemos de antemano cuántas veces queremos que se repita el bloque de instrucciones. Generalmente se usa un **contador** que aumenta o disminuye en cada iteración.

Ejemplo en pseudocódigo:

```
para i desde 1 hasta 5 hacer
    escribir("Iteración número ", i)
fin para
```

Ejemplo en Python:

```
for i in range(1, 6):
    print("Iteración número", i)
```

En este ejemplo, el bucle se ejecuta 5 veces, con valores de *i* de 1 a 5.

2.2. Bucle **while** (controlado por condición)

Se utiliza cuando **no sabemos cuántas veces** se repetirá la acción, pero sí sabemos la condición que debe cumplirse para seguir iterando.

Ejemplo en pseudocódigo:

```
contador = 1
mientras contador <= 5 hacer
    escribir("Iteración número ", contador)
    contador = contador + 1
```

fin mientras

Ejemplo en Python:

```
contador = 1
while contador <= 5:
    print("Iteración número", contador)
    contador += 1
```

El bucle `while` sigue ejecutándose mientras la condición sea verdadera.

2.3. Bucle `do while` (repetición garantizada)

Este bucle no existe directamente en Python, pero sí en otros lenguajes como C o Java. La diferencia es que la condición se evalúa **después** de ejecutar el bloque, lo que garantiza que al menos se ejecute una vez.

Ejemplo en pseudocódigo:

```
contador = 1
repetir
    escribir("Iteración número ", contador)
    contador = contador + 1
mientras contador <= 5
```

En Python, se puede simular con un `while True` y un `break`:

```
contador = 1
while True:
    print("Iteración número", contador)
    contador += 1
    if contador > 5:
        break
```

3. Componentes de un bucle

Todo bucle debe cumplir con tres elementos fundamentales:

1. **Inicialización:** asignar un valor inicial a la variable de control.
2. **Condición:** establece cuándo el bucle debe seguir ejecutándose.
3. **Actualización:** cambio del valor de la variable de control para evitar bucles infinitos.

Si falta alguno de estos pasos, se puede generar un **bucle infinito**, es decir, un ciclo que nunca termina.

4. Bucles infinitos

Un bucle infinito ocurre cuando la condición de salida nunca se cumple. Puede ser intencional (como en servidores que esperan conexiones) o un error de programación.

Ejemplo de bucle infinito en Python:

```
while True:
    print("Este bucle nunca termina")
```

Para salir de un bucle infinito se puede usar la instrucción **break**.

5. Instrucciones especiales en los bucles

Dentro de los bucles podemos usar palabras clave que modifican su comportamiento:

1. **break** → interrumpe el bucle inmediatamente.
2. **continue** → salta la iteración actual y pasa a la siguiente.
3. **pass** (en Python) → no hace nada, sirve como marcador de posición.

Ejemplo con break:

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
# Imprime 1, 2, 3, 4
```

Ejemplo con continue:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
# Imprime 1, 2, 4, 5 (salta el 3)
```

6. Anidación de bucles

Un bucle puede estar dentro de otro bucle. Esto se llama **bucles anidados**. Se utilizan en situaciones como recorrer matrices o generar combinaciones.

Ejemplo en Python:

```
for i in range(1, 4):
    for j in range(1, 4):
        print("i =", i, ", j =", j)
```

Salida:

```
i = 1 , j = 1
i = 1 , j = 2
i = 1 , j = 3
i = 2 , j = 1
...
```

7. Ejemplos prácticos

Ejemplo 1: Suma de los primeros 10 números

```
suma = 0
for i in range(1, 11):
    suma += i
print("La suma es:", suma)
```

Ejemplo 2: Validar entrada de usuario con while

```
contraseña = ""
while contraseña != "python123":
    contraseña = input("Introduce la contraseña: ")
print("Acceso concedido")
```

Ejemplo 3: Tabla de multiplicar

```
n = 7
for i in range(1, 11):
    print(n, "x", i, "=", n * i)
```

8. Conclusión

Las estructuras repetitivas son una herramienta esencial en la programación, pues permiten automatizar tareas repetitivas y ahorrar cientos de líneas de código.

- El bucle `for` es ideal cuando se conoce la cantidad de repeticiones.
- El bucle `while` se usa cuando solo conocemos la condición de parada.
- El `do while` asegura al menos una ejecución.
- Se pueden controlar los bucles con `break`, `continue` y anidación.

El dominio de los bucles es fundamental para resolver problemas complejos de manera eficiente.

8. Ejercicios

8.1 Ejercicios de Python – Estructuras Repetitivas, Bucles (Nivel Principiante)

EJERCICIO 1

Enunciado: Muestra los números del 1 al 10 usando un bucle for.

Solución:

```
for i in range(1, 11):  
    print(i)
```

Explicación: El bucle for recorre los valores del 1 al 10 y los imprime uno a uno.

EJERCICIO 2

Enunciado: Muestra los números pares del 2 al 20 usando un bucle for.

Solución:

```
for i in range(2, 21, 2):  
    print(i)
```

Explicación: Se usa range con paso 2 para mostrar solo números pares.

EJERCICIO 3

Enunciado: Muestra los números del 10 al 1 en orden descendente usando for.

Solución:

```
for i in range(10, 0, -1):  
    print(i)
```

Explicación: El rango va de 10 a 1 con paso negativo.

EJERCICIO 4

Enunciado: Calcula la suma de los números del 1 al 100 con un bucle for.

Solución:

```
suma = 0  
for i in range(1, 101):  
    suma += i  
print(suma)
```

Explicación: Se inicializa suma en 0 y se van acumulando los valores del 1 al 100.

EJERCICIO 5

Enunciado: Muestra la tabla de multiplicar del 7.

Solución:

```
for i in range(1, 11):  
    print("7 x", i, "=", 7 * i)
```

Explicación: Se recorre del 1 al 10 y se multiplica por 7 en cada paso.

EJERCICIO 6

Enunciado: Pide un número y muestra su factorial con un bucle for.

Solución:

```
n = int(input("Número: "))  
fact = 1  
for i in range(1, n+1):  
    fact *= i  
print("Factorial:", fact)
```

Explicación: Se multiplica el acumulador por cada número desde 1 hasta n.

EJERCICIO 7

Enunciado: Pide números hasta que el usuario introduzca 0, luego muestra la suma.

Solución:

```
suma = 0  
num = int(input("Número: "))  
while num != 0:  
    suma += num  
    num = int(input("Número: "))  
print("Suma total:", suma)
```

Explicación: El bucle while repite mientras el número no sea 0.

EJERCICIO 8

Enunciado: Muestra los primeros 10 números de la serie Fibonacci.

Solución:

```
a, b = 0, 1  
for i in range(10):  
    print(a)  
    a, b = b, a + b
```

Explicación: Se usa la fórmula de Fibonacci intercambiando valores en cada iteración.

EJERCICIO 9

Enunciado: Muestra solo los números impares del 1 al 20 usando continue.

Solución:

```
for i in range(1, 21):  
    if i % 2 == 0:  
        continue  
    print(i)
```

Explicación: Se salta la iteración cuando el número es par.

EJERCICIO 10

Enunciado: Muestra los números del 1 al 10 pero se detiene cuando llega al 5.

Solución:

```
for i in range(1, 11):  
    if i == 5:  
        break  
    print(i)
```

Explicación: Cuando i es 5, break interrumpe el bucle.

EJERCICIO 11

Enunciado: Muestra un cuadrado de asteriscos de 5x5 con bucles anidados.

Solución:

```
for i in range(5):  
    for j in range(5):  
        print("*", end=" ")  
    print()
```

Explicación: Un bucle imprime filas y el otro columnas.

EJERCICIO 12

Enunciado: Pide un número y muestra si es primo usando un bucle.

Solución:

```
n = int(input("Número: "))  
es_primo = True  
for i in range(2, n):  
    if n % i == 0:  
        es_primo = False  
        break  
print("Es primo" if es_primo else "No es primo")
```

Explicación: Se comprueba si el número tiene divisores distintos de 1 y él mismo.

EJERCICIO 13

Enunciado: Pide un número y muestra su tabla de multiplicar del 1 al 10.

Solución:

```
n = int(input("Número: "))
for i in range(1, 11):
    print(n, "x", i, "=", n*i)
```

Explicación: Se recorre del 1 al 10 multiplicando por n.

EJERCICIO 14

Enunciado: Imprime los números del 1 al 50 pero reemplaza los múltiplos de 3 por "Fizz".

Solución:

```
for i in range(1, 51):
    if i % 3 == 0:
        print("Fizz")
    else:
        print(i)
```

Explicación: Se evalúa la condición y se sustituye en caso de múltiplos de 3.

EJERCICIO 15

Enunciado: Pide una palabra y muestra sus letras una por una con un bucle for.

Solución:

```
palabra = input("Palabra: ")
for letra in palabra:
    print(letra)
```

Explicación: El for recorre directamente los caracteres de la cadena.

EJERCICIO 16

Enunciado: Pide una frase y cuenta cuántas vocales tiene usando un bucle.

Solución:

```
frase = input("Frase: ").lower()
contador = 0
for letra in frase:
    if letra in "aeiou":
        contador += 1
print("Número de vocales:", contador)
```

Explicación: Se recorre la frase y se suma cuando la letra es vocal.

EJERCICIO 17

Enunciado: Imprime los múltiplos de 5 entre 1 y 100.

Solución:

```
for i in range(5, 101, 5):  
    print(i)
```

Explicación: Se usa range con paso de 5.

EJERCICIO 18

Enunciado: Pide 5 números y calcula su promedio.

Solución:

```
suma = 0  
  
for i in range(5):  
    n = int(input("Número: "))  
    suma += n  
  
print("Promedio:", suma/5)
```

Explicación: Se acumulan los números y al final se divide entre 5.

EJERCICIO 19

Enunciado: Genera un triángulo de asteriscos de altura 5.

Solución:

```
for i in range(1, 6):  
    print("'" * i)
```

Explicación: En cada iteración se imprime un número de asteriscos igual al índice.

EJERCICIO 20

Enunciado: Pide una palabra y muéstrala al revés usando un bucle.

Solución:

```
palabra = input("Palabra: ")  
invertida = ""  
  
for letra in palabra:  
    invertida = letra + invertida  
  
print(invertida)
```

Explicación: Se construye la cadena al revés concatenando las letras en orden inverso.

8.2 Ejercicios de Python – Estructuras Repetitivas, Bucles (Nivel medio)

EJERCICIO 1

Enunciado: Muestra todos los números del 1 al 100 que sean múltiplos de 3 y de 5 al mismo tiempo.

Solución:

```
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print(i)
```

Explicación: Se recorre del 1 al 100 y se imprimen solo los múltiplos de 3 y 5.

EJERCICIO 2

Enunciado: Pide un número y muestra todos sus divisores.

Solución:

```
n = int(input("Número: "))
for i in range(1, n+1):
    if n % i == 0:
        print(i)
```

Explicación: Un divisor es aquel que divide al número sin dejar resto.

EJERCICIO 3

Enunciado: Calcula la suma de todos los números pares entre 1 y 100.

Solución:

```
suma = 0
for i in range(2, 101, 2):
    suma += i
print(suma)
```

Explicación: Se recorre solo los pares y se van acumulando.

EJERCICIO 4

Enunciado: Muestra los primeros 20 números de la serie de Fibonacci.

Solución:

```
a, b = 0, 1
for i in range(20):
    print(a, end=" ")
    a, b = b, a+b
```

Explicación: En cada paso se generan los términos sumando los dos anteriores.

EJERCICIO 5

Enunciado: Pide un número y calcula la suma de sus dígitos.

Solución:

```
n = int(input("Número: "))
suma = 0
while n > 0:
    suma += n % 10
    n //= 10
print(suma)
```

Explicación: Se extraen los dígitos con el módulo y se suman.

EJERCICIO 6

Enunciado: Pide una palabra y determina si es un palíndromo (igual al revés).

Solución:

```
palabra = input("Palabra: ")
invertida = ""
for letra in palabra:
    invertida = letra + invertida
print("Es palíndromo" if palabra == invertida else "No es palíndromo")
```

Explicación: Se construye la palabra al revés y se compara con la original.

EJERCICIO 7

Enunciado: Pide un número y muestra si es perfecto (la suma de sus divisores propios es igual al número).

Solución:

```
n = int(input("Número: "))
suma = 0
for i in range(1, n):
    if n % i == 0:
        suma += i
print("Es perfecto" if suma == n else "No es perfecto")
```

Explicación: Se suman los divisores menores que el número y se compara.

EJERCICIO 8

Enunciado: Imprime una pirámide de asteriscos de altura 5.

Solución:

```
for i in range(1, 6):
    print(" "*(5-i) + "*"*(2*i-1))
```

Explicación: Se combinan espacios y asteriscos para formar la figura centrada.

EJERCICIO 9

Enunciado: Genera los primeros 15 números primos.

Solución:

```
primos = []
num = 2
while len(primos) < 15:
    es_primo = True
    for i in range(2, num):
        if num % i == 0:
            es_primo = False
            break
    if es_primo:
        primos.append(num)
    num += 1
print(primos)
```

Explicación: Se prueba cada número y se almacena si no tiene divisores.

EJERCICIO 10

Enunciado: Pide un número y muéstralo en binario usando un bucle.

Solución:

```
n = int(input("Número: "))
binario = ""
while n > 0:
    binario = str(n % 2) + binario
    n //= 2
print(binario)
```

Explicación: Se divide entre 2 repetidamente y se forman los bits al revés.

EJERCICIO 11

Enunciado: Pide una frase y cuenta cuántas consonantes tiene.

Solución:

```
frase = input("Frase: ").lower()
contador = 0
for letra in frase:
    if letra.isalpha() and letra not in "aeiou":
        contador += 1
print(contador)
```

Explicación: Se recorren los caracteres, filtrando letras que no sean vocales.

EJERCICIO 12

Enunciado: Pide un número y determina si es un número Armstrong (la suma de sus dígitos elevados a la cantidad de dígitos es el número).

Solución:

```
n = int(input("Número: "))
d = len(str(n))
suma = 0
temp = n
while temp > 0:
    dig = temp % 10
    suma += dig**d
    temp //= 10
print("Es Armstrong" if suma == n else "No lo es")
```

Explicación: Se elevan los dígitos a la potencia del número de cifras.

EJERCICIO 13

Enunciado: Genera una lista con los cuadrados de los números del 1 al 20.

Solución:

```
cuadrados = []
for i in range(1, 21):
    cuadrados.append(i**2)
print(cuadrados)
```

Explicación: En cada iteración se calcula el cuadrado y se guarda en la lista.

EJERCICIO 14

Enunciado: Pide un número y calcula su potencia de 2 sin usar el operador **.

Solución:

```
n = int(input("Exponente: "))
resultado = 1
for i in range(n):
    resultado *= 2
print(resultado)
```

Explicación: Se multiplica 2 repetidamente n veces.

EJERCICIO 15

Enunciado: Imprime un triángulo invertido de asteriscos de base 7.

Solución:

```
for i in range(7, 0, -1):
    print("'" * i)
```

Explicación: Se imprime una fila de asteriscos decreciente.

EJERCICIO 16

Enunciado: Pide una palabra y cuenta cuántas veces aparece cada letra.

Solución:

```
palabra = input("Palabra: ").lower()
conteo = {}
for letra in palabra:
    if letra in conteo:
        conteo[letra] += 1
    else:
        conteo[letra] = 1
print(conteo)
```

Explicación: Se usa un diccionario y se incrementa el contador en cada iteración.

EJERCICIO 17

Enunciado: Muestra los primeros 10 números triangulares.

Solución:

```
suma = 0
for i in range(1, 11):
    suma += i
    print(suma)
```

Explicación: Un número triangular es la suma acumulada de los naturales.

EJERCICIO 18

Enunciado: Calcula el máximo común divisor (MCD) de dos números usando un bucle.

Solución:

```
a = int(input("Primer número: "))
b = int(input("Segundo número: "))
while b != 0:
    a, b = b, a % b
print("MCD:", a)
```

Explicación: Se aplica el algoritmo de Euclides con un bucle while.

EJERCICIO 19

Enunciado: Pide un número y calcula la suma de los factoriales de sus dígitos.

Solución:

```
import math
n = int(input("Número: "))
suma = 0
temp = n
while temp > 0:
    dig = temp % 10
    suma += math.factorial(dig)
    temp //= 10
print(suma)
```

Explicación: Se separan los dígitos y se calcula su factorial, luego se suman.

EJERCICIO 20

Enunciado: Pide un número n y genera un tablero de ajedrez de n x n con # y espacios.

Solución:

```
n = int(input("Tamaño: "))
for i in range(n):
    for j in range(n):
        if (i + j) % 2 == 0:
            print("#", end=" ")
        else:
            print(" ", end=" ")
    print()
```

Explicación: Se alternan # y espacios según la suma de las posiciones.

8.3 Ejercicios de Python – Estructuras Repetitivas, Bucles (Nivel avanzado)

EJERCICIO 1

Enunciado: Implementa el método de la criba de Eratóstenes para encontrar todos los números primos hasta 100.

Solución:

```
n = 100
primos = [True] * (n+1)
primos[0] = primos[1] = False
for i in range(2, int(n**0.5) + 1):
    if primos[i]:
        for j in range(i*i, n+1, i):
            primos[j] = False
print([i for i in range(n+1) if primos[i]])
```

Explicación: Se eliminan múltiplos de cada número primo en un rango creciente.

EJERCICIO 2

Enunciado: Implementa un programa que genere los primeros 10 números perfectos.

Solución:

```
def es_perfecto(n):
    return sum(i for i in range(1, n) if n % i == 0) == n

contador, num = 0, 2
while contador < 10:
    if es_perfecto(num):
        print(num)
        contador += 1
    num += 1
```

Explicación: Un número perfecto es igual a la suma de sus divisores propios.

EJERCICIO 3

Enunciado: Dibuja un triángulo de Pascal con 10 filas usando bucles.

Solución:

```
filas = 10
for i in range(filas):
    num = 1
    print(" "*(filas-i), end="")
    for j in range(i+1):
        print(num, end=" ")
        num = num * (i-j) // (j+1)
    print()
```

Explicación: Cada número del triángulo se calcula con combinaciones.

EJERCICIO 4

Enunciado: Calcula los 20 primeros números de la sucesión de Collatz a partir de un número dado.

Solución:

```
n = int(input("Número inicial: "))
for _ in range(20):
    print(n, end=" ")
    if n % 2 == 0:
        n //= 2
    else:
        n = 3*n + 1
```

Explicación: La sucesión aplica reglas diferentes si el número es par o impar.

EJERCICIO 5

Enunciado: Genera una espiral de tamaño $n \times n$ con números consecutivos.

Solución:

```
n = 5

matriz = [[0]*n for _ in range(n)]
num, top, left, right, bottom = 1, 0, 0, n-1, n-1

while left <= right and top <= bottom:
    for i in range(left, right+1):
        matriz[top][i] = num; num += 1
    top += 1
    for i in range(top, bottom+1):
        matriz[i][right] = num; num += 1
    right -= 1
    for i in range(right, left-1, -1):
        matriz[bottom][i] = num; num += 1
    bottom -= 1
    for i in range(bottom, top-1, -1):
        matriz[i][left] = num; num += 1
    left += 1

for fila in matriz:
    print(fila)
```

Explicación: Se llenan las filas y columnas por capas.

EJERCICIO 6

Enunciado: Encuentra el número más grande de la secuencia de Collatz para los números del 1 al 100.

Solución:

```
def collatz(n):
    maximo = n
    while n != 1:
        if n % 2 == 0:
            n //= 2
        else:
            n = 3*n + 1
        maximo = max(maximo, n)
    return maximo

mayor = 0
for i in range(1, 101):
    mayor = max(mayor, collatz(i))
print(mayor)
```

Explicación: Se calcula la secuencia y se almacena el valor máximo alcanzado.

EJERCICIO 7

Enunciado: Crea un programa que calcule los primeros 15 números abundantes.

Solución:

```
def es_abundante(n):  
    return sum(i for i in range(1, n) if n % i == 0) > n  
  
num, contador = 12, 0  
while contador < 15:  
    if es_abundante(num):  
        print(num)  
        contador += 1  
    num += 1
```

Explicación: Un número abundante es aquel cuya suma de divisores propios es mayor que él mismo.

EJERCICIO 8

Enunciado: Encuentra todos los palíndromos de 3 dígitos.

Solución:

```
for i in range(100, 1000):  
    if str(i) == str(i)[::-1]:  
        print(i)
```

Explicación: Se comparan los números con su versión invertida.

EJERCICIO 9

Enunciado: Imprime una matriz identidad de tamaño n usando bucles.

Solución:

```
n = 5  
for i in range(n):  
    for j in range(n):  
        if i == j:  
            print(1, end=" ")  
        else:  
            print(0, end=" ")  
    print()
```

Explicación: La matriz identidad tiene unos en la diagonal principal.

EJERCICIO 10

Enunciado: Genera el triángulo de Floyd hasta 20 filas.

Solución:

```
n, num = 20, 1
for i in range(1, n+1):
    for j in range(i):
        print(num, end=" ")
        num += 1
    print()
```

Explicación: Se imprime una serie creciente de números en forma triangular.

EJERCICIO 11

Enunciado: Encuentra el número más grande en una lista generada aleatoriamente con 100 enteros.

Solución:

```
import random
lista = [random.randint(1, 1000) for _ in range(100)]
maximo = lista[0]
for num in lista:
    if num > maximo:
        maximo = num
print(maximo)
```

Explicación: Se recorre la lista comparando cada número con el máximo actual.

EJERCICIO 12

Enunciado: Calcula el producto de matrices cuadradas de 3x3 con bucles anidados.

Solución:

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

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

```
C = [[0]*3 for _ in range(3)]
```

```
for i in range(3):
    for j in range(3):
        for k in range(3):
            C[i][j] += A[i][k] * B[k][j]
```

```
for fila in C:
    print(fila)
```

Explicación: Cada posición se calcula como la suma del producto de fila por columna.

EJERCICIO 13

Enunciado: Encuentra el número más largo de la secuencia de Collatz entre 1 y 1000.

Solución:

```
def longitud_collatz(n):
    count = 1
    while n != 1:
        n = n//2 if n%2==0 else 3*n+1
        count += 1
    return count
```

```
mayor, numero = 0, 0
for i in range(1, 1001):
    l = longitud_collatz(i)
    if l > mayor:
        mayor, numero = l, i
print(numero, mayor)
```

Explicación: Se calcula la longitud de cada secuencia y se guarda la más larga.

EJERCICIO 14

Enunciado: Genera una pirámide numérica de altura 5.

Solución:

```
for i in range(1, 6):  
    for j in range(1, i+1):  
        print(j, end=" ")  
    print()
```

Explicación: Cada fila imprime números desde 1 hasta la fila actual.

EJERCICIO 15

Enunciado: Genera un tablero de ajedrez de 8x8 alternando X y O.

Solución:

```
for i in range(8):
    for j in range(8):
        if (i+j) % 2 == 0:
            print("X", end=" ")
        else:
            print("O", end=" ")
    print()
```

Explicación: La suma de índices determina si va X u O.

EJERCICIO 16

Enunciado: Calcula la suma de los dígitos de los 100 primeros números.

Solución:

```
suma_total = 0
for i in range(1, 101):
    for d in str(i):
        suma_total += int(d)
print(suma_total)
```

Explicación: Se convierten los números en cadenas y se suman sus dígitos.

EJERCICIO 17

Enunciado: Calcula la secuencia de Hailstone más larga hasta 1000.

Solución:

```
def hailstone(n):
    secuencia = []
    while n != 1:
        secuencia.append(n)
        n = n//2 if n%2==0 else 3*n+1
    secuencia.append(1)
    return secuencia

mas_larga = []
for i in range(1, 1001):
    sec = hailstone(i)
    if len(sec) > len(mas_larga):
        mas_larga = sec
print(mas_larga)
```

Explicación: Se genera la secuencia de cada número y se almacena la más larga.

EJERCICIO 18

Enunciado: Muestra un rombo de asteriscos de altura 7.

Solución:

```
n = 7
for i in range(1, n+1, 2):
    print(" " * ((n-i)//2) + "*" * i)
for i in range(n-2, 0, -2):
    print(" " * ((n-i)//2) + "*" * i)
```

Explicación: Se imprimen las líneas superiores e inferiores simétricamente.

EJERCICIO 19

Enunciado: Encuentra los 10 números amigos más pequeños.

Solución:

```
def suma_divisores(n):
    return sum(i for i in range(1, n) if n % i == 0)

amigos = []
a = 220
while len(amigos) < 10:
    b = suma_divisores(a)
    if b != a and suma_divisores(b) == a:
        if (b, a) not in amigos:
            amigos.append((a, b))
    a += 1
print(amigos)
```

Explicación: Dos números son amigos si la suma de los divisores de uno da el otro.

EJERCICIO 20

Enunciado: Simula el triángulo de Sierpinski en texto hasta profundidad 4.

Solución:

```
n = 32
triangulo = [" "] * n
for _ in range(n):
    triangulo[_] = [" "] * n

def sierpinski(x, y, tam):
    if tam == 1:
        triangulo[y][x] = "*"
    else:
        sierpinski(x, y, tam//2)
        sierpinski(x-tam//2, y+tam//2, tam//2)
        sierpinski(x+tam//2, y+tam//2, tam//2)

sierpinski(n//2, 0, n//2)
for fila in triangulo:
    print("".join(fila))
```

Explicación: Se usa recursión y bucles para generar un fractal triangular.

Lección 4.: Estructuras de Datos – Listas en Python

1. Introducción a las listas

En programación, una **lista** es una estructura de datos que permite almacenar múltiples elementos en una sola variable.

En Python, las listas son muy versátiles: pueden contener números, cadenas, booleanos, objetos e incluso otras listas.

Ejemplo:

```
mi_lista = [10, 20, 30, 40, 50]
print(mi_lista) # [10, 20, 30, 40, 50]
```

Características principales:

- Son **ordenadas**: los elementos mantienen el orden en el que se agregan.
 - Son **mutables**: se pueden modificar después de su creación.
 - Pueden contener **tipos heterogéneos**: distintos tipos de datos en una misma lista.
 - Se pueden **anidar**: una lista puede contener otras listas.
-

2. Creación de listas

Existen varias formas de crear listas:

```
# Lista vacía
lista_vacia = []

# Lista con números
numeros = [1, 2, 3, 4, 5]

# Lista con cadenas
nombres = ["Ana", "Luis", "Marta"]

# Lista con valores de diferentes tipos
mixta = [1, "hola", True, 3.14]

# Usando la función list()
otra_lista = list((10, 20, 30))
```

3. Acceso a los elementos

Los elementos de una lista se acceden mediante **índices**, que empiezan en 0.

```
nombres = ["Ana", "Luis", "Marta"]
print(nombres[0]) # Ana
print(nombres[1]) # Luis
```

```
print(nombres[-1]) # Marta (índice negativo: desde el final)
```

4. Modificación de elementos

Las listas son mutables, lo que significa que se pueden cambiar sus elementos.

```
numeros = [10, 20, 30]
numeros[1] = 99
print(numeros) # [10, 99, 30]
```

También se pueden modificar varios elementos a la vez con **slicing**:

```
lista = [1, 2, 3, 4, 5]
lista[1:3] = [9, 8]
print(lista) # [1, 9, 8, 4, 5]
```

5. Recorrido de listas

Se pueden recorrer con bucles.

```
frutas = ["manzana", "pera", "uva"]
for fruta in frutas:
    print(fruta)
```

Con índices:

```
for i in range(len(frutas)):
    print(i, frutas[i])
```

6. Operaciones básicas con listas

Python ofrece varias operaciones útiles:

```
numeros = [1, 2, 3]

# Concatenación
print(numeros + [4, 5]) # [1, 2, 3, 4, 5]

# Repetición
print(numeros * 3) # [1, 2, 3, 1, 2, 3, 1, 2, 3]

# Pertenencia
print(2 in numeros) # True
print(10 not in numeros) # True

# Longitud
print(len(numeros)) # 3
```

7. Métodos principales de listas

Las listas cuentan con métodos muy útiles.

```
lista = [10, 20, 30]

# Agregar elementos
lista.append(40)
print(lista) # [10, 20, 30, 40]

# Insertar en una posición
lista.insert(1, 15)
print(lista) # [10, 15, 20, 30, 40]

# Extender con otra lista
lista.extend([50, 60])
print(lista) # [10, 15, 20, 30, 40, 50, 60]

# Eliminar un valor
lista.remove(20)
print(lista) # [10, 15, 30, 40, 50, 60]

# Eliminar por índice
lista.pop(2)
print(lista) # [10, 15, 40, 50, 60]

# Ordenar
lista.sort()
print(lista) # [10, 15, 40, 50, 60]

# Invertir
lista.reverse()
print(lista) # [60, 50, 40, 15, 10]
```

8. Slicing (rebanado de listas)

Permite obtener sublistas:

```
numeros = [0, 1, 2, 3, 4, 5, 6]

print(numeros[2:5]) # [2, 3, 4]
print(numeros[:4]) # [0, 1, 2, 3]
print(numeros[3:]) # [3, 4, 5, 6]
print(numeros[::2]) # [0, 2, 4, 6]
print(numeros[::-1]) # [6, 5, 4, 3, 2, 1, 0]
```

9. Listas anidadas (matrices)

Las listas pueden contener listas.

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

print(matriz[0]) # [1, 2, 3]
```

```
print(matriz[1][2]) # 6
```

10. List Comprehension

Es una forma concisa de crear listas con bucles.

```
# Crear lista con los cuadrados del 1 al 10
cuadrados = [x**2 for x in range(1, 11)]
print(cuadrados) # [1, 4, 9, ..., 100]

# Filtrar números pares
pares = [x for x in range(20) if x % 2 == 0]
print(pares) # [0, 2, 4, ..., 18]
```

11. Copias de listas

Al copiar listas hay que tener cuidado: si usas `=`, se copia la referencia, no el contenido.

```
a = [1, 2, 3]
b = a
b[0] = 99
print(a) # [99, 2, 3] (se modificó también a)

# Copia real
c = a.copy()
c[0] = 5
print(a) # [99, 2, 3]
print(c) # [5, 2, 3]
```

12. Conclusiones

- Las **listas** son la estructura de datos más usada en Python por su flexibilidad.
 - Permiten almacenar múltiples valores, modificarlos y recorrerlos con facilidad.
 - Disponen de múltiples métodos útiles (`append`, `remove`, `sort`, etc.).
 - Se pueden usar listas anidadas para representar matrices.
 - Las **list comprehensions** facilitan la creación de listas de forma compacta.
-

13. Ejercicios

13.1 Ejercicios de Python – Estructuras Datos, Listas(Nivel Principiante)

EJERCICIO 1

Enunciado: Crea una lista con los números del 1 al 5 e imprímela.

Solución:

```
lista = [1, 2, 3, 4, 5]
print(lista)
```

Explicación: Se define una lista y se muestra con print.

EJERCICIO 2

Enunciado: Crea una lista de nombres e imprime el primero y el último.

Solución:

```
nombres = ["Ana", "Luis", "Marta"]
print(nombres[0])
print(nombres[-1])
```

Explicación: Los índices positivos van desde el inicio, los negativos desde el final.

EJERCICIO 3

Enunciado: Cambia el segundo valor de una lista [10, 20, 30] por 99.

Solución:

```
numeros = [10, 20, 30]
numeros[1] = 99
print(numeros)
```

Explicación: Se asigna un nuevo valor en el índice 1.

EJERCICIO 4

Enunciado: Recorre una lista de frutas e imprime cada una.

Solución:

```
frutas = ["manzana", "pera", "uva"]
for fruta in frutas:
    print(fruta)
```

Explicación: El bucle for recorre todos los elementos.

EJERCICIO 5

Enunciado: Añade el número 40 a la lista [10, 20, 30].

Solución:

```
lista = [10, 20, 30]
lista.append(40)
print(lista)
```

Explicación: El método append agrega un valor al final.

EJERCICIO 6

Enunciado: Inserta el número 15 en la posición 1 de [10, 20, 30].

Solución:

```
lista = [10, 20, 30]
lista.insert(1, 15)
print(lista)
```

Explicación: Insert coloca el valor en el índice indicado.

EJERCICIO 7

Enunciado: Une dos listas [1, 2] y [3, 4].

Solución:

```
a = [1, 2]
b = [3, 4]
c = a + b
print(c)
```

Explicación: El operador + concatena listas.

EJERCICIO 8

Enunciado: Elimina el valor 20 de [10, 20, 30].

Solución:

```
lista = [10, 20, 30]
lista.remove(20)
print(lista)
```

Explicación: Remove borra la primera aparición de un valor.

EJERCICIO 9

Enunciado: Elimina el último elemento de [1, 2, 3].

Solución:

```
lista = [1, 2, 3]
lista.pop()
print(lista)
```

Explicación: Pop sin argumentos elimina el último valor.

EJERCICIO 10

Enunciado: Ordena la lista [3, 1, 2].

Solución:

```
lista = [3, 1, 2]
lista.sort()
print(lista)
```

Explicación: Sort organiza los valores de menor a mayor.

EJERCICIO 11

Enunciado: Invierte el orden de [1, 2, 3].

Solución:

```
lista = [1, 2, 3]
lista.reverse()
print(lista)
```

Explicación: Reverse cambia el orden al inverso.

EJERCICIO 12

Enunciado: Cuenta cuántas veces aparece el número 2 en [1, 2, 2, 3].

Solución:

```
lista = [1, 2, 2, 3]
print(lista.count(2))
```

Explicación: Count devuelve el número de ocurrencias.

EJERCICIO 13

Enunciado: Encuentra el índice del valor 30 en [10, 20, 30, 40].

Solución:

```
lista = [10, 20, 30, 40]
print(lista.index(30))
```

Explicación: Index devuelve la posición del valor buscado.

EJERCICIO 14

Enunciado: Copia la lista [1, 2, 3] en otra variable.

Solución:

```
a = [1, 2, 3]
b = a.copy()
print(b)
```

Explicación: Copy crea una copia independiente de la lista.

EJERCICIO 15

Enunciado: Crea una sublista con los tres primeros elementos de [10, 20, 30, 40, 50].

Solución:

```
lista = [10, 20, 30, 40, 50]
sublista = lista[:3]
print(sublista)
```

Explicación: El slicing [:3] selecciona desde el inicio hasta el índice 2.

EJERCICIO 16

Enunciado: Crea una lista con los números del 1 al 10 usando range().

Solución:

```
lista = list(range(1, 11))
print(lista)
```

Explicación: Range genera números en un rango y list los convierte en lista.

EJERCICIO 17

Enunciado: Comprueba si el valor 5 está en [1, 2, 3, 4, 5].

Solución:

```
lista = [1, 2, 3, 4, 5]
print(5 in lista)
```

Explicación: El operador in verifica la pertenencia en la lista.

EJERCICIO 18

Enunciado: Crea una lista de cuadrados de 1 a 5 usando comprensión de listas.

Solución:

```
cuadrados = [x**2 for x in range(1, 6)]
print(cuadrados)
```

Explicación: Se genera una lista aplicando la operación a cada elemento.

EJERCICIO 19

Enunciado: Encuentra el valor máximo y mínimo en [10, 5, 20, 3].

Solución:

```
lista = [10, 5, 20, 3]
print(max(lista))
print(min(lista))
```

Explicación: Max devuelve el mayor valor y min el menor.

EJERCICIO 20

Enunciado: Crea una lista anidada y accede a un valor interno.

Solución:

```
matriz = [[1, 2], [3, 4], [5, 6]]
print(matriz[1][0])
```

Explicación: El primer índice selecciona la sublista, el segundo el valor dentro de ella.

13.2 Ejercicios de Python – Estructuras Datos, Listas(Nivel Medio)

EJERCICIO 1

Enunciado: Pide 5 números al usuario, guárdalos en una lista y muéstrala ordenada.

Solución:

```
numeros = []  
for i in range(5):  
    n = int(input("Número: "))  
    numeros.append(n)  
numeros.sort()  
print(numeros)
```

Explicación: Se van guardando los números con append, luego se usa sort para ordenarlos.

EJERCICIO 2

Enunciado: Elimina los duplicados de la lista [1, 2, 2, 3, 4, 4, 5].

Solución:

```
lista = [1, 2, 2, 3, 4, 4, 5]  
sin_repetidos = list(set(lista))  
print(sin_repetidos)
```

Explicación: Se convierte en set para eliminar duplicados y luego de nuevo en lista.

EJERCICIO 3

Enunciado: Calcula la suma de todos los valores de la lista [10, 20, 30, 40] sin usar sum().

Solución:

```
lista = [10, 20, 30, 40]  
suma = 0  
for n in lista:  
    suma += n  
print(suma)
```

Explicación: Se acumula el valor de cada elemento en una variable suma.

EJERCICIO 4

Enunciado: Invierte la lista [1, 2, 3, 4, 5] sin usar reverse().

Solución:

```
lista = [1, 2, 3, 4, 5]
invertida = lista[::-1]
print(invertida)
```

Explicación: El slicing con paso -1 devuelve la lista en orden inverso.

EJERCICIO 5

Enunciado: Encuentra el segundo mayor número en [10, 40, 30, 20].

Solución:

```
lista = [10, 40, 30, 20]
lista.sort()
print(lista[-2])
```

Explicación: Se ordena la lista y se toma el penúltimo valor.

EJERCICIO 6

Enunciado: Pide una palabra y crea una lista con sus letras.

Solución:

```
palabra = input("Palabra: ")
lista = list(palabra)
print(lista)
```

Explicación: La función list convierte una cadena en lista de caracteres.

EJERCICIO 7

Enunciado: Dada una lista de números, crea otra con los números pares.

Solución:

```
numeros = [1, 2, 3, 4, 5, 6]
pares = [n for n in numeros if n % 2 == 0]
print(pares)
```

Explicación: Se filtran con comprensión de listas los que son divisibles entre 2.

EJERCICIO 8

Enunciado: Crea una lista con los múltiplos de 3 entre 1 y 30.

Solución:

```
lista = [n for n in range(1, 31) if n % 3 == 0]
print(lista)
```

Explicación: Se generan los números de 1 a 30 y se seleccionan los divisibles entre 3.

EJERCICIO 9

Enunciado: Calcula el promedio de los valores de [5, 10, 15, 20].

Solución:

```
lista = [5, 10, 15, 20]
promedio = sum(lista) / len(lista)
print(promedio)
```

Explicación: Se divide la suma de los elementos entre el número de elementos.

EJERCICIO 10

Enunciado: Convierte la lista [1, 2, 3] en una cadena "1-2-3".

Solución:

```
lista = [1, 2, 3]
cadena = "-".join(map(str, lista))
print(cadena)
```

Explicación: Join une elementos de una lista de cadenas, por eso se convierten con map a str.

EJERCICIO 11

Enunciado: Recorre la lista [10, 20, 30] mostrando índice y valor.

Solución:

```
lista = [10, 20, 30]
for i, valor in enumerate(lista):
    print(i, valor)
```

Explicación: Enumerate devuelve índice y valor en cada iteración.

EJERCICIO 12

Enunciado: Elimina los valores menores que 10 de [5, 12, 3, 18, 20].

Solución:

```
lista = [5, 12, 3, 18, 20]
filtrada = [n for n in lista if n >= 10]
print(filtrada)
```

Explicación: Se construye una nueva lista con los valores mayores o iguales a 10.

EJERCICIO 13

Enunciado: Crea una lista de listas que represente una matriz de 3x3 con valores del 1 al 9.

Solución:

```
matriz = [[1, 2, 3],  
          [4, 5, 6],  
          [7, 8, 9]]  
  
print(matriz)
```

Explicación: Una lista puede contener otras listas para simular una matriz.

EJERCICIO 14

Enunciado: Extrae la primera columna de la matriz del ejercicio anterior.

Solución:

```
columna = [fila[0] for fila in matriz]  
  
print(columna)
```

Explicación: Se recorre cada fila y se toma el valor en la posición 0.

EJERCICIO 15

Enunciado: Pide al usuario 5 nombres y guárdalos en una lista.

Solución:

```
nombres = []  
  
for i in range(5):  
    nombre = input("Nombre: ")  
    nombres.append(nombre)  
  
print(nombres)
```

Explicación: Con un bucle se van pidiendo nombres y se almacenan en una lista.

EJERCICIO 16

Enunciado: Combina dos listas [1, 2, 3] y ['a', 'b', 'c'] en una lista de tuplas.

Solución:

```
a = [1, 2, 3]  
b = ['a', 'b', 'c']  
combinada = list(zip(a, b))  
  
print(combinada)
```

Explicación: Zip combina elementos de ambas listas en pares.

EJERCICIO 17

Enunciado: Calcula el producto de todos los valores de [2, 3, 4].

Solución:

```
lista = [2, 3, 4]
producto = 1
for n in lista:
    producto *= n
print(producto)
```

Explicación: Se inicializa el acumulador en 1 y se multiplica por cada elemento.

EJERCICIO 18

Enunciado: Encuentra los elementos comunes entre [1, 2, 3, 4] y [3, 4, 5, 6].

Solución:

```
a = [1, 2, 3, 4]
b = [3, 4, 5, 6]
comunes = list(set(a) & set(b))
print(comunes)
```

Explicación: Se convierten en conjuntos y se obtiene la intersección.

EJERCICIO 19

Enunciado: Crea una lista con los cuadrados de los números pares del 1 al 10.

Solución:

```
lista = [n**2 for n in range(1, 11) if n % 2 == 0]
print(lista)
```

Explicación: Se recorren los números del 1 al 10 y se guardan solo los pares elevados al cuadrado.

EJERCICIO 20

Enunciado: Encuentra el valor más largo en la lista de palabras ["sol", "estrella", "luna"].

Solución:

```
palabras = ["sol", "estrella", "luna"]
larga = max(palabras, key=len)
print(larga)
```

Explicación: La función max con el argumento key=len selecciona la palabra con mayor longitud.

13.3 Ejercicios de Python – Estructuras Datos, Listas(Nivel Avanzado)

EJERCICIO 1

Enunciado: Dada una lista de listas, aplánala en una sola lista. Ejemplo `[[1, 2], [3, 4], [5]]` → `[1, 2, 3, 4, 5]`.

Solución:

```
listas = [[1, 2], [3, 4], [5]]
plana = [x for sublista in listas for x in sublista]
print(plana)
```

Explicación: Se usa comprensión de listas anidada para recorrer cada sublista y extraer sus elementos.

EJERCICIO 2

Enunciado: Elimina duplicados de una lista pero manteniendo el orden.

Solución:

```
lista = [1, 2, 2, 3, 1, 4, 3]
sin_repetidos = []
for n in lista:
    if n not in sin_repetidos:
        sin_repetidos.append(n)
print(sin_repetidos)
```

Explicación: Se verifica si el elemento ya está en la nueva lista antes de añadirlo.

EJERCICIO 3

Enunciado: Rota una lista 2 posiciones a la derecha. `[1, 2, 3, 4]` → `[3, 4, 1, 2]`.

Solución:

```
lista = [1, 2, 3, 4]
k = 2
rotada = lista[-k:] + lista[:-k]
print(rotada)
```

Explicación: Se toma el final de la lista y se coloca delante.

EJERCICIO 4

Enunciado: Encuentra el tercer mayor elemento de [10, 50, 30, 20, 40].

Solución:

```
lista = [10, 50, 30, 20, 40]
tercero = sorted(set(lista))[-3]
print(tercero)
```

Explicación: Se ordena la lista sin duplicados y se toma el tercer elemento desde el final.

EJERCICIO 5

Enunciado: Divide una lista en sublistas de tamaño 3.

Solución:

```
lista = [1, 2, 3, 4, 5, 6, 7]
sublistas = [lista[i:i+3] for i in range(0, len(lista), 3)]
print(sublistas)
```

Explicación: Se usa slicing con saltos de 3.

EJERCICIO 6

Enunciado: Genera una lista con todas las combinaciones posibles de [1, 2] y ['a', 'b'].

Solución:

```
a = [1, 2]
b = ['a', 'b']
combinaciones = [(x, y) for x in a for y in b]
print(combinaciones)
```

Explicación: Se generan pares de cada elemento de la primera lista con cada uno de la segunda.

EJERCICIO 7

Enunciado: Encuentra la subsecuencia más larga de elementos consecutivos en [1, 2, 3, 10, 11, 12, 13, 20].

Solución:

```
lista = [1,2,3,10,11,12,13,20]
lista.sort()
max_long = 1
actual = 1
for i in range(1, len(lista)):
    if lista[i] == lista[i-1] + 1:
        actual += 1
        max_long = max(max_long, actual)
    else:
        actual = 1
print(max_long)
```

Explicación: Se recorre la lista ordenada y se cuenta la secuencia consecutiva más larga.

EJERCICIO 8

Enunciado: Aplana una lista con varios niveles de anidación [[1, [2, 3]], [4]] → [1, 2, 3, 4].

Solución:

```
def aplanar(lista):
    resultado = []
    for elem in lista:
        if isinstance(elem, list):
            resultado.extend(aplanar(elem))
        else:
            resultado.append(elem)
    return resultado

print(aplanar([[1, [2, 3]], [4]]))
```

Explicación: Se usa recursividad para recorrer y descomponer listas dentro de listas.

EJERCICIO 9

Enunciado: Calcula la moda (valor más repetido) de una lista.

Solución:

```
lista = [1, 2, 2, 3, 3, 3, 4]
moda = max(set(lista), key=lista.count)
print(modas)
```

Explicación: La función max selecciona el valor con mayor frecuencia en la lista.

EJERCICIO 10

Enunciado: Transpone una matriz 3x3 representada como lista de listas.

Solución:

```
matriz = [[1,2,3],[4,5,6],[7,8,9]]
transpuesta = [[fila[i] for fila in matriz] for i in range(len(matriz[0]))]
print(transpuesta)
```

Explicación: Se intercambian filas por columnas mediante comprensión anidada.

EJERCICIO 11

Enunciado: Encuentra todos los pares de números en [1, 2, 3, 4, 5] cuya suma sea 6.

Solución:

```
lista = [1,2,3,4,5]
pares = [(x, y) for i, x in enumerate(lista) for y in lista[i+1:] if x+y == 6]
print(pares)
```

Explicación: Se recorren combinaciones únicas y se filtran las que suman 6.

EJERCICIO 12

Enunciado: Genera una lista con los números primos del 1 al 30.

Solución:

```
def es_primo(n):  
    if n < 2: return False  
    for i in range(2, int(n**0.5)+1):  
        if n % i == 0:  
            return False  
    return True  
  
primos = [n for n in range(1,31) if es_primo(n)]  
print(primos)
```

Explicación: Se define una función que verifica primalidad y se usa comprensión de listas.

EJERCICIO 13

Enunciado: Calcula la intersección de varias listas.

Solución:

```
listas = [[1,2,3,4], [2,3,5], [2,3,6]]  
interseccion = list(set(listas[0]).intersection(*listas[1:]))  
print(interseccion)
```

Explicación: Se usa intersection sobre conjuntos para obtener elementos comunes.

EJERCICIO 14

Enunciado: Encuentra el subarreglo con la suma máxima en [1, -2, 3, 5, -1, 2].

Solución:

```
lista = [1,-2,3,5,-1,2]  
max_actual = max_total = lista[0]  
for n in lista[1:]:  
    max_actual = max(n, max_actual+n)  
    max_total = max(max_total, max_actual)  
print(max_total)
```

Explicación: Se aplica el algoritmo de Kadane para encontrar la suma máxima continua.

EJERCICIO 15

Enunciado: Divide una lista en partes iguales. Ejemplo [1, 2, 3, 4, 5, 6] en 3 partes → [[1, 2], [3, 4], [5, 6]].

Solución:

```
lista = [1,2,3,4,5,6]
n = 3
tam = len(lista)//n
partes = [lista[i*tam:(i+1)*tam] for i in range(n)]
print(partes)
```

Explicación: Se calculan cortes iguales de la lista con slicing.

EJERCICIO 16

Enunciado: Genera todas las permutaciones de [1, 2, 3].

Solución:

```
import itertools
lista = [1,2,3]
permutaciones = list(itertools.permutations(lista))
print(permutaciones)
```

Explicación: La librería itertools genera todas las permutaciones posibles.

EJERCICIO 17

Enunciado: Genera todas las combinaciones de 2 elementos de [1, 2, 3, 4].

Solución:

```
import itertools
lista = [1,2,3,4]
combinaciones = list(itertools.combinations(lista, 2))
print(combinaciones)
```

Explicación: Combinations devuelve subconjuntos de longitud especificada.

EJERCICIO 18

Enunciado: Dada una lista de listas, calcula la suma de cada sublista.

Solución:

```
listas = [[1,2,3],[4,5],[6]]
sumas = [sum(sublista) for sublista in listas]
print(sumas)
```

Explicación: Se aplica sum sobre cada sublista mediante comprensión.

EJERCICIO 19

Enunciado: Encuentra el elemento más frecuente de una lista de palabras.

Solución:

```
palabras = ["sol", "luna", "sol", "estrella", "luna", "sol"]
frecuente = max(set(palabras), key=palabras.count)
print(frecuente)
```

Explicación: Se convierte en conjunto y se usa max con la clave count.

EJERCICIO 20

Enunciado: Recorre una lista de números y clasifica cada uno en otra lista según sea par o impar.

Solución:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = []
impares = []
for n in numeros:
    (pares if n%2==0 else impares).append(n)
print("Pares:", pares)
print("Impares:", impares)
```

Explicación: Se usa un condicional dentro del bucle para decidir en qué lista agregar el número.

Lección 5.: Estructuras de Datos en Python – Conjuntos

1. Introducción a los Conjuntos

Un **conjunto** es una colección **no ordenada**, **mutable** (se puede modificar) y que **no permite elementos duplicados**. Se utiliza cuando necesitamos almacenar elementos únicos y realizar operaciones como unión, intersección o diferencia.

En Python, los conjuntos se representan con la clase `set`.

Características principales:

- No permiten duplicados.
 - No tienen un orden definido (no se puede acceder a un elemento por índice).
 - Se pueden modificar (agregar o eliminar elementos).
 - Son útiles para operaciones matemáticas como unión, intersección y diferencia.
-

2. Creación de Conjuntos

a) Conjunto vacío

```
conjunto_vacio = set() # No se puede usar {} porque eso crea un diccionario vacío
print(conjunto_vacio) # Output: set()
```

b) Conjunto con elementos

```
numeros = {1, 2, 3, 4, 5}
frutas = {"manzana", "banana", "cereza"}
print(numeros) # Output: {1, 2, 3, 4, 5}
print(frutas) # Output: {'banana', 'manzana', 'cereza'}
```

c) Duplicados

```
conjunto = {1, 2, 2, 3, 3, 3}
print(conjunto) # Output: {1, 2, 3} -> duplicados eliminados automáticamente
```

3. Operaciones básicas con conjuntos

a) Añadir elementos

```
conjunto = {1, 2, 3}
conjunto.add(4)
print(conjunto) # Output: {1, 2, 3, 4}
```

b) Eliminar elementos

```
conjunto.remove(2) # Elimina el elemento 2; da error si no existe
conjunto.discard(5) # Elimina el elemento 5 si existe; no da error si no existe
print(conjunto) # Output: {1, 3, 4}
```

c) Verificar existencia

```
print(3 in conjunto) # Output: True
print(10 in conjunto) # Output: False
```

d) Longitud de un conjunto

```
print(len(conjunto)) # Output: 3
```

4. Operaciones matemáticas con conjuntos

Python permite realizar operaciones matemáticas entre conjuntos:

a) Unión

Une dos conjuntos eliminando duplicados.

```
A = {1, 2, 3}
B = {3, 4, 5}
union = A | B
print(union) # Output: {1, 2, 3, 4, 5}
```

b) Intersección

Elementos comunes a ambos conjuntos.

```
interseccion = A & B
print(interseccion) # Output: {3}
```

c) Diferencia

Elementos de un conjunto que no están en el otro.

```
diferencia = A - B
print(diferencia) # Output: {1, 2}
```

d) Diferencia simétrica

Elementos que están en uno u otro, pero no en ambos.

```
diferencia_simetrica = A ^ B
print(diferencia_simetrica) # Output: {1, 2, 4, 5}
```

5. Métodos útiles de conjuntos

Método	Descripción
<code>add(elemento)</code>	Añade un elemento al conjunto
<code>remove(elemento)</code>	Elimina un elemento; da error si no existe
<code>discard(elemento)</code>	Elimina un elemento si existe; no da error
<code>pop()</code>	Elimina un elemento aleatorio y lo devuelve
<code>clear()</code>	Elimina todos los elementos del conjunto
<code>copy()</code>	Crea una copia del conjunto
<code>update(otro_set)</code>	Agrega todos los elementos de otro conjunto
<code>intersection_update(otro_set)</code>	Mantiene solo los elementos comunes
<code>difference_update(otro_set)</code>	Elimina elementos presentes en otro conjunto
<code>symmetric_difference_update(otro_set)</code>	Actualiza con la diferencia simétrica

6. Conjuntos inmutables: frozenset

Si necesitas un conjunto **immutable** (no se puede modificar), se usa `frozenset`.

```
A = frozenset([1, 2, 3])
# A.add(4) -> Esto generaría un error
```

`frozenset` es útil cuando quieres usar conjuntos como **llaves de diccionarios** o almacenarlos dentro de otros conjuntos.

7. Aplicaciones prácticas de conjuntos

- **Eliminar duplicados:**

```
lista = [1, 2, 2, 3, 4, 4, 5]
sin_duplicados = list(set(lista))
print(sin_duplicados) # Output: [1, 2, 3, 4, 5]
```

- **Comparación de datos:**

```
usuarios_movil = {"Ana", "Luis", "Pedro"}
usuarios_web = {"Luis", "Pedro", "Carlos"}
usuarios_comunes = usuarios_movil & usuarios_web
print(usuarios_comunes) # Output: {'Pedro', 'Luis'}
```

- **Operaciones de teoría de conjuntos:** unión, intersección, diferencia, etc., son ideales para bases de datos y análisis de datos.
-

8. Buenas prácticas

1. Usa conjuntos cuando necesites elementos únicos y no te importe el orden.
 2. Evita usar `remove` si no estás seguro de que el elemento exista; `discard` es más seguro.
 3. Prefiere `frozenset` cuando necesites un conjunto inmutable.
 4. Para conjuntos grandes, las operaciones de intersección y diferencia son más eficientes que iterar manualmente.
-

9. Ejercicios prácticos

9.1 Ejercicios de Python – Estructuras de Datos, Conjuntos (Nivel Principiante)

Ejercicio 1

Enunciado: Crea un conjunto llamado `frutas` con los elementos "manzana", "banana" y "cereza".

Solución:

```
frutas = {"manzana", "banana", "cereza"}  
print(frutas)
```

Explicación: Se usa `{ }` para crear un conjunto con elementos únicos. Al imprimirlo, Python puede mostrar los elementos en cualquier orden porque los conjuntos no son ordenados.

Ejercicio 2

Enunciado: Añade "naranja" al conjunto `frutas`.

Solución:

```
frutas.add("naranja")  
print(frutas)
```

Explicación: `add( )` agrega un elemento al conjunto. Si el elemento ya existiera, no haría nada.

Ejercicio 3

Enunciado: Elimina "banana" del conjunto `frutas`.

Solución:

```
frutas.remove("banana")  
print(frutas)
```

Explicación: `remove( )` elimina un elemento específico del conjunto. Si el elemento no existe, genera un error.

Ejercicio 4

Enunciado: Elimina "pera" del conjunto `frutas` sin que genere error si no existe.

Solución:

```
frutas.discard("pera")
print(frutas)
```

Explicación: `discard()` elimina un elemento si existe; si no, no hace nada. Es más seguro que `remove()` cuando no se está seguro de la existencia del elemento.

Ejercicio 5

Enunciado: Comprueba si "manzana" está en el conjunto `frutas`.

Solución:

```
print("manzana" in frutas)
```

Explicación: El operador `in` devuelve `True` si el elemento está en el conjunto, `False` en caso contrario.

Ejercicio 6

Enunciado: Crea un conjunto de números del 1 al 5.

Solución:

```
numeros = {1, 2, 3, 4, 5}
print(numeros)
```

Explicación: Se usan llaves `{}` para definir un conjunto de enteros. Los elementos duplicados se eliminan automáticamente.

Ejercicio 7

Enunciado: Combina los conjuntos `{1, 2, 3}` y `{3, 4, 5}` usando **unión**.

Solución:

```
A = {1, 2, 3}
B = {3, 4, 5}
union = A | B
print(union)
```

Explicación: El operador `|` realiza la unión de conjuntos, devolviendo todos los elementos sin duplicados.

Ejercicio 8

Enunciado: Encuentra los elementos comunes entre {1, 2, 3} y {2, 3, 4} usando **intersección**.

Solución:

```
A = {1, 2, 3}
B = {2, 3, 4}
interseccion = A & B
print(interseccion)
```

Explicación: El operador & devuelve los elementos que están en ambos conjuntos.

Ejercicio 9

Enunciado: Encuentra los elementos que están en {1, 2, 3} pero no en {2, 3, 4} usando **diferencia**.

Solución:

```
diferencia = {1, 2, 3} - {2, 3, 4}
print(diferencia)
```

Explicación: El operador - devuelve los elementos que están en el primer conjunto pero no en el segundo.

Ejercicio 10

Enunciado: Encuentra los elementos que están en uno u otro conjunto, pero no en ambos (diferencia simétrica).

Solución:

```
A = {1, 2, 3}
B = {2, 3, 4}
diferencia_sim = A ^ B
print(diferencia_sim)
```

Explicación: El operador ^ devuelve la diferencia simétrica: los elementos únicos de cada conjunto.

Ejercicio 11

Enunciado: Crea un conjunto vacío llamado `mi_conjunto`.

Solución:

```
mi_conjunto = set()
print(mi_conjunto)
```

Explicación: {} crea un diccionario vacío, no un conjunto. Para un conjunto vacío se usa `set()`.

Ejercicio 12

Enunciado: Convierte la lista `[1, 2, 2, 3, 3, 3]` en un conjunto para eliminar duplicados.

Solución:

```
lista = [1, 2, 2, 3, 3, 3]
conjunto = set(lista)
print(conjunto)
```

Explicación: `set(lista)` convierte la lista en un conjunto eliminando los duplicados automáticamente.

Ejercicio 13

Enunciado: Copia el conjunto `{1, 2, 3}` en otra variable llamada `copia`.

Solución:

```
original = {1, 2, 3}
copia = original.copy()
print(copia)
```

Explicación: `copy()` crea una copia independiente del conjunto original.

Ejercicio 14

Enunciado: Borra todos los elementos del conjunto `{1, 2, 3}`.

Solución:

```
A = {1, 2, 3}
A.clear()
print(A)
```

Explicación: `clear()` elimina todos los elementos del conjunto, dejándolo vacío.

Ejercicio 15

Enunciado: Añade todos los elementos del conjunto `{4, 5}` al conjunto `{1, 2, 3}`.

Solución:

```
A = {1, 2, 3}
B = {4, 5}
A.update(B)
print(A)
```

Explicación: `update()` añade todos los elementos de otro conjunto al conjunto original.

Ejercicio 16

Enunciado: Crea un `frozenset` con los días de la semana.

Solución:

```
dias = frozenset(["lunes", "martes", "miércoles", "jueves", "viernes", "sábado",  
"domingo"])  
print(dias)
```

Explicación: `frozenset` crea un conjunto inmutable que no puede ser modificado.

Ejercicio 17

Enunciado: Usa un conjunto para eliminar duplicados de la lista `["a", "b", "a", "c", "b"]`.

Solución:

```
lista = ["a", "b", "a", "c", "b"]  
sin_duplicados = list(set(lista))  
print(sin_duplicados)
```

Explicación: Se convierte la lista a conjunto para eliminar duplicados y luego se convierte de nuevo a lista.

Ejercicio 18

Enunciado: Encuentra los elementos que están en `{1, 2, 3, 4}` pero no en `{3, 4, 5, 6}`.

Solución:

```
A = {1, 2, 3, 4}  
B = {3, 4, 5, 6}  
diferencia = A - B  
print(diferencia)
```

Explicación: `-` devuelve los elementos de A que no están en B: `{1, 2}`.

Ejercicio 19

Enunciado: Encuentra todos los elementos únicos de los conjuntos `{1, 2, 3}` y `{2, 3, 4}`.

Solución:

```
A = {1, 2, 3}  
B = {2, 3, 4}  
unicos = A ^ B  
print(unicos)
```

Explicación: `^` devuelve los elementos que están en un conjunto o en otro, pero no en ambos: `{1, 4}`.

Ejercicio 20

Enunciado: Comprueba la longitud del conjunto $\{1, 2, 3, 4, 5\}$.

Solución:

```
A = {1, 2, 3, 4, 5}
print(len(A))
```

Explicación: `len( )` devuelve el número de elementos del conjunto.

9.2 Ejercicios de Python – Estructuras de Datos, Conjuntos (Nivel Medio)

Ejercicio 1

Enunciado: Dados los conjuntos $A = \{1, 2, 3, 4\}$ y $B = \{3, 4, 5, 6\}$, encuentra la unión de ambos y ordénala en una lista.

Solución:

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
union_ordenada = sorted(A | B)
print(union_ordenada)
```

Explicación: $A | B$ hace la unión, `sorted()` convierte el conjunto en lista ordenada.

Ejercicio 2

Enunciado: Encuentra los elementos que están en A o B, pero no en ambos (diferencia simétrica).

Solución:

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
diferencia_sim = A ^ B
print(diferencia_sim)
```

Explicación: El operador $\wedge$ devuelve los elementos únicos de cada conjunto.

Ejercicio 3

Enunciado: Comprueba si A es subconjunto de B ($A = \{1, 2\}$ y $B = \{1, 2, 3, 4\}$).

Solución:

```
A = {1, 2}
B = {1, 2, 3, 4}
print(A.issubset(B))
```

Explicación: `issubset()` devuelve `True` si todos los elementos de A están en B.

Ejercicio 4

Enunciado: Comprueba si B es superconjunto de A.

Solución:

```
A = {1, 2}
B = {1, 2, 3, 4}
print(B.issuperset(A))
```

Explicación: `issuperset()` devuelve True si B contiene todos los elementos de A.

Ejercicio 5

Enunciado: Encuentra los elementos que están en A pero no en B y viceversa, usando `difference()`.

Solución:

```
A = {1, 2, 3}
B = {2, 3, 4}
print("A-B:", A.difference(B))
print("B-A:", B.difference(A))
```

Explicación: `difference()` devuelve los elementos que están en un conjunto y no en el otro.

Ejercicio 6

Enunciado: Fusiona tres conjuntos $A = \{1, 2\}$, $B = \{2, 3\}$, $C = \{3, 4\}$ en uno solo.

Solución:

```
A = {1, 2}
B = {2, 3}
C = {3, 4}
fusion = A | B | C
print(fusion)
```

Explicación: Se pueden encadenar uniones con `|` para combinar varios conjuntos.

Ejercicio 7

Enunciado: Elimina todos los elementos de A que también están en B usando `difference_update()`.

Solución:

```
A = {1, 2, 3, 4}
B = {3, 4, 5}
A.difference_update(B)
print(A)
```

Explicación: `difference_update()` elimina del conjunto original los elementos presentes en otro conjunto.

Ejercicio 8

Enunciado: Mantén en A solo los elementos que también están en B usando `intersection_update()`.

Solución:

```
A = {1, 2, 3, 4}
B = {3, 4, 5}
A.intersection_update(B)
print(A)
```

Explicación: `intersection_update()` modifica el conjunto original para que contenga solo elementos comunes.

Ejercicio 9

Enunciado: Obtén el número de elementos únicos en la lista `[1, 2, 2, 3, 3, 3, 4]` usando conjuntos.

Solución:

```
lista = [1, 2, 2, 3, 3, 3, 4]
unicos = len(set(lista))
print(unicos)
```

Explicación: Convertir la lista en conjunto elimina duplicados; `len()` devuelve el total de elementos únicos.

Ejercicio 10

Enunciado: Comprueba si dos conjuntos $\{1, 2, 3\}$ y $\{3, 2, 1\}$ son iguales.

Solución:

```
A = {1, 2, 3}
B = {3, 2, 1}
print(A == B)
```

Explicación: Los conjuntos no tienen orden, por lo que estos dos conjuntos son iguales aunque el orden difiera.

Ejercicio 11

Enunciado: Combina los elementos de dos listas $[1, 2, 3]$ y $[3, 4, 5]$ en un conjunto sin duplicados.

Solución:

```
lista1 = [1, 2, 3]
lista2 = [3, 4, 5]
conjunto = set(lista1 + lista2)
print(conjunto)
```

Explicación: La suma de listas crea una lista combinada; convertirla en conjunto elimina duplicados automáticamente.

Ejercicio 12

Enunciado: Convierte un conjunto en lista ordenada $[3, 1, 4, 2]$ y calcula su suma.

Solución:

```
A = {3, 1, 4, 2}
lista_ordenada = sorted(A)
suma = sum(lista_ordenada)
print(lista_ordenada, suma)
```

Explicación: `sorted()` ordena el conjunto; `sum()` suma sus elementos.

Ejercicio 13

Enunciado: Elimina un elemento aleatorio de un conjunto `{1, 2, 3, 4, 5}` usando `pop()`.

Solución:

```
A = {1, 2, 3, 4, 5}
eliminado = A.pop()
print("Eliminado:", eliminado)
print("Conjunto:", A)
```

Explicación: `pop()` elimina y devuelve un elemento aleatorio. Los conjuntos no tienen orden, por eso no se sabe cuál se elimina.

Ejercicio 14

Enunciado: Comprueba si los conjuntos `{1, 2}` y `{2, 3}` son disjuntos (no tienen elementos en común).

Solución:

```
A = {1, 2}
B = {2, 3}
print(A.isdisjoint(B))
```

Explicación: `isdisjoint()` devuelve `True` si no hay elementos en común; en este caso devuelve `False`.

Ejercicio 15

Enunciado: Crea un `frozenset` y verifica que no puedes agregar elementos.

Solución:

```
dias = frozenset(["lunes", "martes"])
# dias.add("miércoles") -> Esto generaría un error
print(dias)
```

Explicación: `frozenset` es inmutable, por lo que no se pueden añadir ni eliminar elementos.

Ejercicio 16

Enunciado: Encuentra los elementos que están en A o B pero no en ambos, usando `symmetric_difference()`.

Solución:

```
A = {1, 2, 3}
B = {2, 3, 4}
sim = A.symmetric_difference(B)
print(sim)
```

Explicación: `symmetric_difference()` devuelve los elementos únicos de cada conjunto.

Ejercicio 17

Enunciado: Combina tres listas `[1, 2]`, `[2, 3]`, `[3, 4]` en un conjunto de elementos únicos.

Solución:

```
lista1 = [1, 2]
lista2 = [2, 3]
lista3 = [3, 4]
conjunto = set(lista1 + lista2 + lista3)
print(conjunto)
```

Explicación: La suma de listas une todas las listas; el conjunto elimina duplicados automáticamente.

Ejercicio 18

Enunciado: Dadas dos listas de nombres, encuentra los que aparecen en ambas usando conjuntos.

Solución:

```
nombres1 = ["Ana", "Luis", "Pedro"]
nombres2 = ["Luis", "Pedro", "Carlos"]
comunes = set(nombres1) & set(nombres2)
print(comunes)
```

Explicación: Convertir listas a conjuntos permite usar `&` para obtener la intersección.

Ejercicio 19

Enunciado: Elimina de un conjunto $\{1, 2, 3, 4\}$ los elementos presentes en otra lista $[2, 3]$.

Solución:

```
A = {1, 2, 3, 4}
B = [2, 3]
A.difference_update(B)
print(A)
```

Explicación: `difference_update()` elimina los elementos presentes en otro iterable.

Ejercicio 20

Enunciado: Ordena los elementos de $\{5, 3, 1, 4, 2\}$ y muestra el menor y mayor elemento.

Solución:

```
A = {5, 3, 1, 4, 2}
ordenada = sorted(A)
print("Ordenada:", ordenada)
print("Menor:", min(A))
print("Mayor:", max(A))
```

Explicación: `sorted()` devuelve la lista ordenada; `min()` y `max()` encuentran el valor mínimo y máximo del conjunto.

9.3 Ejercicios de Python – Estructuras de Datos, Conjuntos (Nivel Avanzado)

Ejercicio 1

Enunciado: Dados los conjuntos $A = \{1, 2, 3, 4, 5\}$ y $B = \{4, 5, 6, 7\}$, encuentra los elementos únicos de A y B que no estén en ambos (diferencia simétrica) y ordénalos descendientemente.

Solución:

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7}
resultado = sorted(A ^ B, reverse=True)
print(resultado)
```

Explicación: $\wedge$ devuelve la diferencia simétrica. `sorted(..., reverse=True)` ordena los elementos de mayor a menor.

Ejercicio 2

Enunciado: A partir de la lista `[1, 2, 2, 3, 3, 4, 5, 5]`, encuentra los elementos duplicados usando conjuntos.

Solución:

```
lista = [1, 2, 2, 3, 3, 4, 5, 5]
duplicados = set([x for x in lista if lista.count(x) > 1])
print(duplicados)
```

Explicación: Se usa comprensión de listas para contar las apariciones y se convierte en conjunto para obtener solo elementos únicos duplicados.

Ejercicio 3

Enunciado: Dadas dos listas de palabras, encuentra las palabras que aparecen exactamente en una de las listas, pero no en ambas.

Solución:

```
lista1 = ["python", "java", "c++"]
lista2 = ["java", "c#", "python"]
resultado = set(lista1) ^ set(lista2)
print(resultado)
```

Explicación: La diferencia simétrica $\wedge$ devuelve elementos exclusivos de cada conjunto.

Ejercicio 4

Enunciado: Comprueba si un conjunto $\{1, 2, 3\}$ es subconjunto estricto de $\{1, 2, 3, 4\}$.

Solución:

```
A = {1, 2, 3}
B = {1, 2, 3, 4}
print(A < B)
```

Explicación: El operador `<` devuelve `True` si A es un subconjunto propio de B.

Ejercicio 5

Enunciado: Encuentra los elementos comunes de tres conjuntos $A=\{1, 2, 3\}$, $B=\{2, 3, 4\}$, $C=\{3, 4, 5\}$.

Solución:

```
A = {1, 2, 3}
B = {2, 3, 4}
C = {3, 4, 5}
interseccion = A & B & C
print(interseccion)
```

Explicación: Se pueden encadenar `&` para encontrar elementos comunes entre múltiples conjuntos.

Ejercicio 6

Enunciado: A partir de una lista $[1, 2, 3, 4, 5, 6, 7, 8, 9]$, crea un conjunto de números pares y otro de impares, luego obtén la diferencia entre pares e impares.

Solución:

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
pares = {x for x in lista if x % 2 == 0}
impares = {x for x in lista if x % 2 != 0}
diferencia = pares - impares
print(diferencia)
```

Explicación: Se usan **conjuntos por comprensión** para filtrar pares e impares y `-` para la diferencia.

Ejercicio 7

Enunciado: Combina tres listas de estudiantes y elimina duplicados usando conjuntos.

Solución:

```
lista1 = ["Ana", "Luis"]
lista2 = ["Luis", "Pedro"]
lista3 = ["Pedro", "Maria"]
todos = set(lista1 + lista2 + lista3)
print(todos)
```

Explicación: La suma de listas crea una lista unificada; `set()` elimina duplicados automáticamente.

Ejercicio 8

Enunciado: Dadas dos listas de enteros, encuentra los elementos que aparecen al menos dos veces en ambas listas.

Solución:

```
lista1 = [1,2,2,3,4]
lista2 = [2,2,3,5]
resultado = set([x for x in lista1 if lista1.count(x) > 1]) & set([x for x in lista2 if lista2.count(x) > 1])
print(resultado)
```

Explicación: Se crean conjuntos de elementos duplicados en cada lista y se intersectan para obtener los que se repiten en ambas.

Ejercicio 9

Enunciado: Encuentra los números primos en un conjunto $\{2, 3, 4, 5, 6, 7, 8, 9, 10\}$ usando conjuntos.

Solución:

```
numeros = {2,3,4,5,6,7,8,9,10}
primos = {x for x in numeros if all(x % i != 0 for i in range(2, x))}
print(primos)
```

Explicación: Se utiliza **conjunto por comprensión** y la función `all()` para filtrar los números primos.

Ejercicio 10

Enunciado: Dadas dos cadenas "abracadabra" y "alakazam", encuentra las letras comunes.

Solución:

```
s1 = set("abracadabra")
s2 = set("alakazam")
comunes = s1 & s2
print(comunes)
```

Explicación: Convertir cadenas en conjuntos permite encontrar letras únicas y comunes usando intersección &.

Ejercicio 11

Enunciado: Encuentra los elementos únicos de una lista de listas `[[1, 2], [2, 3], [3, 4]]`.

Solución:

```
listas = [[1, 2], [2, 3], [3, 4]]
unicos = set()
for sub in listas:
    unicos.update(sub)
print(unicos)
```

Explicación: `update()` añade todos los elementos de un iterable al conjunto, combinando varias sublistas en un conjunto único.

Ejercicio 12

Enunciado: Comprueba si dos conjuntos `{1, 2, 3}` y `{3, 4, 5}` son disjuntos.

Solución:

```
A = {1, 2, 3}
B = {3, 4, 5}
print(A.isdisjoint(B))
```

Explicación: `isdisjoint()` devuelve `True` si no hay elementos en común; aquí devuelve `False`.

Ejercicio 13

Enunciado: Ordena los elementos de $\{5, 1, 3, 2, 4\}$ en una lista descendente y elimina duplicados.

Solución:

```
A = {5, 1, 3, 2, 4}
ordenada = sorted(A, reverse=True)
print(ordenada)
```

Explicación: Los conjuntos eliminan duplicados automáticamente; `sorted(..., reverse=True)` ordena descendente.

Ejercicio 14

Enunciado: Calcula la diferencia simétrica de múltiples conjuntos $\{1, 2, 3\}$, $\{3, 4, 5\}$, $\{5, 6, 1\}$.

Solución:

```
A = {1, 2, 3}
B = {3, 4, 5}
C = {5, 6, 1}
resultado = A ^ B ^ C
print(resultado)
```

Explicación: $\wedge$ es asociativo, por lo que puede encadenarse para múltiples conjuntos.

Ejercicio 15

Enunciado: Convierte un diccionario $\{ "a": 1, "b": 2, "c": 3 \}$ en un conjunto de sus claves.

Solución:

```
d = {"a": 1, "b": 2, "c": 3}
claves = set(d.keys())
print(claves)
```

Explicación: `dict.keys()` devuelve un iterable con las claves; `set()` lo convierte en conjunto.

Ejercicio 16

Enunciado: Encuentra todas las letras únicas en la frase "el rápido zorro marrón" ignorando espacios.

Solución:

```
frase = "el rápido zorro marrón"
letras_unicas = set(frase.replace(" ", ""))
print(letras_unicas)
```

Explicación: `replace(" ", "")` elimina espacios; `set()` obtiene letras únicas.

Ejercicio 17

Enunciado: De la lista `[1, 2, 3, 4, 5]`, elimina todos los elementos que sean múltiplos de 2 o 3 usando conjuntos.

Solución:

```
lista = [1, 2, 3, 4, 5]
resultado = {x for x in lista if x % 2 != 0 and x % 3 != 0}
print(resultado)
```

Explicación: Se usa **conjunto por comprensión** con condiciones para filtrar los elementos.

Ejercicio 18

Enunciado: De dos listas `[1, 2, 3, 4]` y `[3, 4, 5, 6]`, crea un diccionario donde las claves sean los elementos comunes y el valor sea la suma de su índice en cada lista.

Solución:

```
lista1 = [1, 2, 3, 4]
lista2 = [3, 4, 5, 6]
comunes = set(lista1) & set(lista2)
dic = {x: lista1.index(x) + lista2.index(x) for x in comunes}
print(dic)
```

Explicación: `&` obtiene elementos comunes. `index()` obtiene la posición de cada elemento; se crea un diccionario por comprensión.

Ejercicio 19

Enunciado: A partir de la lista `["a", "b", "c", "a", "b"]`, crea un conjunto de elementos únicos y calcula la frecuencia de cada uno en la lista original.

Solución:

```
lista = ["a", "b", "c", "a", "b"]
unicos = set(lista)
frecuencia = {x: lista.count(x) for x in unicos}
print(frecuencia)
```

Explicación: Se obtiene conjunto de elementos únicos y luego un diccionario que cuenta la frecuencia de cada elemento.

Ejercicio 20

Enunciado: Dadas las listas `[1, 2, 3]`, `[2, 3, 4]`, `[3, 4, 5]`, encuentra los elementos que aparecen en exactamente dos listas.

Solución:

```
listas = [{1, 2, 3}, {2, 3, 4}, {3, 4, 5}]
contador = {}
for s in listas:
    for x in s:
        contador[x] = contador.get(x, 0) + 1
resultado = {k for k, v in contador.items() if v == 2}
print(resultado)
```

Explicación: Se usa un diccionario para contar apariciones de cada elemento en todas las listas. Luego, se filtran los que aparecen exactamente dos veces.

Lección 6.: Estructuras de Datos – Tuplas en Python

1. Introducción a las Tuplas

En Python, las **tuplas** son estructuras de datos que permiten almacenar **una colección de elementos ordenados e inmutables**. Se representan mediante paréntesis () y los elementos se separan por comas.

Sintaxis básica:

```
mi_tupla = (1, 2, 3, 4)
print(mi_tupla) # Salida: (1, 2, 3, 4)
```

Características principales:

1. **Ordenadas:** Los elementos tienen un índice, empezando en 0.
 2. **Inmutables:** Una vez creada una tupla, no se puede modificar (agregar, eliminar o cambiar elementos directamente).
 3. **Permite duplicados:** Los elementos pueden repetirse.
 4. **Puede contener distintos tipos de datos:** Números, cadenas, booleanos, listas, otras tuplas, etc.
-

2. Creación de tuplas

2.1 Tupla vacía

```
tupla_vacia = ()
print(tupla_vacia) # Salida: ()
```

2.2 Tupla con un solo elemento

Para crear una tupla con **un solo elemento**, es obligatorio usar una coma:

```
tupla_un_elemento = (5,)
print(type(tupla_un_elemento)) # Salida: <class 'tuple'>
```

```
no_tupla = (5)
print(type(no_tupla)) # Salida: <class 'int'>
```

2.3 Tupla con múltiples elementos

```
tupla_mixta = (1, "hola", 3.14, True)
print(tupla_mixta) # Salida: (1, 'hola', 3.14, True)
```

3. Acceso a elementos

Los elementos de una tupla se acceden usando **índices**, de forma similar a las listas:

```
tupla = (10, 20, 30, 40, 50)
print(tupla[0]) # Primer elemento: 10
print(tupla[2]) # Tercer elemento: 30
print(tupla[-1]) # Último elemento: 50
```

También se puede acceder a **subtuplas** mediante *slicing*:

```
print(tupla[1:4]) # Salida: (20, 30, 40)
print(tupla[:3])  # Salida: (10, 20, 30)
print(tupla[2:])  # Salida: (30, 40, 50)
```

4. Operaciones básicas con tuplas

4.1 Concatenación

Se pueden unir tuplas usando +:

```
t1 = (1, 2)
t2 = (3, 4)
t3 = t1 + t2
print(t3) # Salida: (1, 2, 3, 4)
```

4.2 Repetición

Se pueden repetir elementos con *:

```
t = (1, 2)
print(t * 3) # Salida: (1, 2, 1, 2, 1, 2)
```

4.3 Verificar pertenencia

Usando in:

```
t = (1, 2, 3)
print(2 in t) # True
print(5 in t) # False
```

5. Funciones y métodos útiles con tuplas

Aunque las tuplas son inmutables, Python ofrece algunas funciones para trabajar con ellas:

1. **len()** – Devuelve la longitud:

```
t = (1, 2, 3)
print(len(t)) # Salida: 3
```

2. **max()** y **min()** – Devuelve el máximo y mínimo (solo con elementos comparables):

```
t = (10, 20, 5, 40)
print(max(t)) # 40
```

```
print(min(t)) # 5
```

3. **sum()** – Suma los elementos numéricos:

```
t = (1, 2, 3)
print(sum(t)) # 6
```

4. **index()** – Devuelve la posición de un elemento:

```
t = (10, 20, 30)
print(t.index(20)) # 1
```

5. **count()** – Cuenta cuántas veces aparece un elemento:

```
t = (1, 2, 2, 3)
print(t.count(2)) # 2
```

6. Inmutabilidad y conversiones

Aunque no se pueden modificar las tuplas directamente, se pueden:

- **Convertir en listas**, modificar y volver a tupla:

```
t = (1, 2, 3)
lista = list(t)
lista.append(4)
t = tuple(lista)
print(t) # Salida: (1, 2, 3, 4)
```

- **Eliminar una tupla** completa:

```
t = (1, 2, 3)
del t
# print(t) # Esto dará error porque la tupla ya no existe
```

7. Tuplas anidadas

Las tuplas pueden contener otras tuplas:

```
t = (1, 2, (3, 4, 5), 6)
print(t[2]) # Salida: (3, 4, 5)
print(t[2][1]) # Salida: 4
```

Esto permite estructurar datos complejos, como coordenadas o matrices inmutables.

8. Tuplas vs Listas

Característica	Tupla	Lista
Mutabilidad	Inmutable	Mutable
Sintaxis	(a, b, c)	[a, b, c]
Velocidad	Más rápida	Más lenta
Uso recomendado	Datos constantes	Datos que cambian

Ejemplo práctico: Si quieres representar los días de la semana, es mejor usar una tupla:

```
dias = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado", "domingo")
```

9. Desempaquetado de tuplas

El **desempaquetado** permite asignar elementos de una tupla a variables:

```
t = (1, 2, 3)
a, b, c = t
print(a, b, c) # 1 2 3
```

Con el operador `*` también se puede desempaquetar parcialmente:

```
t = (1, 2, 3, 4, 5)
a, *b, c = t
print(a) # 1
print(b) # [2, 3, 4]
print(c) # 5
```

10. Aplicaciones prácticas de las tuplas

1. Devolver múltiples valores en funciones:

```
def operaciones(a, b):
    return a+b, a-b, a*b

suma, resta, producto = operaciones(5, 3)
print(suma, resta, producto) # 8 2 15
```

2. Claves de diccionarios: Las tuplas se pueden usar como claves porque son inmutables:

```
coordenadas = {(0, 0): "origen", (1, 2): "punto A"}
print(coordenadas[(1, 2)]) # punto A
```

3. Datos constantes: Para representar información que no debe cambiar, como días de la semana, colores fijos, etc.

11. Buenas prácticas

- Usa tuplas cuando los datos **no deban cambiar**.
 - Para colecciones grandes y estáticas, las tuplas son **más eficientes** que las listas.
 - Utiliza el **desempaquetado** para código más limpio y legible.
 - Para estructuras de datos complejas, combina tuplas con listas o diccionarios según la necesidad.
-

12. Resumen rápido

- Tupla: colección **ordenada e inmutable**.
 - Sintaxis: (elemento1, elemento2, ...).
 - Acceso mediante índices y slicing.
 - Métodos: `count()`, `index()`.
 - Operaciones: concatenación `+`, repetición `*`, pertenencia `in`.
 - Útil para datos constantes y como claves de diccionarios.
 - Diferencia clave con listas: las listas son **mutables**, las tuplas **inmutables**.
-

13. Ejercicios

13.1 Ejercicios de Python – Estructuras de Datos, Tuplas (Nivel Principiante)

Ejercicio 1

Enunciado: Crea una tupla llamada `numeros` con los elementos 1, 2, 3, 4 y 5.

Solución:

```
numeros = (1, 2, 3, 4, 5)
print(numeros)
```

Explicación: Se utiliza la sintaxis (`elemento1, elemento2, ...`) para crear una tupla.

Ejercicio 2

Enunciado: Imprime el primer elemento de la tupla `numeros = (10, 20, 30)`.

Solución:

```
numeros = (10, 20, 30)
print(numeros[0])
```

Explicación: Los elementos de una tupla se acceden por índice, comenzando en 0.

Ejercicio 3

Enunciado: Verifica si el número 5 está en la tupla `numeros = (1, 2, 3, 4, 5)`.

Solución:

```
numeros = (1, 2, 3, 4, 5)
print(5 in numeros)
```

Explicación: El operador `in` devuelve `True` si el elemento está presente.

Ejercicio 4

Enunciado: Crea una tupla de un solo elemento 7 y verifica su tipo.

Solución:

```
tupla_uno = (7,)
print(type(tupla_uno))
```

Explicación: Para tuplas de un elemento es obligatorio poner la coma, de lo contrario Python interpreta el paréntesis como un número entero.

Ejercicio 5

Enunciado: Imprime la longitud de la tupla $t = (5, 10, 15, 20)$.

Solución:

```
t = (5, 10, 15, 20)
print(len(t))
```

Explicación: La función `len()` devuelve el número de elementos de la tupla.

Ejercicio 6

Enunciado: Devuelve la última posición de la tupla $t = (3, 6, 9, 12)$ usando índices negativos.

Solución:

```
t = (3, 6, 9, 12)
print(t[-1])
```

Explicación: `-1` accede al último elemento de la tupla.

Ejercicio 7

Enunciado: Concatenar dos tuplas: $a = (1, 2)$ y $b = (3, 4)$.

Solución:

```
a = (1, 2)
b = (3, 4)
c = a + b
print(c)
```

Explicación: El operador `+` une dos tuplas formando una nueva.

Ejercicio 8

Enunciado: Repetir la tupla $t = (0, 1)$ tres veces.

Solución:

```
t = (0, 1)
print(t * 3)
```

Explicación: El operador `*` repite la tupla el número de veces indicado.

Ejercicio 9

Enunciado: Cuenta cuántas veces aparece el número 2 en la tupla `t = (1, 2, 2, 3, 2)`.

Solución:

```
t = (1, 2, 2, 3, 2)
print(t.count(2))
```

Explicación: El método `count()` devuelve el número de veces que un elemento aparece en la tupla.

Ejercicio 10

Enunciado: Encuentra la posición del número 3 en `t = (5, 3, 7, 3)`.

Solución:

```
t = (5, 3, 7, 3)
print(t.index(3))
```

Explicación: `index()` devuelve la primera posición donde se encuentra el elemento.

Ejercicio 11

Enunciado: Accede a los elementos 2 y 3 de `t = (1, 2, 3, 4, 5)` usando slicing.

Solución:

```
t = (1, 2, 3, 4, 5)
print(t[1:3])
```

Explicación: `t[inicio:fin]` devuelve una subtupla desde el índice `inicio` hasta `fin-1`.

Ejercicio 12

Enunciado: Convierte la tupla `t = (1, 2, 3)` en lista y agrega el número 4.

Solución:

```
t = (1, 2, 3)
lista = list(t)
lista.append(4)
print(lista)
```

Explicación: Las tuplas son inmutables, pero podemos convertirlas a lista, modificar y luego convertir de nuevo si queremos.

Ejercicio 13

Enunciado: Crea una tupla anidada: `((1, 2), (3, 4))` y accede al número 4.

Solución:

```
t = ((1, 2), (3, 4))  
print(t[1][1])
```

Explicación: Se accede primero a la sub-tupla con `t[1]` y luego al elemento con `t[1][1]`.

Ejercicio 14

Enunciado: Desempaqueta la tupla `t = (10, 20, 30)` en tres variables `a`, `b`, `c`.

Solución:

```
t = (10, 20, 30)  
a, b, c = t  
print(a, b, c)
```

Explicación: El desempaqueado asigna cada elemento a una variable correspondiente.

Ejercicio 15

Enunciado: Crea una tupla con diferentes tipos de datos: `1, "hola", True, 3.14`.

Solución:

```
t = (1, "hola", True, 3.14)  
print(t)
```

Explicación: Las tuplas pueden almacenar números, cadenas, booleanos, etc., incluso mezclados.

Ejercicio 16

Enunciado: Suma todos los elementos de la tupla `t = (5, 10, 15)`.

Solución:

```
t = (5, 10, 15)  
print(sum(t))
```

Explicación: La función `sum()` calcula la suma de elementos numéricos de una tupla.

Ejercicio 17

Enunciado: Encuentra el valor máximo y mínimo de `t = (8, 3, 7, 1)`.

Solución:

```
t = (8, 3, 7, 1)
print(max(t), min(t))
```

Explicación: `max()` devuelve el mayor y `min()` el menor valor de la tupla.

Ejercicio 18

Enunciado: Crea una tupla con los días de la semana.

Solución:

```
dias = ("lunes", "martes", "miércoles", "jueves", "viernes", "sábado",
"domingo")
print(dias)
```

Explicación: Es recomendable usar tuplas para datos constantes que no deben cambiar.

Ejercicio 19

Enunciado: Usa una tupla como clave de un diccionario: `coordenadas = {(0,0): "origen"}`.

Solución:

```
coordenadas = {(0,0): "origen"}
print(coordenadas[(0,0)])
```

Explicación: Las tuplas son inmutables y pueden usarse como claves de diccionarios.

Ejercicio 20

Enunciado: Devuelve múltiples valores desde una función usando una tupla.

Solución:

```
def operaciones(a, b):
    return a+b, a-b, a*b

suma, resta, producto = operaciones(5, 3)
print(suma, resta, producto)
```

Explicación: Las funciones pueden retornar tuplas para devolver varios valores a la vez, que se pueden desempaquetar en variables.

13.2 Ejercicios de Python – Estructuras de Datos, Tuplas (Nivel Medio)

Ejercicio 1

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5)`, crea una nueva tupla que contenga solo los elementos pares.

Solución:

```
t = (1, 2, 3, 4, 5)
t_pares = tuple(x for x in t if x % 2 == 0)
print(t_pares)
```

Explicación: Usamos comprensión de tuplas (convertida desde lista por `tuple()`) para filtrar solo los números pares.

Ejercicio 2

Enunciado: Invierte la tupla `t = (10, 20, 30, 40)`.

Solución:

```
t = (10, 20, 30, 40)
t_invertida = t[::-1]
print(t_invertida)
```

Explicación: El slicing con `[::-1]` devuelve la tupla en orden inverso.

Ejercicio 3

Enunciado: Dada la tupla `t = ((1,2), (3,4), (5,6))`, suma los elementos de cada sub-tupla y crea una tupla con los resultados.

Solución:

```
t = ((1,2), (3,4), (5,6))
suma_sub = tuple(sum(x) for x in t)
print(suma_sub)
```

Explicación: Iteramos por cada sub-tupla y usamos `sum()` para obtener la suma de sus elementos.

Ejercicio 4

Enunciado: Dada la tupla `t = (5, 3, 8, 2)`, encuentra el índice del mayor valor sin usar `index()`.

Solución:

```
t = (5, 3, 8, 2)
max_valor = max(t)
indice = [i for i, v in enumerate(t) if v == max_valor][0]
print(indice)
```

Explicación: Usamos `enumerate()` para obtener índices y valores, luego filtramos para encontrar el índice del mayor.

Ejercicio 5

Enunciado: Combina las tuplas `a = (1, 2)` y `b = ('x', 'y')` en una tupla de tuplas usando `zip()`.

Solución:

```
a = (1, 2)
b = ('x', 'y')
combinada = tuple(zip(a, b))
print(combinada)
```

Explicación: `zip()` combina los elementos de ambas tuplas en pares, y `tuple()` convierte el resultado en tupla.

Ejercicio 6

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5, 6)`, separa el primer elemento, los del medio y el último usando desempaqueado avanzado.

Solución:

```
t = (1, 2, 3, 4, 5, 6)
primero, *medio, ultimo = t
print(primero, medio, ultimo)
```

Explicación: El operador `*` permite capturar los elementos intermedios en una lista durante el desempaqueado.

Ejercicio 7

Enunciado: Dada la tupla `t = (1, 2, 3, 2, 4, 2)`, reemplaza todas las ocurrencias de 2 por 9 **sin modificar la tupla original**.

Solución:

```
t = (1, 2, 3, 2, 4, 2)
t_nueva = tuple(9 if x==2 else x for x in t)
print(t_nueva)
```

Explicación: Las tuplas son inmutables, pero podemos crear una nueva usando comprensión y condicional.

Ejercicio 8

Enunciado: Dada la tupla `t = ((1, 2), (3, 4), (5, 6))`, crea una lista que contenga todos los elementos individuales.

Solución:

```
t = ((1, 2), (3, 4), (5, 6))
lista = [x for sub in t for x in sub]
print(lista)
```

Explicación: Usamos comprensión de listas con doble bucle para "aplanar" la tupla anidada.

Ejercicio 9

Enunciado: Ordena los elementos de la tupla `t = (4, 1, 7, 3)` de menor a mayor.

Solución:

```
t = (4, 1, 7, 3)
t_ordenada = tuple(sorted(t))
print(t_ordenada)
```

Explicación: `sorted()` devuelve una lista ordenada; convertimos a tupla usando `tuple()`.

Ejercicio 10

Enunciado: Dada la tupla `t = (1, 2, 3, 4)`, crea un diccionario donde la clave sea el elemento y el valor sea su cuadrado.

Solución:

```
t = (1, 2, 3, 4)
dic = {x: x**2 for x in t}
print(dic)
```

Explicación: Usamos comprensión de diccionarios con la tupla como fuente de claves.

Ejercicio 11

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5, 6)`, crea dos tuplas: una con los elementos en posiciones pares y otra con las impares.

Solución:

```
t = (1, 2, 3, 4, 5, 6)
pares = t[::2]
impares = t[1::2]
print(pares, impares)
```

Explicación: El slicing `::2` selecciona cada segundo elemento, comenzando desde 0 o 1.

Ejercicio 12

Enunciado: Dada la tupla `t = ("a", "b", "c", "a", "b")`, elimina duplicados creando una nueva tupla.

Solución:

```
t = ("a", "b", "c", "a", "b")
t_nueva = tuple(dict.fromkeys(t))
print(t_nueva)
```

Explicación: `dict.fromkeys()` elimina duplicados y mantiene el orden; convertimos el resultado en tupla.

Ejercicio 13

Enunciado: Une las tuplas `t1 = (1, 2)` y `t2 = (3, 4)` en una sola tupla y calcula la suma de todos los elementos.

Solución:

```
t1 = (1, 2)
t2 = (3, 4)
t_total = t1 + t2
print(sum(t_total))
```

Explicación: Primero concatenamos con `+`, luego usamos `sum()` para obtener la suma total.

Ejercicio 14

Enunciado: Dada la tupla `t = (5, 3, 8, 2)`, devuelve el segundo mayor elemento.

Solución:

```
t = (5, 3, 8, 2)
t_orden = sorted(t, reverse=True)
print(t_orden[1])
```

Explicación: Ordenamos de mayor a menor y accedemos al segundo elemento.

Ejercicio 15

Enunciado: Dada la tupla `t = ((1, 2, 3), (4, 5, 6))`, crea una tupla con la suma de cada sub-tupla.

Solución:

```
t = ((1, 2, 3), (4, 5, 6))
t_sumas = tuple(sum(sub) for sub in t)
print(t_sumas)
```

Explicación: Iteramos por cada sub-tupla y usamos `sum()` para obtener la suma de sus elementos.

Ejercicio 16

Enunciado: Dada la tupla `t = ("Python", "es", "divertido")`, crea una nueva tupla con las longitudes de cada palabra.

Solución:

```
t = ("Python", "es", "divertido")
longitudes = tuple(len(palabra) for palabra in t)
print(longitudes)
```

Explicación: Usamos `len()` para calcular la longitud de cada elemento y generamos una nueva tupla.

Ejercicio 17

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5)`, intercambia el primer y último elemento creando una nueva tupla.

Solución:

```
t = (1, 2, 3, 4, 5)
t_nueva = (t[-1],) + t[1:-1] + (t[0],)
print(t_nueva)
```

Explicación: Como las tuplas son inmutables, creamos una nueva con los elementos reorganizados.

Ejercicio 18

Enunciado: Dada la tupla `t = ((1, 2), (3, 4), (5, 6))`, aplanar todos los elementos en una sola tupla.

Solución:

```
t = ((1, 2), (3, 4), (5, 6))
t_aplanada = tuple(x for sub in t for x in sub)
print(t_aplanada)
```

Explicación: Comprensión de tuplas con doble bucle para extraer todos los elementos individuales.

Ejercicio 19

Enunciado: Dada la tupla `t = (10, 20, 30, 40)`, crea una tupla de booleanos indicando si cada elemento es mayor que 25.

Solución:

```
t = (10, 20, 30, 40)
t_bool = tuple(x > 25 for x in t)
print(t_bool)
```

Explicación: Usamos expresión condicional dentro de `tuple()` para generar la tupla de valores booleanos.

Ejercicio 20

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5)`, intercala 0 entre cada elemento creando una nueva tupla.

Solución:

```
t = (1, 2, 3, 4, 5)
t_nueva = tuple(x for item in t for x in (item, 0))[:-1]
print(t_nueva)
```

Explicación: Usamos comprensión con doble bucle para insertar 0 después de cada elemento; `[:-1]` elimina el último 0 extra.

13. 3 Ejercicios de Python – Estructuras de Datos, Tuplas (Nivel Avanzado)

Ejercicio 1

Enunciado: Dada la tupla `t = ((1, 2), (3, 4), (5, 6))`, intercambia los elementos de cada sub-tupla y crea una nueva tupla.

Solución:

```
t = ((1, 2), (3, 4), (5, 6))
t_nueva = tuple((b, a) for a, b in t)
print(t_nueva)
```

Explicación: Se itera sobre cada sub-tupla y se invierte el orden de sus elementos usando desempaquetado.

Ejercicio 2

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5, 6)`, crea una tupla de pares (elemento, cuadrado) para cada número.

Solución:

```
t = (1, 2, 3, 4, 5, 6)
t_cuadrados = tuple((x, x**2) for x in t)
print(t_cuadrados)
```

Explicación: Se crea una tupla de tuplas usando comprensión, donde cada elemento es acompañado de su cuadrado.

Ejercicio 3

Enunciado: Dada la tupla `t = ((1, 2), (3, 4), (5, 6))`, aplanarla y ordenar los elementos en una nueva tupla.

Solución:

```
t = ((1, 2), (3, 4), (5, 6))
t_aplanada = tuple(sorted(x for sub in t for x in sub))
print(t_aplanada)
```

Explicación: Se usa comprensión con doble bucle para aplanar y `sorted()` para ordenar.

Ejercicio 4

Enunciado: Dada la tupla `t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))`, intercambia las filas y columnas (transposición).

Solución:

```
t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
t_transpuesta = tuple(zip(*t))
print(t_transpuesta)
```

Explicación: `zip(*t)` realiza la transposición de la tupla anidada.

Ejercicio 5

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5, 6)`, crear dos tuplas: una con números pares y otra con números impares usando `filter`.

Solución:

```
t = (1, 2, 3, 4, 5, 6)
pares = tuple(filter(lambda x: x%2==0, t))
impares = tuple(filter(lambda x: x%2!=0, t))
print(pares, impares)
```

Explicación: `filter()` permite filtrar elementos según una condición.

Ejercicio 6

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5)`, crea una tupla acumulativa de sumas `(1, 3, 6, 10, 15)`.

Solución:

```
t = (1, 2, 3, 4, 5)
acumulativa = tuple(sum(t[:i+1]) for i in range(len(t)))
print(acumulativa)
```

Explicación: Para cada índice, sumamos los elementos desde el inicio hasta ese índice.

Ejercicio 7

Enunciado: Dada la tupla `t = ((1, 2), (3, 4), (5, 6))`, crea un diccionario donde la clave sea el primer elemento de cada sub-tupla y el valor el segundo.

Solución:

```
t = ((1, 2), (3, 4), (5, 6))
dic = {a:b for a,b in t}
print(dic)
```

Explicación: Se usa comprensión de diccionarios con desempaquetado de sub-tuplas.

Ejercicio 8

Enunciado: Dada la tupla `t = ("a", 1), ("b", 2), ("c", 3))`, invierte cada par y crea una nueva tupla.

Solución:

```
t = ("a", 1), ("b", 2), ("c", 3)
t_nueva = tuple((b,a) for a,b in t)
print(t_nueva)
```

Explicación: Usamos desempaquetado y comprensión para invertir cada par.

Ejercicio 9

Enunciado: Dada la tupla `t = ((1, 2, 3), (4, 5, 6))`, suma elemento a elemento de cada sub-tupla formando una nueva tupla `(5, 7, 9)`.

Solución:

```
t = ((1, 2, 3), (4, 5, 6))
t_suma = tuple(sum(x) for x in zip(*t))
print(t_suma)
```

Explicación: `zip(*t)` agrupa elementos por posición y `sum()` suma los elementos correspondientes.

Ejercicio 10

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5)`, crea una tupla que contenga los elementos en posiciones pares y su doble.

Solución:

```
t = (1, 2, 3, 4, 5)
t_doble = tuple((x, x*2) for x in t[::2])
print(t_doble)
```

Explicación: Se seleccionan los elementos en posiciones pares y se forma una tupla de pares (elemento, doble).

Ejercicio 11

Enunciado: Dada la tupla `t = ((1, 2), (3, 4), (5, 6))`, aplanar y crear una tupla de pares (elemento, posición).

Solución:

```
t = ((1, 2), (3, 4), (5, 6))
t_pares = tuple((x, i) for i, x in enumerate(x for sub in t for x in sub))
print(t_pares)
```

Explicación: Se usa doble bucle para aplanar y `enumerate()` para obtener la posición.

Ejercicio 12

Enunciado: Dada la tupla `t = (10, 20, 30, 40, 50)`, crea una tupla con la diferencia entre cada elemento y el siguiente.

Solución:

```
t = (10, 20, 30, 40, 50)
diferencias = tuple(t[i+1]-t[i] for i in range(len(t)-1))
print(diferencias)
```

Explicación: Iteramos por índices y calculamos la diferencia entre elemento siguiente y actual.

Ejercicio 13

Enunciado: Dada la tupla `t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))`, calcula la suma diagonal (1+5+9).

Solución:

```
t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
diagonal = sum(t[i][i] for i in range(len(t)))
print(diagonal)
```

Explicación: Iteramos por los índices de las filas y columnas para obtener la diagonal principal.

Ejercicio 14

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5, 6)`, crear una tupla de tuplas con todos los pares posibles (a, b) donde $a < b$.

Solución:

```
t = (1, 2, 3, 4, 5, 6)
pares_posibles = tuple((t[i], t[j]) for i in range(len(t)) for j in range(i+1, len(t)))
print(pares_posibles)
```

Explicación: Usamos doble bucle para combinar elementos sin repetir y respetando $a < b$.

Ejercicio 15

Enunciado: Dada la tupla `t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))`, suma de columnas usando `zip`.

Solución:

```
t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
suma_columnas = tuple(sum(col) for col in zip(*t))
print(suma_columnas)
```

Explicación: `zip(*t)` agrupa elementos por columnas; luego sumamos cada columna.

Ejercicio 16

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5)`, crea una tupla donde cada elemento sea `(elemento, factorial(elemento))`.

Solución:

```
import math
t = (1, 2, 3, 4, 5)
t_factorial = tuple((x, math.factorial(x)) for x in t)
print(t_factorial)
```

Explicación: Se calcula el factorial de cada elemento con `math.factorial` y se genera la tupla de pares.

Ejercicio 17

Enunciado: Dada la tupla `t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))`, crea una tupla con la suma de cada fila.

Solución:

```
t = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
suma_filas = tuple(sum(fila) for fila in t)
print(suma_filas)
```

Explicación: Se itera por cada sub-tupla (fila) y se calcula la suma.

Ejercicio 18

Enunciado: Dada la tupla `t = (1, 2, 3, 4, 5, 6, 7, 8, 9)`, reorganiza en tupla de 3 sub-tuplas de 3 elementos cada una.

Solución:

```
t = (1, 2, 3, 4, 5, 6, 7, 8, 9)
t_reorganizada = tuple(t[i:i+3] for i in range(0, len(t), 3))
print(t_reorganizada)
```

Explicación: Se usa slicing en pasos de 3 para crear sub-tuplas.

Ejercicio 19

Enunciado: Dada la tupla `t = ("python", "java", "c++")`, crea una tupla con la primera letra de cada palabra.

Solución:

```
t = ("python", "java", "c++")
prim_letra = tuple(x[0] for x in t)
print(prim_letra)
```

Explicación: Se itera sobre la tupla y se selecciona la primera letra de cada elemento.

Ejercicio 20

Enunciado: Dada la tupla `t = ((1, 2), (3, 4), (5, 6))`, crea una tupla que contenga los elementos de forma alternada `(1, 3, 5, 2, 4, 6)`.

Solución:

```
t = ((1, 2), (3, 4), (5, 6))
alternada = tuple(x[i] for i in range(2) for x in t)
print(alternada)
```

Explicación: Iteramos primero por posición dentro de cada sub-tupla, luego por cada sub-tupla para reorganizar elementos alternados.

Lección 7.: Diccionarios en Python

1. Introducción a los Diccionarios

Un **diccionario** es una estructura de datos en Python que almacena **pares de clave-valor**.

- Cada **clave** debe ser única e inmutable (por ejemplo, cadenas, números o tuplas).
- Cada **valor** puede ser cualquier objeto de Python, incluyendo listas, otros diccionarios, funciones, etc.

Se usan para **asociar información** de forma rápida y eficiente.

Sintaxis básica:

```
mi_diccionario = {  
    "nombre": "Ana",  
    "edad": 25,  
    "ciudad": "Zaragoza"  
}
```

Aquí:

- "nombre", "edad" y "ciudad" son **claves**.
- "Ana", 25 y "Zaragoza" son los **valores** asociados.

Nota: Los diccionarios no mantienen orden antes de Python 3.7; desde 3.7, mantienen el orden de inserción.

2. Crear Diccionarios

2.1 Diccionario vacío

```
dic_vacio = {}  
dic_vacio2 = dict()
```

2.2 Diccionario con elementos

```
persona = {"nombre": "Violeta", "edad": 22}
```

2.3 Diccionario con claves de diferentes tipos

```
datos = {  
    1: "uno",  
    (2, 3): "tupla",  
    True: "booleano"  
}
```

3. Acceder a los valores

Para acceder a un valor, usamos la clave entre corchetes o el método `get()`.

```
persona = {"nombre": "Ana", "edad": 25}

# Usando corchetes
print(persona["nombre"]) # Ana

# Usando get()
print(persona.get("edad")) # 25

# get() con valor por defecto
print(persona.get("altura", "No disponible")) # No disponible
```

Diferencia importante:

- `diccionario["clave"]` genera un **error** si la clave no existe.
 - `diccionario.get("clave", valor_default)` devuelve un valor por defecto si la clave no existe.
-

4. Modificar Diccionarios

4.1 Agregar elementos

```
persona["altura"] = 1.70
```

4.2 Modificar elementos

```
persona["edad"] = 26
```

4.3 Eliminar elementos

```
del persona["altura"] # elimina la clave "altura"
```

O usando métodos:

```
persona.pop("edad") # elimina y devuelve el valor
persona.clear()     # elimina todos los elementos
```

5. Métodos útiles de Diccionarios

Método	Descripción
<code>keys()</code>	Devuelve todas las claves
<code>values()</code>	Devuelve todos los valores
<code>items()</code>	Devuelve una lista de pares (clave, valor)
<code>update()</code>	Actualiza el diccionario con otro diccionario o pares de clave-valor
<code>pop()</code>	Elimina y devuelve el valor de la clave indicada
<code>popitem()</code>	Elimina y devuelve el último par insertado (Python ≥3.7)
<code>copy()</code>	Devuelve una copia superficial del diccionario

Ejemplos:

```
persona = {"nombre": "Ana", "edad": 25}

print(persona.keys()) # dict_keys(['nombre', 'edad'])
print(persona.values()) # dict_values(['Ana', 25])
print(persona.items()) # dict_items([('nombre', 'Ana'), ('edad', 25)])

persona.update({"ciudad": "Zaragoza"})
print(persona) # {'nombre': 'Ana', 'edad': 25, 'ciudad': 'Zaragoza'}
```

6. Iterar sobre Diccionarios

Podemos recorrer **claves**, **valores** o **pares clave-valor**.

```
# Iterar sobre claves
for clave in persona:
    print(clave)

# Iterar sobre valores
for valor in persona.values():
    print(valor)

# Iterar sobre pares clave-valor
for clave, valor in persona.items():
    print(clave, valor)
```

7. Diccionarios anidados

Un diccionario puede contener otros diccionarios, listas, etc.

```
agenda = {
    "Ana": {"telefono": "1234", "ciudad": "Zaragoza"},
    "Luis": {"telefono": "5678", "ciudad": "Madrid"}
}

print(agenda["Ana"]["telefono"]) # 1234
```

8. Comprensión de Diccionarios

Python permite crear diccionarios de forma compacta usando **comprensiones**:

```
cuadrados = {x: x**2 for x in range(5)}  
print(cuadrados) # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Podemos aplicar condiciones:

```
pares = {x: x**2 for x in range(10) if x % 2 == 0}  
print(pares) # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

9. Buenas prácticas

1. Usar **claves inmutables** (strings, números, tuplas).
 2. Usar `get()` cuando no estamos seguros de que la clave exista.
 3. Evitar modificar un diccionario mientras se itera sobre él; usar copias si es necesario.
 4. Usar diccionarios anidados para estructuras complejas, pero documentarlas bien.
 5. Prefiere `items()` para recorrer pares clave-valor, es más legible y eficiente.
-

10. Ejercicios

10.1 Ejercicios de Python – Estructuras de Datos, Diccionario (Nivel Principiante)

Ejercicio 1

Enunciado: Crea un diccionario llamado `persona` con las claves `"nombre"`, `"edad"` y `"ciudad"` y asígnale valores.

Solución:

```
persona = {"nombre": "Ana", "edad": 25, "ciudad": "Zaragoza"}  
print(persona)
```

Explicación: Creamos un diccionario con `{clave: valor, ...}` y asignamos los valores deseados.

Ejercicio 2

Enunciado: Accede al valor de la clave `"nombre"` en el diccionario anterior.

Solución:

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

Explicación: Usamos `diccionario[clave]` para obtener el valor asociado.

Ejercicio 3

Enunciado: Agrega la clave `"profesion"` con el valor `"estudiante"` al diccionario `persona`.

Solución:

```
persona["profesion"] = "estudiante"  
print(persona)
```

Explicación: Para agregar un elemento, se asigna directamente `diccionario[clave] = valor`.

Ejercicio 4

Enunciado: Cambia la edad de `persona` a 26.

Solución:

```
persona["edad"] = 26  
print(persona)
```

Explicación: Modificar un valor se hace asignando un nuevo valor a la clave existente.

Ejercicio 5

Enunciado: Elimina la clave "ciudad" del diccionario.

Solución:

```
del persona["ciudad"]  
print(persona)
```

Explicación: `del diccionario[clave]` elimina la clave y su valor del diccionario.

Ejercicio 6

Enunciado: Usa `get()` para obtener la clave "altura" de `persona`, devolviendo "No disponible" si no existe.

Solución:

```
altura = persona.get("altura", "No disponible")  
print(altura)
```

Explicación: `get()` permite acceder a una clave sin causar error si no existe, usando un valor por defecto.

Ejercicio 7

Enunciado: Muestra todas las claves del diccionario `persona`.

Solución:

```
print(persona.keys())
```

Explicación: `keys()` devuelve un objeto iterable con todas las claves del diccionario.

Ejercicio 8

Enunciado: Muestra todos los valores del diccionario `persona`.

Solución:

```
print(persona.values())
```

Explicación: `values()` devuelve un objeto iterable con todos los valores del diccionario.

Ejercicio 9

Enunciado: Muestra todos los pares clave-valor del diccionario `persona`.

Solución:

```
print(persona.items())
```

Explicación: `items()` devuelve una vista de tuplas (`clave`, `valor`) de todos los elementos.

Ejercicio 10

Enunciado: Itera sobre el diccionario `persona` e imprime cada clave y su valor.

Solución:

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

Explicación: `for clave, valor in diccionario.items()` permite recorrer todos los pares del diccionario.

Ejercicio 11

Enunciado: Crea un diccionario `numeros` con claves del 1 al 5 y valores sus cuadrados.

Solución:

```
numeros = {x: x**2 for x in range(1, 6)}  
print(numeros)
```

Explicación: Usamos **comprensión de diccionarios** `{clave: valor for ...}` para generar los pares dinámicamente.

Ejercicio 12

Enunciado: Crea un diccionario `frutas` y agrega `"manzana": 3` y `"plátano": 5`.

Solución:

```
frutas = {}  
frutas["manzana"] = 3  
frutas["plátano"] = 5  
print(frutas)
```

Explicación: Inicializamos un diccionario vacío y agregamos pares clave-valor uno por uno.

Ejercicio 13

Enunciado: Elimina el par `"plátano"` del diccionario `frutas` usando `pop()`.

Solución:

```
frutas.pop("plátano")  
print(frutas)
```

Explicación: `pop(clave)` elimina la clave indicada y devuelve su valor.

Ejercicio 14

Enunciado: Combina dos diccionarios `a = {"x": 1}` y `b = {"y": 2}` usando `update()`.

Solución:

```
a = {"x": 1}
b = {"y": 2}
a.update(b)
print(a)
```

Explicación: `update()` añade todos los pares de otro diccionario al diccionario original.

Ejercicio 15

Enunciado: Comprueba si la clave "nombre" existe en `persona`.

Solución:

```
if "nombre" in persona:
    print("Existe")
else:
    print("No existe")
```

Explicación: Se usa `in` para verificar si una clave está en el diccionario.

Ejercicio 16

Enunciado: Crea un diccionario anidado `alumnos` con dos alumnos, cada uno con "edad" y "nota".

Solución:

```
alumnos = {
    "Ana": {"edad": 20, "nota": 8},
    "Luis": {"edad": 22, "nota": 7}
}
print(alumnos)
```

Explicación: Cada valor del diccionario principal es a su vez otro diccionario.

Ejercicio 17

Enunciado: Accede a la nota de "Ana" en el diccionario `alumnos`.

Solución:

```
print(alumnos["Ana"]["nota"])
```

Explicación: Primero accedemos a "Ana" y luego a la clave "nota" dentro de su diccionario.

Ejercicio 18

Enunciado: Itera sobre los alumnos e imprime su nombre y edad.

Solución:

```
for nombre, info in alumnos.items():  
    print(nombre, "tiene", info["edad"], "años")
```

Explicación: `info` es el diccionario interno, accedemos a su clave `"edad"`.

Ejercicio 19

Enunciado: Usa `copy()` para crear una copia de `persona` y luego modifica la copia.

Solución:

```
copia_persona = persona.copy()  
copia_persona["edad"] = 30  
print("Original:", persona)  
print("Copia:", copia_persona)
```

Explicación: `copy()` crea una copia superficial que permite modificar sin afectar al diccionario original.

Ejercicio 20

Enunciado: Borra todos los elementos del diccionario `frutas`.

Solución:

```
frutas.clear()  
print(frutas)
```

Explicación: `clear()` elimina todos los pares clave-valor del diccionario, dejando un diccionario vacío.

10.2 Ejercicios de Python – Estructuras de Datos, Diccionario (Nivel medio)

Ejercicio 1

Enunciado: Crea un diccionario `producto` con claves "nombre", "precio" y "cantidad".

Solución:

```
producto = {"nombre": "Laptop", "precio": 1200, "cantidad": 5}
print(producto)
```

Explicación: Diccionario básico con tres pares clave-valor.

Ejercicio 2

Enunciado: Calcula el valor total del producto ($\text{precio} \times \text{cantidad}$) y añádelo al diccionario.

Solución:

```
producto["total"] = producto["precio"] * producto["cantidad"]
print(producto)
```

Explicación: Se accede a valores existentes y se agrega un nuevo par con el resultado.

Ejercicio 3

Enunciado: Crea un diccionario `inventario` con 3 productos (cada producto es un diccionario) y muestra el inventario.

Solución:

```
inventario = {
    "Laptop": {"precio": 1200, "cantidad": 5},
    "Mouse": {"precio": 20, "cantidad": 50},
    "Teclado": {"precio": 45, "cantidad": 30}
}
print(inventario)
```

Explicación: Diccionario anidado: cada producto es una clave con un diccionario como valor.

Ejercicio 4

Enunciado: Itera sobre el inventario y muestra el valor total (precio $\times$ cantidad) de cada producto.

Solución:

```
for nombre, info in inventario.items():
    total = info["precio"] * info["cantidad"]
    print(nombre, "-> Valor total:", total)
```

Explicación: Se recorre cada par clave-valor; `info` es un diccionario interno.

Ejercicio 5

Enunciado: Suma el valor total de todos los productos en el inventario.

Solución:

```
valor_total = sum(info["precio"] * info["cantidad"] for info in
inventario.values())

print("Valor total del inventario:", valor_total)
```

Explicación: Uso de comprensión y `sum()` para acumular valores de todos los productos.

Ejercicio 6

Enunciado: Filtra los productos cuyo precio sea mayor a 100 y crea un nuevo diccionario `caros`.

Solución:

```
caros = {nombre: info for nombre, info in inventario.items() if info["precio"] >
100}

print(caros)
```

Explicación: Comprensión de diccionarios con condición `if` para filtrar elementos.

Ejercicio 7

Enunciado: Crea un diccionario que cuente la cantidad de letras de cada palabra en la lista `palabras = ["manzana", "banana", "cereza"]`.

Solución:

```
palabras = ["manzana", "banana", "cereza"]

longitudes = {palabra: len(palabra) for palabra in palabras}

print(longitudes)
```

Explicación: La comprensión recorre la lista y calcula `len()` para cada palabra.

Ejercicio 8

Enunciado: Une dos diccionarios `a = {"x": 1}` y `b = {"y": 2}` sin modificar los originales.

Solución:

```
a = {"x": 1}
b = {"y": 2}
union = {**a, **b}
print(union)
```

Explicación: Uso de `**` para desempaquetar diccionarios en uno nuevo.

Ejercicio 9

Enunciado: Intercambia los valores y claves de un diccionario `dic = {"a": 1, "b": 2}`.

Solución:

```
dic = {"a": 1, "b": 2}
invertido = {valor: clave for clave, valor in dic.items()}
print(invertido)
```

Explicación: Comprensión que intercambia cada par clave-valor.

Ejercicio 10

Enunciado: Crea un diccionario de números del 1 al 10 con sus cubos como valores.

Solución:

```
cubos = {x: x**3 for x in range(1, 11)}
print(cubos)
```

Explicación: Uso de comprensión para generar diccionario de manera compacta.

Ejercicio 11

Enunciado: Elimina todas las claves cuyo valor sea menor que 50 en `cubos`.

Solución:

```
cubos = {x: x**3 for x in range(1, 11)}
cubos_filtrado = {k: v for k, v in cubos.items() if v >= 50}
print(cubos_filtrado)
```

Explicación: Comprensión con condición para filtrar claves según valor.

Ejercicio 12

Enunciado: Combina un diccionario de nombres y otro de edades en un solo diccionario de personas.

```
nombres = {1: "Ana", 2: "Luis"}
```

```
edades = {1: 20, 2: 22}
```

Solución:

```
personas = {clave: {"nombre": nombres[clave], "edad": edades[clave]} for clave
in nombres}
```

```
print(personas)
```

Explicación: Diccionario anidado combinando información de dos diccionarios usando comprensión.

Ejercicio 13

Enunciado: Incrementa en 1 la edad de cada persona en `personas`.

Solución:

```
for info in personas.values():
```

```
    info["edad"] += 1
```

```
print(personas)
```

Explicación: Recorremos los diccionarios internos y modificamos sus valores directamente.

Ejercicio 14

Enunciado: Ordena un diccionario `notas = {"Ana": 8, "Luis": 7, "Marta": 9}` por valor descendente.

Solución:

```
notas = {"Ana": 8, "Luis": 7, "Marta": 9}
```

```
notas_ordenadas = dict(sorted(notas.items(), key=lambda item: item[1],
reverse=True))
```

```
print(notas_ordenadas)
```

Explicación: `sorted()` con `key` permite ordenar por valores; luego se convierte a diccionario.

Ejercicio 15

Enunciado: Verifica si todas las edades en `personas` son mayores de 18.

Solución:

```
mayores = all(info["edad"] > 18 for info in personas.values())
print(mayores)
```

Explicación: `all()` devuelve `True` si todas las condiciones son verdaderas.

Ejercicio 16

Enunciado: Encuentra la persona con la mayor edad en `personas`.

Solución:

```
mayor = max(personas.items(), key=lambda item: item[1]["edad"])
print(mayor)
```

Explicación: `max()` con `key` nos permite comparar por edad dentro de diccionarios anidados.

Ejercicio 17

Enunciado: Crea un diccionario `cuentas` con 3 personas y su saldo; agrega un 10% a cada saldo.

Solución:

```
cuentas = {"Ana": 100, "Luis": 200, "Marta": 150}
cuentas_actualizadas = {persona: saldo*1.1 for persona, saldo in
cuentas.items()}
print(cuentas_actualizadas)
```

Explicación: Comprensión con operación sobre los valores.

Ejercicio 18

Enunciado: Combina tres listas en un diccionario:

```
keys = ["a", "b", "c"]
values1 = [1, 2, 3]
values2 = [4, 5, 6]
```

Usa valores como tuplas (`values1[i]`, `values2[i]`).

Solución:

```
dic = {k: (v1, v2) for k, v1, v2 in zip(keys, values1, values2)}
print(dic)
```

Explicación: `zip()` permite iterar varias listas a la vez y construir diccionario.

Ejercicio 19

Enunciado: Invierte un diccionario con valores repetidos y guarda los valores como listas.

```
dic = {"a": 1, "b": 2, "c": 1}
```

Solución:

```
invertido = {}  
for k, v in dic.items():  
    invertido.setdefault(v, []).append(k)  
print(invertido)
```

Explicación: `setdefault()` crea una lista si no existe y permite agrupar claves con mismo valor.

Ejercicio 20

Enunciado: Combina dos diccionarios de productos y si hay claves iguales, suma sus cantidades.

```
stock1 = {"Laptop": 5, "Mouse": 10}  
stock2 = {"Laptop": 3, "Teclado": 7}
```

Solución:

```
from collections import Counter  
total_stock = dict(Counter(stock1) + Counter(stock2))  
print(total_stock)
```

Explicación: `Counter` permite sumar los valores de claves iguales automáticamente.

10.3 Ejercicios de Python – Estructuras de Datos, Diccionario (Nivel Avanzado)

Ejercicio 1

Enunciado: Crea un diccionario anidado `empleados` con 3 empleados, cada uno con "departamento", "salario" y "años_experiencia".

Solución:

```
empleados = {  
    "Ana": {"departamento": "IT", "salario": 3000, "años_experiencia": 5},  
    "Luis": {"departamento": "HR", "salario": 2500, "años_experiencia": 3},  
    "Marta": {"departamento": "IT", "salario": 3200, "años_experiencia": 6}  
}  
  
print(empleados)
```

Explicación: Diccionario con diccionarios internos para cada empleado.

Ejercicio 2

Enunciado: Calcula el salario promedio de todos los empleados.

Solución:

```
promedio = sum(emp["salario"] for emp in empleados.values()) / len(empleados)  
print(promedio)
```

Explicación: Se suman los salarios con comprensión y se divide entre el número de empleados.

Ejercicio 3

Enunciado: Filtra los empleados del departamento "IT" y crea un nuevo diccionario.

Solución:

```
it_empleados = {nombre: info for nombre, info in empleados.items() if  
info["departamento"] == "IT"}  
  
print(it_empleados)
```

Explicación: Comprensión de diccionarios con condición para filtrar por departamento.

Ejercicio 4

Enunciado: Encuentra el empleado con mayor salario.

Solución:

```
max_salario = max(empleados.items(), key=lambda item: item[1]["salario"])
print(max_salario)
```

Explicación: `max()` con `key` permite comparar valores dentro de diccionarios anidados.

Ejercicio 5

Enunciado: Ordena los empleados por años de experiencia descendente.

Solución:

```
ordenados = dict(sorted(empleados.items(), key=lambda item: item[1]
["años_experiencia"], reverse=True))
print(ordenados)
```

Explicación: `sorted()` con `key` y `reverse=True` permite ordenar diccionarios complejos.

Ejercicio 6

Enunciado: Suma 500 al salario de todos los empleados del departamento "IT".

Solución:

```
for info in empleados.values():
    if info["departamento"] == "IT":
        info["salario"] += 500
print(empleados)
```

Explicación: Modificación in-place de valores dentro de diccionarios anidados.

Ejercicio 7

Enunciado: Crea un diccionario que cuente la cantidad de cada palabra en la lista:

```
palabras = ["python", "java", "python", "c++", "java", "python"]
```

Solución:

```
from collections import Counter
conteo = dict(Counter(palabras))
print(conteo)
```

Explicación: `Counter` cuenta automáticamente la frecuencia de cada elemento.

Ejercicio 8

Enunciado: Invierte un diccionario con valores repetidos guardando las claves en listas.

```
dic = {"a": 1, "b": 2, "c": 1}
```

Solución:

```
invertido = {}  
for k, v in dic.items():  
    invertido.setdefault(v, []).append(k)  
print(invertido)
```

Explicación: `setdefault()` crea una lista si no existe y agrupa claves con mismo valor.

Ejercicio 9

Enunciado: Combina dos diccionarios sumando valores de claves coincidentes:

```
stock1 = {"Laptop": 5, "Mouse": 10}  
stock2 = {"Laptop": 3, "Teclado": 7}
```

Solución:

```
from collections import Counter  
total_stock = dict(Counter(stock1) + Counter(stock2))  
print(total_stock)
```

Explicación: `Counter` permite sumar los valores de claves iguales automáticamente.

Ejercicio 10

Enunciado: Crea un diccionario que convierta una lista de tuplas en diccionario:

```
datos = [("Ana", 25), ("Luis", 30), ("Marta", 28)]
```

Solución:

```
dic = {nombre: edad for nombre, edad in datos}  
print(dic)
```

Explicación: Comprensión de diccionarios a partir de iterables de tuplas.

Ejercicio 11

Enunciado: Crea un diccionario que contenga los números del 1 al 10 y sus factoriales como valores.

Solución:

```
import math

factoriales = {x: math.factorial(x) for x in range(1, 11)}

print(factoriales)
```

Explicación: Uso de `math.factorial()` en comprensión para valores calculados dinámicamente.

Ejercicio 12

Enunciado: Filtra un diccionario eliminando todos los pares cuyo valor sea menor que 50.

```
datos = {"a": 30, "b": 60, "c": 45, "d": 80}
```

Solución:

```
filtrado = {k: v for k, v in datos.items() if v >= 50}

print(filtrado)
```

Explicación: Comprensión de diccionarios con condición para filtrar por valor.

Ejercicio 13

Enunciado: Crea un diccionario que agrupe palabras por su primera letra:

```
palabras = ["python", "java", "perl", "php", "c", "c++"]
```

Solución:

```
agrupadas = {}

for palabra in palabras:
    agrupadas.setdefault(palabra[0], []).append(palabra)

print(agrupadas)
```

Explicación: `setdefault()` permite agrupar elementos por clave común.

Ejercicio 14

Enunciado: Normaliza los valores de un diccionario dividiendo cada valor por la suma total.

```
valores = {"a": 2, "b": 3, "c": 5}
```

Solución:

```
suma = sum(valores.values())
normalizado = {k: v/suma for k, v in valores.items()}
print(normalizado)
```

Explicación: Comprensión de diccionarios con operación matemática sobre los valores.

Ejercicio 15

Enunciado: Fusiona tres listas en un diccionario, usando valores como tuplas:

```
keys = ["x", "y", "z"]
```

```
v1 = [1, 2, 3]
```

```
v2 = [4, 5, 6]
```

Solución:

```
dic = {k: (val1, val2) for k, val1, val2 in zip(keys, v1, v2)}
print(dic)
```

Explicación: `zip()` permite iterar simultáneamente varias listas.

Ejercicio 16

Enunciado: Usa `defaultdict` para contar la frecuencia de letras en un texto.

```
texto = "python es poderoso"
```

Solución:

```
from collections import defaultdict
frecuencia = defaultdict(int)
for letra in texto.replace(" ", ""):
    frecuencia[letra] += 1
print(dict(frecuencia))
```

Explicación: `defaultdict(int)` inicializa valores en 0 automáticamente para evitar `KeyError`.

Ejercicio 17

Enunciado: Encuentra la clave con el valor máximo en un diccionario.

```
datos = {"a": 10, "b": 25, "c": 15}
```

Solución:

```
max_clave = max(datos, key=datos.get)
print(max_clave)
```

Explicación: `max()` con `key=datos.get` devuelve la clave asociada al mayor valor.

Ejercicio 18

Enunciado: Combina dos diccionarios, concatenando valores si las claves coinciden.

```
d1 = {"x": "a", "y": "b"}
d2 = {"x": "c", "z": "d"}
```

Solución:

```
resultado = {}
for k in set(d1) | set(d2):
    resultado[k] = d1.get(k, "") + d2.get(k, "")
print(resultado)
```

Explicación: `set(d1) | set(d2)` toma todas las claves únicas; se concatenan valores existentes.

Ejercicio 19

Enunciado: Invierte un diccionario anidado de la forma `{clave: {subclave: valor}}` a `{subclave: {clave: valor}}`.

```
dic = {"A": {"x": 1, "y": 2}, "B": {"x": 3, "y": 4}}
```

Solución:

```
invertido = {}
for k, subdic in dic.items():
    for sk, v in subdic.items():
        invertido.setdefault(sk, {})[k] = v
print(invertido)
```

Explicación: `setdefault()` permite crear subdiccionarios y reorganizar la estructura.

Ejercicio 20

Enunciado: Dado un diccionario de productos y ventas, calcula un ranking de productos por ventas totales.

```
ventas = {"Laptop": [3,5,2], "Mouse": [10,5,8], "Teclado": [2,3,1]}
```

Solución:

```
ranking = {producto: sum(cantidades) for producto, cantidades in ventas.items()}  
ranking_ordenado = dict(sorted(ranking.items(), key=lambda x: x[1],  
reverse=True))  
print(ranking_ordenado)
```

Explicación: Se suman las listas de ventas, luego se ordena el diccionario por valor descendente.

Lección 8.: Funciones en Python

1. Introducción a las funciones

Una **función** es un bloque de código reutilizable que realiza una tarea específica.

- Permite **organizar el código, evitar repeticiones y hacer programas más legibles**.
- Se define usando la palabra clave `def`.

Sintaxis básica:

```
def nombre_funcion(parametro1, parametro2):  
    # Bloque de código  
    resultado = parametro1 + parametro2  
    return resultado
```

- `def` indica que vamos a definir una función.
 - `nombre_funcion` es el identificador de la función.
 - `parametro1, parametro2` son los **parámetros de entrada**.
 - `return` devuelve un valor al llamar la función.
-

2. Definir y llamar funciones

2.1 Función simple sin parámetros

```
def saludar():  
    print("¡Hola, mundo!")  
  
saludar()
```

Explicación: La función no recibe argumentos y simplemente ejecuta una acción.

2.2 Función con parámetros

```
def saludar(nombre):  
    print(f"¡Hola, {nombre}!")  
  
saludar("Ana")
```

Explicación: `nombre` es un parámetro que se pasa al llamar la función.

2.3 Función con retorno de valor

```
def sumar(a, b):  
    return a + b  
  
resultado = sumar(5, 3)  
print(resultado)
```

Explicación: return permite que la función devuelva un valor que puede almacenarse o usarse en expresiones.

3. Tipos de parámetros

3.1 Parámetros por posición

```
def restar(a, b):  
    return a - b  
  
print(restar(10, 3)) # 7
```

Los argumentos se asignan según el orden de definición.

3.2 Parámetros por nombre (keyword arguments)

```
print(restar(b=3, a=10)) # 7
```

Se asignan los valores indicando explícitamente la clave, permitiendo cambiar el orden.

3.3 Parámetros por defecto

```
def multiplicar(a, b=2):  
    return a * b  
  
print(multiplicar(5)) # 10
```

Si no se pasa un argumento, se usa el valor por defecto.

3.4 Argumentos variables (*args)

```
def suma_todos(*numeros):  
    return sum(numeros)  
  
print(suma_todos(1, 2, 3, 4)) # 10
```

- `*args` permite recibir un número **indeterminado de argumentos** como una tupla.

3.5 Argumentos con palabras clave (**kwargs)

```
def mostrar_info(**info):  
    for clave, valor in info.items():  
        print(f"{clave}: {valor}")  
  
mostrar_info(nombre="Ana", edad=25, ciudad="Zaragoza")
```

- `**kwargs` recibe un número **indeterminado de argumentos con nombre** como un diccionario.

4. Funciones anidadas

Una función puede definirse dentro de otra.

```
def funcion_exterior(x):
    def funcion_interior(y):
        return y + 1
    return funcion_interior(x) * 2

print(funcion_exterior(5)) # 12
```

Explicación: Las funciones internas solo existen dentro del ámbito de la función externa.

5. Funciones lambda (funciones anónimas)

- Funciones pequeñas de una sola línea, generalmente para operaciones rápidas.

```
cuadrado = lambda x: x**2
print(cuadrado(5)) # 25
```

- Uso en combinación con `map()`, `filter()`, `sorted()`:

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

6. Funciones recursivas

Una función que se llama a sí misma.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5)) # 120
```

Explicación: La recursión permite resolver problemas repetitivos dividiéndolos en subproblemas.

7. Documentación de funciones (docstring)

Se recomienda documentar la función usando triple comillas.

```
def sumar(a, b):
    """
    Devuelve la suma de dos números.
    Parámetros:
    a (int/float): primer número
    b (int/float): segundo número
    """
    return a + b

print(sumar.__doc__)
```

Explicación: `__doc__` muestra la documentación de la función, útil para mantener el código legible.

8. Buenas prácticas

1. Nombrar funciones con verbos descriptivos (`calcular_area`, `obtener_datos`).
 2. Usar parámetros con nombres claros.
 3. Documentar funciones con docstrings.
 4. Evitar funciones demasiado largas; una función debe hacer **una sola tarea**.
 5. Usar `*args` y `**kwargs` solo cuando sea necesario.
 6. Preferir funciones puras (sin modificar variables externas) cuando sea posible.
-

9. Ejemplos prácticos

1. Función que calcule el área de un círculo:

```
import math
def area_circulo(radio):
    return math.pi * radio**2
```

2. Función que filtre números pares de una lista:

```
def filtrar_pares(lista):
    return [x for x in lista if x % 2 == 0]
```

3. Función que reciba varias notas y devuelva la nota máxima:

```
def nota_maxima(*notas):
    return max(notas)
```

4. Función recursiva que calcule la suma de los primeros n números:

```
def suma_numeros(n):
    if n == 1:
        return 1
    return n + suma_numeros(n-1)
```

5. Función que reciba información de usuario con **kwargs**:

```
def mostrar_usuario(**info):
    for clave, valor in info.items():
        print(f"{clave}: {valor}")
```

10. Ejercicios

10.1 Ejercicios de Python – Funciones (Nivel Principiante)

Ejercicio 1

Enunciado: Crea una función `saludar` que imprima "¡Hola!".

Solución:

```
def saludar():  
    print("¡Hola!")
```

```
saludar()
```

Explicación: Función sin parámetros que imprime un mensaje.

Ejercicio 2

Enunciado: Crea una función `saludar_nombre` que reciba un nombre y lo salude.

Solución:

```
def saludar_nombre(nombre):  
    print(f"¡Hola, {nombre}!")
```

```
saludar_nombre("Ana")
```

Explicación: Función con parámetro para personalizar el saludo.

Ejercicio 3

Enunciado: Crea una función `sumar` que reciba dos números y devuelva su suma.

Solución:

```
def sumar(a, b):  
    return a + b
```

```
resultado = sumar(5, 3)  
print(resultado)
```

Explicación: Uso de `return` para devolver el resultado.

Ejercicio 4

Enunciado: Crea una función `es_par` que devuelva `True` si un número es par y `False` si es impar.

Solución:

```
def es_par(n):  
    return n % 2 == 0
```

```
print(es_par(4))
```

```
print(es_par(7))
```

Explicación: Operador módulo `%` permite verificar divisibilidad.

Ejercicio 5

Enunciado: Crea una función que reciba un número y devuelva su cuadrado.

Solución:

```
def cuadrado(n):  
    return n ** 2
```

```
print(cuadrado(5))
```

Explicación: Operador `**` calcula la potencia.

Ejercicio 6

Enunciado: Crea una función que reciba un número y devuelva "Positivo", "Negativo" o "Cero".

Solución:

```
def signo(n):  
    if n > 0:  
        return "Positivo"  
    elif n < 0:  
        return "Negativo"  
    else:  
        return "Cero"
```

```
print(signo(5))  
print(signo(-2))  
print(signo(0))
```

Explicación: Uso de condicionales dentro de una función.

Ejercicio 7

Enunciado: Crea una función que reciba una lista y devuelva su longitud.

Solución:

```
def longitud(lista):  
    return len(lista)
```

```
print(longitud([1,2,3,4]))
```

Explicación: `len()` devuelve la cantidad de elementos en la lista.

Ejercicio 8

Enunciado: Crea una función que reciba una lista de números y devuelva la suma.

Solución:

```
def sumar_lista(lista):  
    return sum(lista)
```

```
print(sumar_lista([1,2,3,4]))
```

Explicación: `sum()` suma todos los elementos de la lista.

Ejercicio 9

Enunciado: Crea una función que reciba un número y diga si es divisible por 3.

Solución:

```
def divisible_por_3(n):  
    return n % 3 == 0
```

```
print(divisible_por_3(9))  
print(divisible_por_3(10))
```

Explicación: Verificación de divisibilidad usando %.

Ejercicio 10

Enunciado: Crea una función que reciba dos números y devuelva el mayor.

Solución:

```
def mayor(a, b):  
    if a > b:  
        return a  
    else:  
        return b
```

```
print(mayor(5, 8))
```

Explicación: Uso de condicional `if-else` para comparar valores.

Ejercicio 11

Enunciado: Crea una función que reciba un número y devuelva su valor absoluto.

Solución:

```
def valor_absoluto(n):  
    if n < 0:  
        return -n  
    return n
```

```
print(valor_absoluto(-5))  
print(valor_absoluto(3))
```

Explicación: Cambiamos el signo si el número es negativo.

Ejercicio 12

Enunciado: Crea una función que reciba un nombre y un apellido y los imprima juntos.

Solución:

```
def nombre_completo(nombre, apellido):  
    print(f"{nombre} {apellido}")
```

```
nombre_completo("Ana", "García")
```

Explicación: Concatenación de cadenas usando f-strings.

Ejercicio 13

Enunciado: Crea una función con un parámetro por defecto `saludo="Hola"`.

Solución:

```
def saludar_persona(nombre, saludo="Hola"):  
    print(f"{saludo}, {nombre}!")
```

```
saludar_persona("Ana")
```

```
saludar_persona("Luis", "Buenos días")
```

Explicación: Parámetros por defecto permiten que algunos argumentos sean opcionales.

Ejercicio 14

Enunciado: Crea una función que reciba varios números y devuelva su promedio.

Solución:

```
def promedio(*numeros):  
    return sum(numeros)/len(numeros)
```

```
print(promedio(1,2,3,4,5))
```

Explicación: `*numeros` permite recibir un número indefinido de argumentos.

Ejercicio 15

Enunciado: Crea una función que reciba una lista y devuelva solo los números pares.

Solución:

```
def filtrar_pares(lista):  
    return [x for x in lista if x % 2 == 0]  
  
print(filtrar_pares([1,2,3,4,5,6]))
```

Explicación: Comprensión de listas para filtrar elementos según condición.

Ejercicio 16

Enunciado: Crea una función que reciba un número y devuelva `True` si es primo y `False` si no.

Solución:

```
def es_primo(n):  
    if n < 2:  
        return False  
    for i in range(2, int(n**0.5)+1):  
        if n % i == 0:  
            return False  
    return True  
  
print(es_primo(7))  
print(es_primo(8))
```

Explicación: Verifica divisibilidad hasta la raíz cuadrada del número.

Ejercicio 17

Enunciado: Crea una función que reciba un diccionario y devuelva las claves.

Solución:

```
def claves(dic):  
    return list(dic.keys())  
  
print(claves({"a":1, "b":2}))
```

Explicación: `keys()` devuelve las claves, convertidas a lista.

Ejercicio 18

Enunciado: Crea una función que reciba un diccionario y devuelva los valores.

Solución:

```
def valores(dic):  
    return list(dic.values())
```

```
print(valores({"a":1, "b":2}))
```

Explicación: `values()` devuelve los valores, convertidos a lista.

Ejercicio 19

Enunciado: Crea una función que reciba un número y devuelva "Fizz" si es divisible por 3, "Buzz" si es divisible por 5 y "FizzBuzz" si es divisible por ambos.

Solución:

```
def fizzbuzz(n):  
    if n % 3 == 0 and n % 5 == 0:  
        return "FizzBuzz"  
    elif n % 3 == 0:  
        return "Fizz"  
    elif n % 5 == 0:  
        return "Buzz"  
    else:  
        return n
```

```
print(fizzbuzz(15))  
print(fizzbuzz(9))  
print(fizzbuzz(10))  
print(fizzbuzz(7))
```

Explicación: Uso de condicionales encadenados para verificar divisibilidad múltiple.

Ejercicio 20

Enunciado: Crea una función recursiva que calcule la suma de los números del 1 al n.

Solución:

```
def suma_n(n):  
    if n == 1:  
        return 1  
    return n + suma_n(n-1)  
  
print(suma_n(5))
```

Explicación: La función se llama a sí misma hasta llegar al caso base $n==1$.

10.2 Ejercicios de Python – Funciones (Nivel Medio)

Ejercicio 1

Enunciado: Crea una función que reciba una lista de números y devuelva la suma de los pares.

Solución:

```
def suma_pares(lista):  
    return sum(x for x in lista if x % 2 == 0)
```

```
print(suma_pares([1,2,3,4,5,6]))
```

Explicación: Se usa comprensión de listas para filtrar los pares y `sum()` para sumarlos.

Ejercicio 2

Enunciado: Crea una función que reciba una lista de números y devuelva la lista ordenada de menor a mayor.

Solución:

```
def ordenar_lista(lista):  
    return sorted(lista)
```

```
print(ordenar_lista([5,2,9,1]))
```

Explicación: `sorted()` devuelve una lista ordenada sin modificar la original.

Ejercicio 3

Enunciado: Crea una función que reciba una cadena y devuelva la misma cadena invertida.

Solución:

```
def invertir_cadena(cadena):  
    return cadena[::-1]
```

```
print(invertir_cadena("python"))
```

Explicación: Se usa slicing con paso negativo para invertir la cadena.

Ejercicio 4

Enunciado: Crea una función que reciba un número y devuelva una lista con todos los divisores de ese número.

Solución:

```
def divisores(n):  
    return [i for i in range(1, n+1) if n % i == 0]  
  
print(divisores(12))
```

Explicación: Comprensión de listas con condición de divisibilidad.

Ejercicio 5

Enunciado: Crea una función que reciba dos listas y devuelva una lista con los elementos comunes.

Solución:

```
def comunes(lista1, lista2):  
    return [x for x in lista1 if x in lista2]  
  
print(comunes([1,2,3],[2,3,4]))
```

Explicación: Se filtran elementos de la primera lista que existen en la segunda.

Ejercicio 6

Enunciado: Crea una función que reciba una lista de números y devuelva un diccionario con "pares" y "impares" como claves.

Solución:

```
def separar_pares_impares(lista):  
    return {  
        "pares": [x for x in lista if x % 2 == 0],  
        "impares": [x for x in lista if x % 2 != 0]  
    }  
  
print(separar_pares_impares([1,2,3,4,5,6]))
```

Explicación: Diccionario con listas generadas por comprensión según condición.

Ejercicio 7

Enunciado: Crea una función que reciba una cadena y cuente cuántas vocales tiene.

Solución:

```
def contar_vocales(cadena):  
    return sum(1 for letra in cadena.lower() if letra in "aeiou")  
  
print(contar_vocales("Python es divertido"))
```

Explicación: Se recorre la cadena y se suma 1 por cada vocal encontrada.

Ejercicio 8

Enunciado: Crea una función que reciba cualquier número de argumentos y devuelva la multiplicación de todos ellos.

Solución:

```
def multiplicar_todos(*args):  
    resultado = 1  
    for n in args:  
        resultado *= n  
    return resultado  
  
print(multiplicar_todos(2,3,4))
```

Explicación: *args permite recibir cualquier cantidad de argumentos.

Ejercicio 9

Enunciado: Crea una función que reciba una lista de números y devuelva el mayor y el menor.

Solución:

```
def mayor_menor(lista):  
    return max(lista), min(lista)  
  
print(mayor_menor([5,2,9,1]))
```

Explicación: max() y min() devuelven el mayor y menor elemento de la lista.

Ejercicio 10

Enunciado: Crea una función recursiva que calcule el factorial de un número.

Solución:

```
def factorial(n):  
    if n == 0:  
        return 1  
    return n * factorial(n-1)  
  
print(factorial(5))
```

Explicación: La función se llama a sí misma hasta el caso base $n==0$.

Ejercicio 11

Enunciado: Crea una función que reciba un número y devuelva `True` si es primo.

Solución:

```
def es_primo(n):  
    if n < 2:  
        return False  
    for i in range(2, int(n**0.5)+1):  
        if n % i == 0:  
            return False  
    return True  
  
print(es_primo(7))  
print(es_primo(8))
```

Explicación: Se verifica divisibilidad hasta la raíz cuadrada de n para eficiencia.

Ejercicio 12

Enunciado: Crea una función que reciba una lista de cadenas y devuelva solo las que tengan más de 5 caracteres.

Solución:

```
def filtrar_largas(lista):  
    return [s for s in lista if len(s) > 5]  
  
print(filtrar_largas(["python", "java", "javascript", "c"]))
```

Explicación: Comprensión de listas con condición sobre la longitud de cadenas.

Ejercicio 13

Enunciado: Crea una función que reciba un diccionario y devuelva las claves cuyo valor sea mayor a 10.

Solución:

```
def claves_mayores(dic):  
    return [k for k,v in dic.items() if v > 10]  
  
print(claves_mayores({"a":5, "b":15, "c":12}))
```

Explicación: Se filtran claves según condición sobre sus valores.

Ejercicio 14

Enunciado: Crea una función que reciba una lista de números y devuelva otra lista con cada número al cuadrado.

Solución:

```
def cuadrados(lista):  
    return [x**2 for x in lista]  
  
print(cuadrados([1,2,3,4]))
```

Explicación: Comprensión de listas con operación sobre cada elemento.

Ejercicio 15

Enunciado: Crea una función que reciba una lista de números y devuelva `True` si todos son positivos.

Solución:

```
def todos_positivos(lista):  
    return all(x > 0 for x in lista)  
  
print(todos_positivos([1,2,3]))  
print(todos_positivos([1,-2,3]))
```

Explicación: `all()` verifica que todos los elementos cumplan la condición.

Ejercicio 16

Enunciado: Crea una función que reciba una lista de números y devuelva `True` si al menos uno es par.

Solución:

```
def algun_par(lista):  
    return any(x % 2 == 0 for x in lista)  
  
print(algun_par([1,3,5]))  
print(algun_par([1,4,5]))
```

Explicación: `any()` devuelve `True` si **algún** elemento cumple la condición.

Ejercicio 17

Enunciado: Crea una función que reciba un texto y devuelva un diccionario con la cantidad de cada letra.

Solución:

```
def contar_letras(texto):  
    resultado = {}  
    for letra in texto.replace(" ", "").lower():  
        resultado[letra] = resultado.get(letra,0) + 1  
    return resultado  
  
print(contar_letras("Hola Mundo"))
```

Explicación: Se usa `get()` para inicializar en 0 si la letra no existe aún en el diccionario.

Ejercicio 18

Enunciado: Crea una función que reciba dos listas y devuelva un diccionario que las combine como clave-valor.

Solución:

```
def combinar_listas(claves, valores):  
    return dict(zip(claves, valores))  
  
print(combinar_listas(["a","b","c"], [1,2,3]))
```

Explicación: `zip()` combina dos listas elemento a elemento y `dict()` lo convierte en diccionario.

Ejercicio 19

Enunciado: Crea una función que reciba una lista de números y devuelva un diccionario con `min`, `max` y `promedio`.

Solución:

```
def estadisticas(lista):  
    return {  
        "min": min(lista),  
        "max": max(lista),  
        "promedio": sum(lista)/len(lista)  
    }  
  
print(estadisticas([1,2,3,4,5]))
```

Explicación: Uso de funciones integradas `min()`, `max()` y `sum()` para obtener estadísticas.

Ejercicio 20

Enunciado: Crea una función que reciba una lista de números y devuelva otra lista con los números multiplicados por un factor opcional `factor=2`.

Solución:

```
def multiplicar_por(lista, factor=2):  
    return [x*factor for x in lista]  
  
print(multiplicar_por([1,2,3]))  
print(multiplicar_por([1,2,3], factor=5))
```

Explicación: Parámetro opcional permite multiplicar por un valor diferente al predeterminado.

10.3 Ejercicios de Python – Funciones (Nivel Avanzado)

Ejercicio 1

Enunciado: Crea una función que reciba otra función y una lista, y devuelva una nueva lista con el resultado de aplicar la función a cada elemento.

Solución:

```
def aplicar_funcion(func, lista):  
    return [func(x) for x in lista]  
  
print(aplicar_funcion(lambda x: x**2, [1,2,3,4]))
```

Explicación: Función de orden superior que recibe otra función como argumento y aplica comprensión de listas.

Ejercicio 2

Enunciado: Crea una función que reciba una lista de números y una función, y devuelva solo los elementos que cumplan la condición de la función.

Solución:

```
def filtrar(lista, condicion):  
    return [x for x in lista if condicion(x)]  
  
print(filtrar([1,2,3,4,5], lambda x: x%2==0))
```

Explicación: Uso de función como filtro; `lambda` define la condición al vuelo.

Ejercicio 3

Enunciado: Crea una función recursiva que calcule la suma de los dígitos de un número.

Solución:

```
def suma_digitos(n):  
    if n < 10:  
        return n  
    return n % 10 + suma_digitos(n // 10)  
  
print(suma_digitos(1234))
```

Explicación: Recursión que divide el número en dígitos hasta llegar a un solo dígito.

Ejercicio 4

Enunciado: Crea una función que reciba cualquier número de argumentos y devuelva el promedio.

Solución:

```
def promedio(*args):  
    return sum(args)/len(args)
```

```
print(promedio(2,4,6,8))
```

Explicación: *args permite recibir un número indefinido de argumentos.

Ejercicio 5

Enunciado: Crea una función que reciba un diccionario y un valor y devuelva las claves que tengan ese valor.

Solución:

```
def claves_por_valor(dic, valor):  
    return [k for k, v in dic.items() if v == valor]
```

```
print(claves_por_valor({"a":1, "b":2, "c":1}, 1))
```

Explicación: Comprensión de listas sobre los items del diccionario.

Ejercicio 6

Enunciado: Crea una función que reciba otra función y devuelva un **decorador** que imprima "Ejecutando función" antes de ejecutar la función.

Solución:

```
def decorador(func):  
    def wrapper(*args, **kwargs):  
        print("Ejecutando función")  
        return func(*args, **kwargs)  
    return wrapper
```

```
@decorador
```

```
def saludar(nombre):  
    print(f"Hola {nombre}")
```

```
saludar("Ana")
```

Explicación: Uso de closures y decoradores; `*args` y `**kwargs` permiten cualquier firma de función.

Ejercicio 7

Enunciado: Crea una función que genere un **contador** usando closure.

Solución:

```
def contador_inicial(n=0):  
    def contador():  
        nonlocal n  
        n += 1  
        return n  
    return contador
```

```
c = contador_inicial()  
print(c())  
print(c())
```

Explicación: `nonlocal` permite modificar la variable del scope exterior dentro del closure.

Ejercicio 8

Enunciado: Crea una función que reciba una lista de funciones y un valor, y devuelva la lista de resultados de aplicar cada función al valor.

Solución:

```
def aplicar_lista(funcs, valor):  
    return [f(valor) for f in funcs]  
  
print(aplicar_lista([lambda x: x+1, lambda x: x*2], 5))
```

Explicación: Función de orden superior que aplica varias funciones a un mismo valor.

Ejercicio 9

Enunciado: Crea una función recursiva que genere la serie de Fibonacci hasta el n-ésimo término.

Solución:

```
def fibonacci(n):  
    if n <= 0:  
        return []  
    elif n == 1:  
        return [0]  
    elif n == 2:  
        return [0, 1]  
    else:  
        serie = fibonacci(n-1)  
        serie.append(serie[-1] + serie[-2])  
        return serie  
  
print(fibonacci(7))
```

Explicación: Recursión que construye la serie añadiendo el último número más el penúltimo.

Ejercicio 10

Enunciado: Crea una función que reciba un diccionario y devuelva otro diccionario con los valores al cuadrado.

Solución:

```
def cuadrado_valores(dic):  
    return {k: v**2 for k, v in dic.items()}  
  
print(cuadrado_valores({"a":2, "b":3}))
```

Explicación: Comprensión de diccionarios con operación sobre valores.

Ejercicio 11

Enunciado: Crea una función que reciba dos listas y devuelva un diccionario combinándolas como clave-valor; si las listas no son del mismo tamaño, ignora los sobrantes.

Solución:

```
def combinar_listas(claves, valores):  
    return dict(zip(claves, valores))  
  
print(combinar_listas(["a","b","c"], [1,2]))
```

Explicación: `zip()` detiene la combinación cuando la lista más corta termina.

Ejercicio 12

Enunciado: Crea una función que reciba una lista de números y un valor `n`, y devuelva otra lista con cada número multiplicado por `n` usando `map`.

Solución:

```
def multiplicar_lista(lista, n):  
    return list(map(lambda x: x*n, lista))  
  
print(multiplicar_lista([1,2,3], 5))
```

Explicación: `map` aplica la función `lambda` a cada elemento de la lista.

Ejercicio 13

Enunciado: Crea una función que reciba una lista de números y devuelva los números pares al cuadrado usando `filter` y `map`.

Solución:

```
def pares_cuadrado(lista):  
    return list(map(lambda x: x**2, filter(lambda x: x%2==0, lista)))  
  
print(pares_cuadrado([1,2,3,4,5,6]))
```

Explicación: `filter` selecciona pares, luego `map` eleva al cuadrado.

Ejercicio 14

Enunciado: Crea una función que reciba una lista de palabras y devuelva las palabras únicas usando `set`.

Solución:

```
def palabras_unicas(lista):  
    return list(set(lista))  
  
print(palabras_unicas(["python", "java", "python", "c"]))
```

Explicación: `set` elimina duplicados, luego se convierte nuevamente en lista.

Ejercicio 15

Enunciado: Crea una función que reciba un número y devuelva `True` si es palíndromo (igual al revés).

Solución:

```
def es_palindromo(n):  
    s = str(n)  
    return s == s[::-1]  
  
print(es_palindromo(121))  
print(es_palindromo(123))
```

Explicación: Se convierte a cadena y se compara con su inversa usando slicing.

Ejercicio 16

Enunciado: Crea una función que reciba un número y devuelva la suma de los dígitos pares.

Solución:

```
def suma_pares_digitos(n):  
    return sum(int(d) for d in str(n) if int(d)%2==0)  
  
print(suma_pares_digitos(123456))
```

Explicación: Conversión a cadena permite iterar dígito a dígito; `int(d)%2==0` filtra pares.

Ejercicio 17

Enunciado: Crea una función que reciba una lista de números y devuelva un diccionario con "pares" y "impares" sumados.

Solución:

```
def sumar_pares_impares(lista):  
    return {  
        "pares": sum(x for x in lista if x%2==0),  
        "impares": sum(x for x in lista if x%2!=0)  
    }  
  
print(sumar_pares_impares([1,2,3,4,5]))
```

Explicación: Diccionario con suma de elementos filtrados por condición.

Ejercicio 18

Enunciado: Crea una función que reciba una lista de números y devuelva otra lista con solo los primos.

Solución:

```
def primos_lista(lista):
    def es_primo(n):
        if n<2:
            return False
        for i in range(2,int(n**0.5)+1):
            if n%i==0:
                return False
        return True
    return [x for x in lista if es_primo(x)]

print(primos_lista([1,2,3,4,5,6,7]))
```

Explicación: Función interna para verificar primos; comprensión de listas para filtrar.

Ejercicio 19

Enunciado: Crea una función que reciba una función y un número de veces, y devuelva una nueva función que aplique la original ese número de veces.

Solución:

```
def repetir(func, n):
    def nueva_func(x):
        for _ in range(n):
            x = func(x)
        return x
    return nueva_func

doblar = repetir(lambda x: x*2, 3)
print(doblar(2))
```

Explicación: Closure que aplica repetidamente la función `func` al valor.

Ejercicio 20

Enunciado: Crea un decorador que mida el tiempo de ejecución de cualquier función.

Solución:

```
import time

def medir_tiempo(func):
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = func(*args, **kwargs)
        fin = time.time()
        print(f"Tiempo de ejecución: {fin-inicio:.6f} segundos")
        return resultado
    return wrapper

@medir_tiempo
def suma_grande():
    return sum(range(1000000))
```

```
suma_grande()
```

Explicación: Decorador con `time.time()` mide inicio y fin; `*args` y `**kwargs` permiten cualquier firma de función.

Lección 9.: Programación Orientada a Objetos (POO) en Python

1. Introducción a la POO

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el software en **objetos**, los cuales combinan **datos** (atributos) y **funcionalidad** (métodos).

En lugar de centrarnos en funciones que operan sobre datos, la POO nos permite modelar el mundo real, donde cada objeto representa entidades concretas o abstractas.

Ventajas de la POO:

- Modularidad y organización.
 - Reutilización de código mediante herencia.
 - Encapsulamiento de datos y protección de información.
 - Abstracción que facilita modelar problemas complejos.
-

2. Conceptos Clave

1. **Clase:** Plantilla o molde que define las propiedades y comportamientos de los objetos.
 2. **Objeto:** Instancia de una clase.
 3. **Atributo:** Propiedad o característica de un objeto.
 4. **Método:** Función definida dentro de una clase que describe un comportamiento.
 5. **Encapsulamiento:** Ocultar detalles internos y exponer solo lo necesario.
 6. **Herencia:** Permite crear nuevas clases a partir de clases existentes.
 7. **Polimorfismo:** Permite que distintas clases respondan a los mismos métodos de manera diferente.
 8. **Abstracción:** Representación simplificada de entidades complejas.
-

3. Creación de Clases y Objetos

```
# Definición de una clase
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre # atributo
        self.edad = edad     # atributo

    def saludar(self):        # método
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

# Crear un objeto
persona1 = Persona("Ana", 25)
persona1.saludar()
```

Explicación:

- `__init__`: Método constructor que inicializa los atributos del objeto.
 - `self`: Referencia al propio objeto.
 - `persona1` es una instancia de la clase `Persona`.
-

4. Encapsulamiento

Se puede controlar el acceso a los atributos y métodos:

```
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo # atributo privado

    def depositar(self, cantidad):
        self.__saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.__saldo:
            self.__saldo -= cantidad
        else:
            print("Saldo insuficiente")

    def mostrar_saldo(self):
        print(f"Saldo actual: {self.__saldo}")

cuenta = CuentaBancaria("Juan", 1000)
cuenta.depositar(500)
cuenta.retirar(200)
cuenta.mostrar_saldo()
```

- Los atributos privados se indican con `__` delante del nombre.
 - Se accede a ellos solo mediante métodos públicos.
-

5. Herencia

Permite crear clases que heredan atributos y métodos de otras clases.

```
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hablar(self):
        print("El animal hace un sonido")

class Perro(Animal):
    def hablar(self): # polimorfismo
        print(f"{self.nombre} dice: Guau!")

perro1 = Perro("Rex")
perro1.hablar()
```

- Perro hereda de Animal.
 - Se sobrescribe el método hablar para un comportamiento específico (polimorfismo).
-

6. Métodos y Atributos Especiales

- `__str__`: Permite definir cómo se imprime un objeto.
- `__repr__`: Representación oficial de un objeto.
- `__del__`: Destructor (se ejecuta al eliminar un objeto).

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def __str__(self):
        return f"'{self.titulo}' de {self.autor}"

libro1 = Libro("1984", "George Orwell")
print(libro1)
```

7. Composición de Objetos

Un objeto puede contener otros objetos:

```
class Motor:
    def __init__(self, tipo):
        self.tipo = tipo

class Coche:
    def __init__(self, marca, motor):
        self.marca = marca
        self.motor = motor
```

```
motor1 = Motor("V8")
coche1 = Coche("Ford", motor1)
print(f"El coche {coche1.marca} tiene un motor {coche1.motor.tipo}")
```

8. Polimorfismo

Permite tratar objetos de distintas clases de manera uniforme:

```
class Gato(Animal):
    def hablar(self):
        print(f"{self.nombre} dice: Miau!")

animales = [Perro("Rex"), Gato("Misu")]

for animal in animales:
    animal.hablar()
```

Cada objeto responde según su propia implementación de `hablar`.

9. Clases y Métodos de Clase

- **Métodos de clase:** Usan `@classmethod`, afectan a la clase, no al objeto.
- **Métodos estáticos:** Usan `@staticmethod`, no necesitan `self` ni `cls`.

```
class Circulo:
    pi = 3.1416

    def __init__(self, radio):
        self.radio = radio

    @classmethod
    def cambiar_pi(cls, valor):
        cls.pi = valor

    @staticmethod
    def info():
        print("Esta es una clase para calcular áreas de círculos.")

c1 = Circulo(5)
Circulo.info()
Circulo.cambiar_pi(3.14)
print(Circulo.pi)
```

10. Buenas Prácticas en POO

1. **Nombres claros:** Clases con mayúscula inicial (CamelCase), métodos y atributos en minúscula.
2. **Encapsulamiento:** Evitar acceso directo a atributos.
3. **Reutilización de código:** Usar herencia y composición cuando convenga.

4. **Responsabilidad única:** Cada clase debe tener un propósito claro.
 5. **Evitar duplicación:** Métodos comunes deben ser heredados o compartidos.
-

11. Ejercicio Completo de Aplicación

Problema: Crear un sistema para gestionar una biblioteca.

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def mostrar_info(self):
        print(f"'{self.titulo}' de {self.autor}")

class Biblioteca:
    def __init__(self):
        self.libros = []

    def agregar_libro(self, libro):
        self.libros.append(libro)

    def mostrar_libros(self):
        for libro in self.libros:
            libro.mostrar_info()

libro1 = Libro("Cien años de soledad", "Gabriel García Márquez")
libro2 = Libro("1984", "George Orwell")

mi_biblioteca = Biblioteca()
mi_biblioteca.agregar_libro(libro1)
mi_biblioteca.agregar_libro(libro2)

mi_biblioteca.mostrar_libros()
```

Conceptos aplicados:

- Clases y objetos.
 - Métodos y atributos.
 - Composición (Biblioteca contiene Libro).
 - Iteración sobre listas de objetos.
-

12 Ejercicios

12.1 Ejercicios de Python – Programación Orientada a Objetos (Nivel Principiante)

Ejercicio 1

Enunciado: Crea una clase `Persona` con atributos `nombre` y `edad` y un método que muestre: “Hola, me llamo [nombre] y tengo [edad] años”.

Solución:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print(f"Hola, me llamo {self.nombre} y tengo {self.edad} años")

persona1 = Persona("Ana", 25)
persona1.saludar()
```

Explicación: Se crea la clase con constructor `__init__`, se definen los atributos y un método que imprime la información.

Ejercicio 2

Enunciado: Crear una clase `Coche` con atributos `marca` y `modelo` y un método que muestre “Este coche es [marca] [modelo]”.

Solución:

```
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mostrar(self):
        print(f"Este coche es {self.marca} {self.modelo}")

mi_coche = Coche("Toyota", "Corolla")
mi_coche.mostrar()
```

Explicación: Igual que el ejercicio anterior, pero con atributos de coche.

Ejercicio 3

Enunciado: Crea una clase `CuentaBancaria` con atributo `saldo`. Agrega métodos `depositar` y `retirar` que actualicen el saldo.

Solución:

```
class CuentaBancaria:
    def __init__(self, saldo=0):
        self.saldo = saldo

    def depositar(self, cantidad):
        self.saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.saldo:
            self.saldo -= cantidad
        else:
            print("Saldo insuficiente")

cuenta = CuentaBancaria(1000)
cuenta.depositar(500)
cuenta.retirar(200)
print(cuenta.saldo)
```

Explicación: Se manejan atributos y métodos para modificar el estado del objeto.

Ejercicio 4

Enunciado: Crea una clase `Rectangulo` con atributos `ancho` y `alto` y un método que calcule el área.

Solución:

```
class Rectangulo:
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def area(self):
        return self.ancho * self.alto

r = Rectangulo(5, 3)
print(r.area())
```

Explicación: Método que utiliza los atributos para devolver el área.

Ejercicio 5

Enunciado: Crear una clase `Perro` con atributo `nombre` y un método `ladrar` que imprima “Guau!”.

Solución:

```
class Perro:
    def __init__(self, nombre):
        self.nombre = nombre

    def ladrar(self):
        print(f"{self.nombre} dice: Guau!")

mi_perro = Perro("Rex")
mi_perro.ladrar()
```

Explicación: Método que imprime un mensaje usando un atributo.

Ejercicio 6

Enunciado: Crea una clase `Libro` con atributos `título` y `autor` y un método que imprima la información del libro.

Solución:

```
class Libro:
    def __init__(self, título, autor):
        self.título = título
        self.autor = autor

    def info(self):
        print(f"'{self.título}' de {self.autor}")

libro1 = Libro("1984", "George Orwell")
libro1.info()
```

Explicación: Similar a ejercicios anteriores, se aprende a organizar la información de objetos.

Ejercicio 7

Enunciado: Crear una clase `Estudiante` con atributos `nombre` y `notas` (lista). Agregar método que calcule promedio.

Solución:

```
class Estudiante:
    def __init__(self, nombre, notas):
        self.nombre = nombre
        self.notas = notas

    def promedio(self):
        return sum(self.notas)/len(self.notas)

alumno = Estudiante("Luis", [7, 8, 9])
print(alumno.promedio())
```

Explicación: Se utiliza lista como atributo y un método que opera sobre ella.

Ejercicio 8

Enunciado: Crear una clase `Empleado` con atributos `nombre` y `sueldo`. Agrega un método que aumente el sueldo en un porcentaje.

Solución:

```
class Empleado:
    def __init__(self, nombre, sueldo):
        self.nombre = nombre
        self.sueldo = sueldo

    def aumentar_sueldo(self, porcentaje):
        self.sueldo += self.sueldo * porcentaje/100

e = Empleado("Ana", 2000)
e.aumentar_sueldo(10)
print(e.sueldo)
```

Explicación: Método que modifica un atributo usando matemáticas básicas.

Ejercicio 9

Enunciado: Crea una clase `Alumno` que herede de `Persona` y agregue un atributo `curso`.

Solución:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

class Alumno(Persona):
    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)
        self.curso = curso

alumno = Alumno("Pedro", 20, "Matemáticas")
print(alumno.nombre, alumno.edad, alumno.curso)
```

Explicación: Se aplica herencia y `super()` para inicializar la clase base.

Ejercicio 10

Enunciado: Crear una clase `Triangulo` con atributos `base` y `altura` y método `area` que calcule el área.

Solución:

```
class Triangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def area(self):
        return (self.base * self.altura)/2

t = Triangulo(4, 6)
print(t.area())
```

Explicación: Cálculo simple usando los atributos del objeto.

Ejercicio 11

Enunciado: Crear una clase `Mascota` con atributo `nombre` y método `jugar`. Crear subclase `Perro` que sobrescriba `jugar`.

Solución:

```
class Mascota:
    def __init__(self, nombre):
        self.nombre = nombre

    def jugar(self):
        print(f"{self.nombre} está jugando")

class Perro(Mascota):
    def jugar(self):
        print(f"{self.nombre} corre y juega con la pelota")

p = Perro("Rex")
p.jugar()
```

Explicación: Polimorfismo mediante sobreescritura de métodos.

Ejercicio 12

Enunciado: Crea una clase **Banco** que contenga una lista de cuentas (**CuentaBancaria**) y un método para mostrar todos los saldos.

Solución:

```
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.saldo = saldo

class Banco:
    def __init__(self):
        self.cuentas = []

    def agregar_cuenta(self, cuenta):
        self.cuentas.append(cuenta)

    def mostrar_saldos(self):
        for c in self.cuentas:
            print(f"{c.titular}: {c.saldo}")

banco = Banco()
banco.agregar_cuenta(CuentaBancaria("Ana", 1000))
banco.agregar_cuenta(CuentaBancaria("Luis", 1500))
banco.mostrar_saldos()
```

Explicación: Composición: un objeto contiene otros objetos.

Ejercicio 13

Enunciado: Crear clase Calculadora con métodos sumar, restar, multiplicar y dividir.

Solución:

```
class Calculadora:
    def sumar(self, a, b):
        return a + b
    def restar(self, a, b):
        return a - b
    def multiplicar(self, a, b):
        return a * b
    def dividir(self, a, b):
        if b != 0:
            return a / b
        else:
            return "Error: división por cero"

calc = Calculadora()
print(calc.sumar(5,3))
print(calc.dividir(10,2))
```

Explicación: Métodos que reciben parámetros y devuelven resultados.

Ejercicio 14

Enunciado: Crear una clase `Vehiculo` y subclases `Coche` y `Moto` que tengan método `tipo` que indique su tipo.

Solución:

```
class Vehiculo:
    def tipo(self):
        print("Vehículo genérico")

class Coche(Vehiculo):
    def tipo(self):
        print("Soy un coche")

class Moto(Vehiculo):
    def tipo(self):
        print("Soy una moto")

c = Coche()
m = Moto()
c.tipo()
m.tipo()
```

Explicación: Polimorfismo básico: mismo método con distinto comportamiento.

Ejercicio 15

Enunciado: Crear clase `Producto` con atributos `nombre` y `precio` y método que aplique descuento en porcentaje.

Solución:

```
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def aplicar_descuento(self, porcentaje):
        self.precio -= self.precio * porcentaje / 100

p = Producto("Camisa", 50)
p.aplicar_descuento(10)
print(p.precio)
```

Explicación: Se modifica un atributo con una fórmula simple.

Ejercicio 16

Enunciado: Crear clase `Cuenta` con atributo `titular` y método `mostrar_titular`. Crear 2 instancias y mostrar titulares.

Solución:

```
class Cuenta:
    def __init__(self, titular):
        self.titular = titular

    def mostrar_titular(self):
        print(self.titular)

c1 = Cuenta("Ana")
c2 = Cuenta("Luis")
c1.mostrar_titular()
c2.mostrar_titular()
```

Explicación: Instanciación de objetos y uso de métodos para mostrar atributos.

Ejercicio 17

Enunciado: Crear clase `Animal` con método `hacer_sonido` que imprima “Sonido genérico”. Crear subclase `Gato` que imprima “Miau”.

Solución:

```
class Animal:
    def hacer_sonido(self):
        print("Sonido genérico")

class Gato(Animal):
    def hacer_sonido(self):
        print("Miau")
```

```
a = Animal()
g = Gato()
a.hacer_sonido()
g.hacer_sonido()
```

Explicación: Polimorfismo: se sobrescribe método en subclase.

Ejercicio 18

Enunciado: Crear clase `Telefono` con método `llamar` que reciba un número y lo imprima.

Solución:

```
class Telefono:
    def llamar(self, numero):
        print(f"Llamando al {numero}...")

t = Telefono()
t.llamar("123456789")
```

Explicación: Método con parámetro que usa atributo local.

Ejercicio 19

Enunciado: Crear clase `Profesor` con atributos `nombre` y `materia`, y método que imprima “[nombre] enseña [materia]”.

Solución:

```
class Profesor:
    def __init__(self, nombre, materia):
        self.nombre = nombre
        self.materia = materia

    def info(self):
        print(f"{self.nombre} enseña {self.materia}")

p = Profesor("Carlos", "Matemáticas")
p.info()
```

Explicación: Clase simple con atributos y método de información.

Ejercicio 20

Enunciado: Crear clase `Jugador` con atributo `nombre` y método `marcar_gol` que imprima “[nombre] ha marcado un gol”.

Solución:

```
class Jugador:
    def __init__(self, nombre):
        self.nombre = nombre

    def marcar_gol(self):
        print(f"{self.nombre} ha marcado un gol")

j = Jugador("Leo Messi")
j.marcar_gol()
```

Explicación: Método que interactúa con un atributo para generar mensaje.

12.2 Ejercicios de Python – Programación Orientada a Objetos (Nivel Medio)

Ejercicio 1

Enunciado: Crear una clase `Persona` con atributos `nombre`, `edad` y método `cumplir_anios` que aumente la edad en 1.

Solución:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def cumplir_anios(self):
        self.edad += 1

p = Persona("Ana", 25)
p.cumplir_anios()
print(p.edad)
```

Explicación: Se aplica modificación de un atributo mediante un método de la clase.

Ejercicio 2

Enunciado: Crear clase CuentaBancaria con titular y saldo. Agregar métodos depositar, retirar y transferir a otra cuenta.

Solución:

```
class CuentaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, cantidad):
        self.saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.saldo:
            self.saldo -= cantidad
        else:
            print("Saldo insuficiente")

    def transferir(self, otra_cuenta, cantidad):
        if cantidad <= self.saldo:
            self.retirar(cantidad)
            otra_cuenta.depositar(cantidad)
        else:
            print("Saldo insuficiente para transferir")

cuenta1 = CuentaBancaria("Ana", 1000)
cuenta2 = CuentaBancaria("Luis", 500)
cuenta1.transferir(cuenta2, 300)
print(cuenta1.saldo, cuenta2.saldo)
```

Explicación: Uso de métodos que interactúan con otros objetos de la misma clase.

Ejercicio 3

Enunciado: Crear clase Empleado con atributos nombre y sueldo. Crear subclase Gerente con método bono que aumente el sueldo en 20%.

Solución:

```
class Empleado:
    def __init__(self, nombre, sueldo):
        self.nombre = nombre
        self.sueldo = sueldo
```

```
class Gerente(Empleado):
    def bono(self):
        self.sueldo *= 1.2
```

```
g = Gerente("Carlos", 3000)
g.bono()
print(g.sueldo)
```

Explicación: Herencia y modificación de atributos en subclase.

Ejercicio 4

Enunciado: Crear clase `Rectangulo` con método `area` y `perimetro`. Crear subclase `Cuadrado` que solo necesite un lado.

Solución:

```
class Rectangulo:
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def area(self):
        return self.ancho * self.alto

    def perimetro(self):
        return 2*(self.ancho + self.alto)

class Cuadrado(Rectangulo):
    def __init__(self, lado):
        super().__init__(lado, lado)

c = Cuadrado(5)
print(c.area(), c.perimetro())
```

Explicación: Herencia con constructor `super ( )` y especialización de atributos.

Ejercicio 5

Enunciado: Crear clase `Alumno` con método `promedio`. Crear lista de objetos `Alumno` y calcular el promedio general.

Solución:

```
class Alumno:
    def __init__(self, nombre, notas):
        self.nombre = nombre
        self.notas = notas

    def promedio(self):
        return sum(self.notas)/len(self.notas)

alumnos = [
    Alumno("Ana", [8, 9, 7]),
    Alumno("Luis", [6, 7, 8])
]

promedio_general = sum(a.promedio() for a in alumnos)/len(alumnos)
print(promedio_general)
```

Explicación: Manejo de listas de objetos y cálculo de valores agregados.

Ejercicio 6

Enunciado: Crear clase `Vehiculo` con método `mover`. Crear subclases `Coche` y `Bicicleta` que implementen `mover` de forma diferente.

Solución:

```
class Vehiculo:
    def mover(self):
        print("El vehículo se mueve")

class Coche(Vehiculo):
    def mover(self):
        print("El coche está conduciendo")

class Bicicleta(Vehiculo):
    def mover(self):
        print("La bicicleta está pedaleando")

v = [Coche(), Bicicleta()]
for veh in v:
    veh.mover()
```

Explicación: Polimorfismo: mismo método con comportamiento distinto según clase.

Ejercicio 7

Enunciado: Crear clase `Producto` con atributos `nombre` y `precio`. Crear método `aplicar_descuento` que acepte porcentaje.

Solución:

```
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def aplicar_descuento(self, porcentaje):
        self.precio *= (1 - porcentaje/100)

p = Producto("Laptop", 1500)
p.aplicar_descuento(10)
print(p.precio)
```

Explicación: Uso de métodos que alteran atributos con cálculo matemático.

Ejercicio 8

Enunciado: Crear clase **Banco** que tenga lista de **CuentaBancaria** y método para mostrar saldo total del banco.

Solución:

```
class Banco:
    def __init__(self):
        self.cuentas = []

    def agregar_cuenta(self, cuenta):
        self.cuentas.append(cuenta)

    def saldo_total(self):
        return sum(c.saldo for c in self.cuentas)

banco = Banco()
banco.agregar_cuenta(CuentaBancaria("Ana", 1000))
banco.agregar_cuenta(CuentaBancaria("Luis", 500))
print(banco.saldo_total())
```

Explicación: Composición: un objeto contiene otros objetos y se agregan métodos que los operan en conjunto.

Ejercicio 9

Enunciado: Crear clase `Libro` y subclase `LibroDigital` que agregue atributo `tamano_MB`.

Solución:

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

class LibroDigital(Libro):
    def __init__(self, titulo, autor, tamano_MB):
        super().__init__(titulo, autor)
        self.tamano_MB = tamano_MB

libro = LibroDigital("1984", "George Orwell", 5)
print(libro.titulo, libro.autor, libro.tamano_MB)
```

Explicación: Subclase que agrega atributo extra usando `super()`.

Ejercicio 10

Enunciado: Crear clase `Empleado` con método `calcular_sueldo`. Crear subclases `Fijo` y `PorHora` con implementación distinta.

Solución:

```
class Empleado:
    def calcular_sueldo(self):
        pass

class Fijo(Empleado):
    def __init__(self, sueldo):
        self.sueldo = sueldo
    def calcular_sueldo(self):
        return self.sueldo

class PorHora(Empleado):
    def __init__(self, horas, tarifa):
        self.horas = horas
        self.tarifa = tarifa
    def calcular_sueldo(self):
        return self.horas * self.tarifa

nomina = [Fijo(2000), PorHora(120, 15)]
for e in nomina:
    print(e.calcular_sueldo())
```

Explicación: Polimorfismo avanzado: mismo método, distinto comportamiento según clase.

Ejercicio 11

Enunciado: Crear clase `Animal` con método `hablar`. Crear subclases `Perro` y `Gato` que sobrescriban `hablar`.

Solución:

```
class Animal:
    def hablar(self):
        pass

class Perro(Animal):
    def hablar(self):
        print("Guau!")

class Gato(Animal):
    def hablar(self):
        print("Miau!")

animales = [Perro(), Gato()]
for a in animales:
    a.hablar()
```

Explicación: Uso de polimorfismo con lista de objetos de distintas subclases.

Ejercicio 12

Enunciado: Crear clase `Telefono` con atributo `numero` y método `llamar`. Crear lista de teléfonos y realizar llamadas a todos.

Solución:

```
class Telefono:
    def __init__(self, numero):
        self.numero = numero

    def llamar(self):
        print(f"Llamando al {self.numero}")

telefonos = [Telefono("123"), Telefono("456")]
for t in telefonos:
    t.llamar()
```

Explicación: Iteración sobre lista de objetos y uso de métodos.

Ejercicio 13

Enunciado: Crear clase `Producto` con método `info`. Crear lista de productos y mostrar información de todos.

Solución:

```
productos = [
    Producto("Camisa", 50),
    Producto("Pantalón", 80)
]

for p in productos:
    print(f"{p.nombre} - ${p.precio}")
```

Explicación: Manejo de listas y atributos de objetos.

Ejercicio 14

Enunciado: Crear clase `Figura` con método `area` (vacío). Crear subclases `Rectangulo` y `Circulo` que implementen `area`.

Solución:

```
import math

class Figura:
    def area(self):
        pass

class Rectangulo(Figura):
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto
    def area(self):
        return self.ancho*self.alto

class Circulo(Figura):
    def __init__(self, radio):
        self.radio = radio
    def area(self):
        return math.pi*self.radio**2

figuras = [Rectangulo(4,5), Circulo(3)]
for f in figuras:
    print(f.area())
```

Explicación: Polimorfismo: método común implementado de distinta manera.

Ejercicio 15

Enunciado: Crear clase `Alumno` con método `aprobo` que devuelva `True` si promedio ≥ 6 .

Solución:

```
class Alumno:
    def __init__(self, nombre, notas):
        self.nombre = nombre
        self.notas = notas

    def promedio(self):
        return sum(self.notas)/len(self.notas)

    def aprobo(self):
        return self.promedio() >= 6

a = Alumno("Luis", [5,7,6])
print(a.aprobo())
```

Explicación: Método que retorna valor booleano según atributo.

Ejercicio 16

Enunciado: Crear clase `Juego` con método `sumar_puntos` y atributo `puntos`. Crear método que muestre nivel según puntos.

Solución:

```
class Juego:
    def __init__(self):
        self.puntos = 0

    def sumar_puntos(self, pts):
        self.puntos += pts

    def nivel(self):
        if self.puntos < 10:
            return "Principiante"
        elif self.puntos < 20:
            return "Intermedio"
        else:
            return "Avanzado"

j = Juego()
j.sumar_puntos(15)
print(j.nivel())
```

Explicación: Método que evalúa atributos y devuelve un valor categórico.

Ejercicio 17

Enunciado: Crear clase `Agenda` que almacene lista de `Contacto` y tenga métodos `agregar` y `mostrar_todos`.

Solución:

```
class Contacto:
    def __init__(self, nombre, telefono):
        self.nombre = nombre
        self.telefono = telefono

class Agenda:
    def __init__(self):
        self.contactos = []

    def agregar(self, contacto):
        self.contactos.append(contacto)

    def mostrar_todos(self):
        for c in self.contactos:
            print(c.nombre, c.telefono)

agenda = Agenda()
agenda.agregar(Contacto("Ana", "123"))
agenda.agregar(Contacto("Luis", "456"))
agenda.mostrar_todos()
```

Explicación: Composición de objetos y manejo de listas.

Ejercicio 18

Enunciado: Crear clase CuentaBancaria con atributo privado `__saldo` y métodos depositar, retirar, mostrar_saldo.

Solución:

```
class CuentaBancaria:
    def __init__(self, saldo=0):
        self.__saldo = saldo

    def depositar(self, cantidad):
        self.__saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.__saldo:
            self.__saldo -= cantidad
        else:
            print("Saldo insuficiente")

    def mostrar_saldo(self):
        print(self.__saldo)

c = CuentaBancaria(1000)
c.depositar(500)
c.retirar(200)
c.mostrar_saldo()
```

Explicación: Encapsulamiento de atributos privados y métodos de acceso.

Ejercicio 19

Enunciado: Crear clase `Estudiante` con método `notas_mayores` que devuelva lista de notas mayores a 6.

Solución:

```
class Estudiante:
    def __init__(self, nombre, notas):
        self.nombre = nombre
        self.notas = notas

    def notas_mayores(self):
        return [n for n in self.notas if n>6]

e = Estudiante("Ana", [5, 7, 8, 6])
print(e.notas_mayores())
```

Explicación: Uso de listas y comprensión para filtrar atributos.

Ejercicio 20

Enunciado: Crear clase `Profesor` con atributo `cursos` (lista). Crear método `agregar_curso` y `mostrar_cursos`.

Solución:

```
class Profesor:
    def __init__(self, nombre):
        self.nombre = nombre
        self.cursos = []

    def agregar_curso(self, curso):
        self.cursos.append(curso)

    def mostrar_cursos(self):
        print(self.cursos)

p = Profesor("Carlos")
p.agregar_curso("Matemáticas")
p.agregar_curso("Física")
p.mostrar_cursos()
```

Explicación: Método para manipular listas de atributos dentro de la clase.

12.3 Ejercicios de Python – Programación Orientada a Objetos (Nivel Avanzado)

Ejercicio 1

Enunciado: Crear clase abstracta `Figura` con método `area`. Crear subclases `Rectangulo` y `Circulo` que implementen `area`.

Solución:

```
from abc import ABC, abstractmethod
import math

class Figura(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangulo(Figura):
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto
    def area(self):
        return self.ancho * self.alto

class Circulo(Figura):
    def __init__(self, radio):
        self.radio = radio
    def area(self):
        return math.pi * self.radio**2

r = Rectangulo(5,3)
c = Circulo(2)
print(r.area(), c.area())
```

Explicación: Se usa clase abstracta y subclases implementan métodos obligatorios.

Ejercicio 2

Enunciado: Crear clase Empleado abstracta con método calcular_sueldo. Crear subclases Fijo y PorHora.

Solución:

```
from abc import ABC, abstractmethod

class Empleado(ABC):
    @abstractmethod
    def calcular_sueldo(self):
        pass

class Fijo(Empleado):
    def __init__(self, sueldo):
        self.sueldo = sueldo
    def calcular_sueldo(self):
        return self.sueldo

class PorHora(Empleado):
    def __init__(self, horas, tarifa):
        self.horas = horas
        self.tarifa = tarifa
    def calcular_sueldo(self):
        return self.horas * self.tarifa

nomina = [Fijo(2000), PorHora(100, 15)]
for e in nomina:
    print(e.calcular_sueldo())
```

Explicación: Polimorfismo avanzado mediante herencia abstracta.

Ejercicio 3

Enunciado: Crear clase **Banco** que gestione múltiples **CuentaBancaria**. Incluir método de transferencia y control de errores.

Solución:

```
class CuentaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, cantidad):
        self.saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.saldo:
            self.saldo -= cantidad
        else:
            raise ValueError("Saldo insuficiente")

class Banco:
    def __init__(self):
        self.cuentas = []

    def agregar_cuenta(self, cuenta):
        self.cuentas.append(cuenta)

    def transferir(self, origen, destino, cantidad):
        try:
            origen.retirar(cantidad)
            destino.depositar(cantidad)
        except ValueError as e:
            print(e)

c1 = CuentaBancaria("Ana", 1000)
c2 = CuentaBancaria("Luis", 500)
banco = Banco()
banco.agregar_cuenta(c1)
```

```
banco.agregar_cuenta(c2)
```

```
banco.transferir(c1,c2,1200)
```

Explicación: Composición, manejo de errores y métodos que operan sobre objetos de la misma clase.

Ejercicio 4

Enunciado: Crear clase `Vehiculo` y subclases `Coche`, `Moto`. Implementar herencia múltiple con `VehiculoVolador`.

Solución:

```
class Vehiculo:
    def mover(self):
        print("Vehículo se mueve")

class VehiculoVolador:
    def volar(self):
        print("Vehículo vuela")

class Avion(Vehiculo, VehiculoVolador):
    pass

avion = Avion()
avion.mover()
avion.volar()
```

Explicación: Herencia múltiple: `Avion` hereda métodos de dos clases.

Ejercicio 5

Enunciado: Crear clase `Producto` con método estático `descuento` que calcule precio con porcentaje.

Solución:

```
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    @staticmethod
    def descuento(precio, porcentaje):
        return precio*(1-porcentaje/100)

p = Producto("Laptop", 1500)
print(Producto.descuento(p.precio,10))
```

Explicación: Métodos estáticos que no requieren instancia para ejecutarse.

Ejercicio 6

Enunciado: Crear clase `Cuenta` con atributo privado `__saldo` y propiedades para leer y modificar con validación.

Solución:

```
class Cuenta:
    def __init__(self, saldo):
        self.__saldo = saldo

    @property
    def saldo(self):
        return self.__saldo

    @saldo.setter
    def saldo(self, valor):
        if valor >= 0:
            self.__saldo = valor
        else:
            print("Saldo no puede ser negativo")

c = Cuenta(1000)
c.saldo = -100
print(c.saldo)
```

Explicación: Encapsulamiento y validación mediante `@property` y `@setter`.

Ejercicio 7

Enunciado: Crear clase `Libro` con método `__str__` para mostrar información legible.

Solución:

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def __str__(self):
        return f'"{self.titulo}" por {self.autor}'

libro = Libro("1984", "George Orwell")
print(libro)
```

Explicación: Método especial `__str__` permite impresión legible del objeto.

Ejercicio 8

Enunciado: Crear clase `Empleado` y subclases `Manager` y `Developer` usando polimorfismo para trabajar.

Solución:

```
class Empleado:
    def trabajar(self):
        pass

class Manager(Empleado):
    def trabajar(self):
        print("Planificando tareas")

class Developer(Empleado):
    def trabajar(self):
        print("Programando código")

equipo = [Manager(), Developer()]
for e in equipo:
    e.trabajar()
```

Explicación: Polimorfismo: mismo método con comportamiento distinto.

Ejercicio 9

Enunciado: Crear clase `Producto` con método de clase `total_iva` que calcule precio + IVA.

Solución:

```
class Producto:
    iva = 0.21

    def __init__(self, precio):
        self.precio = precio

    @classmethod
    def total_iva(cls, precio):
        return precio * (1+cls.iva)

p = Producto(100)
print(Producto.total_iva(p.precio))
```

Explicación: Métodos de clase acceden a atributos de clase.

Ejercicio 10

Enunciado: Crear clase abstracta `Volador` con método `volar`. Crear `Pajaro` y `Avion`.

Solución:

```
from abc import ABC, abstractmethod

class Volador(ABC):
    @abstractmethod
    def volar(self):
        pass

class Pajaro(Volador):
    def volar(self):
        print("El pájaro vuela")

class Avion(Volador):
    def volar(self):
        print("El avión despegó")

objetos = [Pajaro(), Avion()]
for o in objetos:
    o.volar()
```

Explicación: Polimorfismo con clase abstracta.

Ejercicio 11

Enunciado: Crear clase `Banco` con método estático `interes` que calcule interés sobre saldo.

Solución:

```
class Banco:
    @staticmethod
    def interes(saldo, tasa):
        return saldo*tasa/100

print(Banco.interes(1000,5))
```

Explicación: Método estático no necesita instancia.

Ejercicio 12

Enunciado: Crear clase `CuentaBancaria` que registre cada operación (composición con lista de strings).

Solución:

```
class CuentaBancaria:
    def __init__(self):
        self.saldo = 0
        self.operaciones = []

    def depositar(self, cantidad):
        self.saldo += cantidad
        self.operaciones.append(f"Depósito {cantidad}")

    def retirar(self, cantidad):
        self.saldo -= cantidad
        self.operaciones.append(f"Retiro {cantidad}")

c = CuentaBancaria()
c.depositar(100)
c.retirar(50)
print(c.operaciones)
```

Explicación: Composición: un atributo almacena historial de operaciones.

Ejercicio 13

Enunciado: Crear clase `Jugador` con método `__repr__` que muestre información del objeto para desarrolladores.

Solución:

```
class Jugador:
    def __init__(self, nombre, goles):
        self.nombre = nombre
        self.goles = goles

    def __repr__(self):
        return f"Jugador({self.nombre},{self.goles})"

j = Jugador("Messi",30)
print(j)
```

Explicación: `__repr__` muestra la representación oficial del objeto.

Ejercicio 14

Enunciado: Crear clase `Agenda` con método `buscar_contacto` que use filtrado en lista de objetos `Contacto`.

Solución:

```
class Contacto:
    def __init__(self, nombre, telefono):
        self.nombre = nombre
        self.telefono = telefono

class Agenda:
    def __init__(self):
        self.contactos = []

    def agregar(self, c):
        self.contactos.append(c)

    def buscar_contacto(self, nombre):
        return [c for c in self.contactos if c.nombre==nombre]

agenda = Agenda()
agenda.agregar(Contacto("Ana", "123"))
print(agenda.buscar_contacto("Ana"))
```

Explicación: Composición y filtrado de listas de objetos.

Ejercicio 15

Enunciado: Crear clase `Logger` como singleton (una sola instancia).

Solución:

```
class Logger:
    _instancia = None
    def new(cls):
        if cls._instancia is None:
            cls._instancia = super().new(cls)
        return cls._instancia

l1 = Logger()
l2 = Logger()
print(l1 is l2)

**Explicación:** Patrón singleton: todas las variables apuntan a la misma
instancia.
```

Ejercicio 16

Enunciado: Crear clase `Vehiculo` con método privado `__chequear` y método público `arrancar` que lo llame.

Solución:

```
python

class Vehiculo:
    def __chequear(self):
        print("Chequeando motor")

    def arrancar(self):
        self.__chequear()
        print("Vehículo arrancado")

v = Vehiculo()
v.arrancar()
```

Explicación: Encapsulamiento de métodos privados.

Ejercicio 17

Enunciado: Crear clase Poligono y subclases que implementen polimorfismo con método `perimetro`.

Solución:

```
class Poligono:
    def perimetro(self):
        pass

class Triangulo(Poligono):
    def __init__(self, lados):
        self.lados = lados
    def perimetro(self):
        return sum(self.lados)

class Cuadrado(Poligono):
    def __init__(self, lado):
        self.lado = lado
    def perimetro(self):
        return 4*self.lado

figs = [Triangulo([3,4,5]), Cuadrado(5)]
for f in figs:
    print(f.perimetro())
```

Explicación: Polimorfismo avanzado: distintas subclases implementan mismo método.

Ejercicio 18

Enunciado: Crear decorador dentro de clase que registre ejecución de método.

Solución:

```
class Test:
    def registrar(func):
        def wrapper(*args, **kwargs):
            print(f"Llamando {func.__name__}")
            return func(*args, **kwargs)
        return wrapper

    @registrar
    def saludar(self):
        print("Hola")

t = Test()
t.saludar()
```

Explicación: Decoradores dentro de clases para modificar comportamiento de métodos.

Ejercicio 19

Enunciado: Crear clase `Estudiante` con métodos estáticos para validar nota entre 0 y 10.

Solución:

```
class Estudiante:
    def __init__(self, nombre, nota):
        self.nombre = nombre
        self.nota = nota

    @staticmethod
    def validar_nota(nota):
        return 0 <= nota <= 10

print(Estudiante.validar_nota(8))
print(Estudiante.validar_nota(12))
```

Explicación: Método estático usado para validación sin necesidad de instancia.

Ejercicio 20

Enunciado: Crear clase `Archivo` con método `leer` que maneje excepciones si archivo no existe.

Solución:

```
class Archivo:
    def __init__(self, nombre):
        self.nombre = nombre

    def leer(self):
        try:
            with open(self.nombre, "r") as f:
                return f.read()
        except FileNotFoundError:
            print("Archivo no encontrado")

a = Archivo("inexistente.txt")
a.leer()
```

Explicación: Manejo de excepciones en métodos de clase.

Lección 10.: Tratamiento de ficheros CSV en Python

1. Introducción a CSV

- **CSV (Comma Separated Values)** es un formato de texto simple para almacenar datos tabulares.
- Cada línea representa un registro, y los valores están separados por comas (,), aunque otros separadores también son posibles (; , \t).
- Ventajas:
 - Legible por humanos.
 - Compatible con Excel, Google Sheets y bases de datos.
- Limitaciones:
 - No almacena tipos de datos complejos.
 - No tiene soporte nativo para jerarquías o estructuras anidadas.

Ejemplo de un archivo `ejemplo.csv`:

```
Nombre,Edad,Email
Ana,25,ana@example.com
Luis,30,luis@example.com
Marta,22,marta@example.com
```

2. Módulo csv en Python

Python incluye un módulo estándar llamado `csv` para trabajar con este tipo de archivos.

2.1 Lectura de CSV

```
import csv

with open("ejemplo.csv", newline='') as archivo:
    lector = csv.reader(archivo)
    for fila in lector:
        print(fila)
```

Explicación:

- `csv.reader` devuelve un iterador de listas.
- Cada fila se representa como una lista de cadenas (`['Ana', '25', 'ana@example.com']`).

Nota: La primera fila suele contener los nombres de las columnas (headers).

2.2 Lectura con DictReader

```
import csv

with open("ejemplo.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        print(f"{fila['Nombre']} tiene {fila['Edad']} años")
```

Explicación:

- DictReader devuelve cada fila como un diccionario {columna: valor}.
 - Facilita acceder a los datos por nombre de columna.
-

3. Escritura de CSV

3.1 Usando writer

```
import csv

datos = [
    ['Nombre', 'Edad', 'Email'],
    ['Juan', 28, 'juan@example.com'],
    ['Lucía', 24, 'lucia@example.com']
]

with open("salida.csv", "w", newline='') as archivo:
    escritor = csv.writer(archivo)
    escritor.writerows(datos)
```

Explicación:

- writerows permite escribir varias filas a la vez.
 - newline='' evita líneas en blanco adicionales en Windows.
-

3.2 Usando DictWriter

```
import csv

datos = [
    {'Nombre': 'Juan', 'Edad': 28, 'Email': 'juan@example.com'},
    {'Nombre': 'Lucía', 'Edad': 24, 'Email': 'lucia@example.com'}
]

with open("salida_dict.csv", "w", newline='') as archivo:
    campos = ['Nombre', 'Edad', 'Email']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(datos)
```

Explicación:

- DictWriter permite escribir diccionarios directamente.

- `writerheader()` escribe los nombres de columna automáticamente.
-

4. Configuración de delimitadores y formatos

- CSV no siempre usa comas. Puedes usar otros separadores:

```
with open("datos_puntoycoma.csv", newline='') as archivo:
    lector = csv.reader(archivo, delimiter=',')
    for fila in lector:
        print(fila)
```

- Otros parámetros útiles:
 - `quotechar='"'`: define el carácter para cadenas con comas.
 - `escapechar='\\'`: permite escapar caracteres especiales.
-

5. Manipulación de datos CSV

5.1 Filtrar filas

```
import csv

with open("ejemplo.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    mayores_25 = [fila for fila in lector if int(fila['Edad']) > 25]

print(mayores_25)
```

Explicación: Se convierte la columna `Edad` a entero para poder filtrar.

5.2 Modificar datos y escribir nuevo CSV

```
import csv

# Leer
with open("ejemplo.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    datos = [fila for fila in lector]

# Modificar
for fila in datos:
    fila['Edad'] = str(int(fila['Edad']) + 1)

# Guardar
with open("ejemplo_actualizado.csv", "w", newline='') as archivo:
    campos = ['Nombre', 'Edad', 'Email']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(datos)
```

Explicación:

- Leer datos en memoria, modificar y guardar un nuevo archivo.
 - Se deben convertir los tipos al guardarlos (`str` en este caso).
-

5.3 Ordenar datos

```
import csv

with open("ejemplo.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    datos = sorted(lector, key=lambda x: int(x['Edad']))

for fila in datos:
    print(fila)
```

Explicación: Se usa `sorted` con `key` para ordenar por columna numérica.

6. Buenas prácticas al tratar CSV

1. Siempre usar `with open(...)` para cerrar archivos automáticamente.
2. Convertir tipos de datos explícitamente (`str` → `int/float`).
3. Manejar excepciones al abrir archivos:

```
try:
    with open("no_existe.csv") as f:
        pass
except FileNotFoundError:
    print("El archivo no existe")
```

4. Usar `DictReader/DictWriter` para mayor claridad en código.
 5. Evitar modificar archivos originales; mejor guardar en uno nuevo.
-

7. CSV con pandas (opcional, más potente)

```
import pandas as pd

# Leer CSV
df = pd.read_csv("ejemplo.csv")
print(df)

# Filtrar
mayores_25 = df[df['Edad'] > 25]
print(mayores_25)

# Modificar y guardar
df['Edad'] = df['Edad'] + 1
df.to_csv("ejemplo_actualizado_pandas.csv", index=False)
```

Ventajas de pandas:

- Manejo de grandes datasets.
 - Operaciones de filtrado, agregación y estadísticas fácilmente.
 - Exportación a múltiples formatos.
-

8. Ejercicios

8.1 Ejercicios de Python – Tratamiento de ficheros CSV (Nivel Principiante)

Ejercicio 1

Enunciado: Leer un archivo CSV llamado `personas.csv` y mostrar todas sus filas.

Solución:

```
import csv

with open("personas.csv", newline='') as archivo:
    lector = csv.reader(archivo)
    for fila in lector:
        print(fila)
```

Explicación: `csv.reader` devuelve cada fila como lista de valores. Se recorre con un bucle `for`.

Ejercicio 2

Enunciado: Leer `personas.csv` usando `DictReader` y mostrar el nombre de cada persona.

Solución:

```
import csv

with open("personas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        print(fila['Nombre'])
```

Explicación: `DictReader` permite acceder a las columnas por su nombre.

Ejercicio 3

Enunciado: Crear un CSV `frutas.csv` con las columnas: Nombre, Cantidad.

Solución:

```
import csv

datos = [
    ['Nombre', 'Cantidad'],
    ['Manzana', 10],
    ['Banana', 5],
    ['Naranja', 8]
]

with open("frutas.csv", "w", newline='') as archivo:
    escritor = csv.writer(archivo)
    escritor.writerows(datos)
```

Explicación: `writerows` escribe varias filas de una vez.

Ejercicio 4

Enunciado: Crear un CSV usando `DictWriter` con productos y precios.

Solución:

```
import csv

productos = [
    {'Producto': 'Lapiz', 'Precio': 0.5},
    {'Producto': 'Cuaderno', 'Precio': 2}
]

with open("productos.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Precio']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(productos)
```

Explicación: `DictWriter` permite escribir diccionarios directamente y `writeheader()` crea la fila de encabezados.

Ejercicio 5

Enunciado: Leer `frutas.csv` y mostrar solo las frutas con cantidad mayor a 6.

Solución:

```
import csv

with open("frutas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Cantidad']) > 6:
            print(fila['Nombre'])
```

Explicación: Convertimos `Cantidad` a entero y filtramos con un `if`.

Ejercicio 6

Enunciado: Añadir una nueva fila al CSV `frutas.csv`.

Solución:

```
import csv

with open("frutas.csv", "a", newline='') as archivo:
    escritor = csv.writer(archivo)
    escritor.writerow(['Pera', 12])
```

Explicación: Abrir en modo `"a"` para **añadir** sin sobrescribir.

Ejercicio 7

Enunciado: Leer `productos.csv` y calcular el precio total de todos los productos.

Solución:

```
import csv

total = 0

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        total += float(fila['Precio'])

print(total)
```

Explicación: Convertimos precios a `float` y sumamos.

Ejercicio 8

Enunciado: Leer un CSV separado por ; llamado `datos.csv`.

Solución:

```
import csv

with open("datos.csv", newline='') as archivo:
    lector = csv.reader(archivo, delimiter=';')
    for fila in lector:
        print(fila)
```

Explicación: Se usa `delimiter=';'` para archivos que no usan comas.

Ejercicio 9

Enunciado: Leer `personas.csv` y crear una lista de diccionarios con los datos.

Solución:

```
import csv

datos = []

with open("personas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        datos.append(fila)

print(datos)
```

Explicación: Se almacenan todas las filas en memoria como diccionarios.

Ejercicio 10

Enunciado: Crear un CSV `notas.csv` con estudiantes y sus notas.

Solución:

```
import csv

notas = [
    {'Estudiante': 'Ana', 'Nota': 8},
    {'Estudiante': 'Luis', 'Nota': 6},
    {'Estudiante': 'Marta', 'Nota': 9}
]

with open("notas.csv", "w", newline='') as archivo:
    campos = ['Estudiante', 'Nota']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(notas)
```

Explicación: Uso de `DictWriter` para mayor claridad.

Ejercicio 11

Enunciado: Leer `notas.csv` y mostrar los estudiantes que aprobaron (nota ≥ 7).

Solución:

```
import csv

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Nota']) >= 7:
            print(fila['Estudiante'])
```

Explicación: Filtrado básico usando `if` y conversión a entero.

Ejercicio 12

Enunciado: Modificar `notas.csv` aumentando 1 punto a cada estudiante y guardar en nuevo archivo.

Solución:

```
import csv

datos = []

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        fila['Nota'] = str(int(fila['Nota']) + 1)
        datos.append(fila)

with open("notas_actualizadas.csv", "w", newline='') as archivo:
    campos = ['Estudiante', 'Nota']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(datos)
```

Explicación: Leer, modificar en memoria y escribir en un nuevo archivo.

Ejercicio 13

Enunciado: Ordenar `notas.csv` por nota descendente y mostrar los resultados.

Solución:

```
import csv

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    datos = sorted(lector, key=lambda x: int(x['Nota']), reverse=True)

for fila in datos:
    print(fila)
```

Explicación: Uso de `sorted` con `key` para ordenar por columna numérica.

Ejercicio 14

Enunciado: Contar cuántos estudiantes hay en `notas.csv`.

Solución:

```
import csv

contador = 0

with open("notas.csv", newline='') as archivo:
    lector = csv.reader(archivo)
    next(lector) # saltar encabezado
    for fila in lector:
        contador += 1

print(contador)
```

Explicación: Se recorre cada fila y se incrementa contador; `next(lector)` salta la primera fila de encabezado.

Ejercicio 15

Enunciado: Crear un CSV `empleados.csv` con nombre y salario, y leerlo mostrando salario mayor a 2000.

Solución:

```
import csv

empleados = [
    {'Nombre': 'Ana', 'Salario': 2500},
    {'Nombre': 'Luis', 'Salario': 1800}
]

with open("empleados.csv", "w", newline='') as archivo:
    campos = ['Nombre', 'Salario']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(empleados)

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Salario']) > 2000:
            print(fila['Nombre'])
```

Explicación: Escritura y lectura con filtrado.

Ejercicio 16

Enunciado: Leer `frutas.csv` y calcular la suma total de la columna Cantidad.

Solución:

```
import csv

total = 0

with open("frutas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        total += int(fila['Cantidad'])

print(total)
```

Explicación: Recorrer cada fila y acumular valores numéricos.

Ejercicio 17

Enunciado: Crear un CSV `ciudades.csv` con nombre y población, y leerlo mostrando ciudades con más de 1 millón de habitantes.

Solución:

```
import csv

ciudades = [
    {'Ciudad': 'Madrid', 'Poblacion': 3265000},
    {'Ciudad': 'Zaragoza', 'Poblacion': 675000}
]

with open("ciudades.csv", "w", newline='') as archivo:
    campos = ['Ciudad', 'Poblacion']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(ciudades)

with open("ciudades.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Poblacion']) > 1000000:
            print(fila['Ciudad'])
```

Explicación: Escritura, lectura y filtrado según valor numérico.

Ejercicio 18

Enunciado: Leer `personas.csv` y guardar solo los nombres en un nuevo archivo `nombres.csv`.

Solución:

```
import csv

nombres = []

with open("personas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        nombres.append([fila['Nombre']])

with open("nombres.csv", "w", newline='') as archivo:
    escritor = csv.writer(archivo)
    escritor.writerow(['Nombre'])
    escritor.writerows(nombres)
```

Explicación: Seleccionar una columna y guardar en un nuevo CSV.

Ejercicio 19

Enunciado: Leer `notas.csv` y mostrar solo los estudiantes con nota impar.

Solución:

```
import csv

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Nota']) % 2 == 1:
            print(fila['Estudiante'])
```

Explicación: Operación módulo para detectar números impares.

Ejercicio 20

Enunciado: Crear un CSV `ventas.csv` con productos y cantidad vendida, y calcular promedio de ventas.

Solución:

```
import csv

ventas = [
    {'Producto': 'A', 'Cantidad': 10},
    {'Producto': 'B', 'Cantidad': 20},
    {'Producto': 'C', 'Cantidad': 30}
]

with open("ventas.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Cantidad']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(ventas)

total = 0
contador = 0
with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        total += int(fila['Cantidad'])
        contador += 1

promedio = total / contador
print(promedio)
```

Explicación: Sumar valores y dividir entre cantidad de filas para obtener promedio.

8.2 Ejercicios de Python – Tratamiento de ficheros CSV (Nivel medio)

Ejercicio 1

Enunciado: Leer un CSV `empleados.csv` y mostrar solo los empleados cuyo salario sea mayor al promedio.

Solución:

```
import csv

salarios = []

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        salarios.append(float(fila['Salario']))

promedio = sum(salarios)/len(salarios)

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if float(fila['Salario']) > promedio:
            print(fila['Nombre'])
```

Explicación: Primero calculamos el promedio y luego filtramos empleados que superan ese valor.

Ejercicio 2

Enunciado: Combinar dos CSV (`ventas_enero.csv` y `ventas_febrero.csv`) en uno solo `ventas_totales.csv`.

Solución:

```
import csv

archivos = ["ventas_enero.csv", "ventas_febrero.csv"]
datos = []

for nombre in archivos:
    with open(nombre, newline='') as archivo:
        lector = csv.DictReader(archivo)
        datos.extend(list(lector))

with open("ventas_totales.csv", "w", newline='') as archivo:
    campos = datos[0].keys()
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(datos)
```

Explicación: Leemos ambos archivos, combinamos filas en memoria y escribimos un CSV final.

Ejercicio 3

Enunciado: Leer productos.csv y ordenar los productos por precio descendente.

Solución:

```
import csv

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    productos = sorted(lector, key=lambda x: float(x['Precio']), reverse=True)

for p in productos:
    print(p)
```

Explicación: sorted con key permite ordenar por columna numérica.

Ejercicio 4

Enunciado: Filtrar notas.csv para estudiantes que aprobaron y guardar en aprobados.csv.

Solución:

```
import csv

aprobados = []

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Nota']) >= 7:
            aprobados.append(fila)

with open("aprobados.csv", "w", newline='') as archivo:
    campos = ['Estudiante', 'Nota']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(aprobados)
```

Explicación: Filtrado y creación de un nuevo archivo con los aprobados.

Ejercicio 5

Enunciado: Calcular la suma y promedio de la columna `Cantidad` en `inventario.csv`.

Solución:

```
import csv

cantidades = []

with open("inventario.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        cantidades.append(int(fila['Cantidad']))

suma = sum(cantidades)
promedio = suma / len(cantidades)

print(f"Suma: {suma}, Promedio: {promedio}")
```

Explicación: Se acumulan valores en lista y luego se calculan suma y promedio.

Ejercicio 6

Enunciado: Leer `empleados.csv` y mostrar solo los que trabajan en "Ventas".

Solución:

```
import csv

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if fila['Departamento'] == 'Ventas':
            print(fila['Nombre'])
```

Explicación: Filtrado de filas según valor de columna.

Ejercicio 7

Enunciado: Crear un CSV `edad_mayor.csv` con los empleados mayores de 30 años.

Solución:

```
import csv

mayores = []

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Edad']) > 30:
            mayores.append(fila)

with open("edad_mayor.csv", "w", newline='') as archivo:
    campos = ['Nombre', 'Edad', 'Departamento', 'Salario']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(mayores)
```

Explicación: Filtrado por condición y guardado en nuevo CSV.

Ejercicio 8

Enunciado: Contar cuántos productos hay de cada categoría en `productos.csv`.

Solución:

```
import csv
from collections import defaultdict

categorias = defaultdict(int)

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        categorias[fila['Categoria']] += 1

print(dict(categorias))
```

Explicación: Usamos `defaultdict` para contar ocurrencias por categoría.

Ejercicio 9

Enunciado: Leer `ventas.csv` y mostrar el producto con mayor cantidad vendida.

Solución:

```
import csv

max_cantidad = 0
producto_max = ""

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        cantidad = int(fila['Cantidad'])
        if cantidad > max_cantidad:
            max_cantidad = cantidad
            producto_max = fila['Producto']

print(producto_max, max_cantidad)
```

Explicación: Recorrer filas y actualizar máximo en cada iteración.

Ejercicio 10

Enunciado: Leer `notas.csv` y crear un diccionario {Estudiante: Nota}.

Solución:

```
import csv

dic = {}

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        dic[fila['Estudiante']] = int(fila['Nota'])

print(dic)
```

Explicación: Conversión de filas a diccionario para acceso rápido por estudiante.

Ejercicio 11

Enunciado: Leer `empleados.csv` y calcular el salario promedio por departamento.

Solución:

```
import csv
from collections import defaultdict

suma_salarios = defaultdict(float)
contador = defaultdict(int)

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        dept = fila['Departamento']
        suma_salarios[dept] += float(fila['Salario'])
        contador[dept] += 1

promedio_dept = {d: suma_salarios[d]/contador[d] for d in suma_salarios}
print(promedio_dept)
```

Explicación: Agrupamos por departamento y calculamos promedio.

Ejercicio 12

Enunciado: Filtrar `ventas.csv` para productos con cantidad entre 10 y 50.

Solución:

```
import csv

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        cantidad = int(fila['Cantidad'])
        if 10 <= cantidad <= 50:
            print(fila['Producto'], cantidad)
```

Explicación: Uso de operadores de rango en condición `if`.

Ejercicio 13

Enunciado: Crear un CSV `productos_rebajados.csv` con 10% menos en precio.

Solución:

```
import csv

productos_mod = []

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        fila['Precio'] = round(float(fila['Precio'])*0.9, 2)
        productos_mod.append(fila)

with open("productos_rebajados.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Precio', 'Categoria']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(productos_mod)
```

Explicación: Modificación de datos y guardado en nuevo CSV.

Ejercicio 14

Enunciado: Ordenar `ventas.csv` por producto alfabéticamente.

Solución:

```
import csv

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    datos = sorted(lector, key=lambda x: x['Producto'])

for fila in datos:
    print(fila)
```

Explicación: `sorted` con `key` sobre columna de texto.

Ejercicio 15

Enunciado: Contar cuántos estudiantes aprobaron y cuántos reprobaron en `notas.csv`.

Solución:

```
import csv

aprobados = 0
reprobados = 0

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Nota']) >= 7:
            aprobados += 1
        else:
            reprobados += 1

print(f"Aprobados: {aprobados}, Reprobados: {reprobados}")
```

Explicación: Uso de contadores separados según condición.

Ejercicio 16

Enunciado: Crear CSV `ventas_filtradas.csv` con ventas solo de productos de categoría "Electrónica".

Solución:

```
import csv

filtradas = []

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if fila['Categoria'] == "Electrónica":
            filtradas.append(fila)

with open("ventas_filtradas.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Precio', 'Categoria']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(filtradas)
```

Explicación: Filtrado avanzado por valor de columna.

Ejercicio 17

Enunciado: Leer `inventario.csv` y mostrar productos cuya cantidad es múltiplo de 5.

Solución:

```
import csv

with open("inventario.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Cantidad']) % 5 == 0:
            print(fila['Producto'], fila['Cantidad'])
```

Explicación: Uso de módulo % para identificar múltiplos.

Ejercicio 18

Enunciado: Crear CSV `top_3_ventas.csv` con los 3 productos más vendidos de `ventas.csv`.

Solución:

```
import csv

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    datos = sorted(lector, key=lambda x: int(x['Cantidad']), reverse=True)

top3 = datos[:3]

with open("top_3_ventas.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Cantidad']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(top3)
```

Explicación: Ordenamiento descendente y selección de primeros 3 elementos.

Ejercicio 19

Enunciado: Crear un CSV `incremento_salarios.csv` aumentando 15% a empleados del departamento "Ventas".

Solución:

```
import csv

modificados = []

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if fila['Departamento'] == 'Ventas':
            fila['Salario'] = round(float(fila['Salario'])*1.15,2)
            modificados.append(fila)

with open("incremento_salarios.csv", "w", newline='') as archivo:
    campos = ['Nombre', 'Edad', 'Departamento', 'Salario']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(modificados)
```

Explicación: Condición por columna y modificación de valores numéricos.

Ejercicio 20

Enunciado: Leer `notas.csv` y crear un diccionario con listas: `{ 'Aprobados': [...], 'Reprobados': [...] }`.

Solución:

```
import csv

resultado = {'Aprobados': [], 'Reprobados': []}

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        if int(fila['Nota']) >= 7:
            resultado['Aprobados'].append(fila['Estudiante'])
        else:
            resultado['Reprobados'].append(fila['Estudiante'])

print(resultado)
```

Explicación: Agrupamos estudiantes en listas según condición.

8.3 Ejercicios de Python – Tratamiento de ficheros CSV (Nivel avanzado)

Ejercicio 1

Enunciado: Leer un CSV `ventas.csv` y calcular la venta total por cada producto.

Solución:

```
import csv
from collections import defaultdict

totales = defaultdict(float)

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        totales[fila['Producto']] += float(fila['Cantidad']) *
float(fila['Precio'])

for producto, total in totales.items():
    print(producto, total)
```

Explicación: Se multiplican cantidad y precio por fila y se acumula por producto usando `defaultdict`.

Ejercicio 2

Enunciado: Combinar tres archivos CSV (`ventas_enero.csv`, `ventas_febrero.csv`, `ventas_marzo.csv`) en uno solo, eliminando duplicados por `ID_venta`.

Solución:

```
import csv

archivos = ["ventas_enero.csv", "ventas_febrero.csv", "ventas_marzo.csv"]
seen = set()
datos = []

for nombre in archivos:
    with open(nombre, newline='') as archivo:
        lector = csv.DictReader(archivo)
        for fila in lector:
            if fila['ID_venta'] not in seen:
                datos.append(fila)
                seen.add(fila['ID_venta'])

with open("ventas_trimestral.csv", "w", newline='') as archivo:
    campos = datos[0].keys()
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(datos)
```

Explicación: Se usa un `set` para evitar duplicados y se combinan múltiples archivos.

Ejercicio 3

Enunciado: Leer `notas.csv` y calcular el promedio por cada letra de inicial de estudiante.

Solución:

```
import csv
from collections import defaultdict

suma = defaultdict(float)
contador = defaultdict(int)

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        inicial = fila['Estudiante'][0].upper()
        suma[inicial] += float(fila['Nota'])
        contador[inicial] += 1

promedio = {k: suma[k]/contador[k] for k in suma}
print(promedio)
```

Explicación: Agrupamos por inicial y calculamos promedio.

Ejercicio 4

Enunciado: Leer `empleados.csv` y mostrar el empleado con mayor salario por departamento.

Solución:

```
import csv
from collections import defaultdict

max_salario = defaultdict(lambda: 0)
empleado_max = {}

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        dept = fila['Departamento']
        salario = float(fila['Salario'])
        if salario > max_salario[dept]:
            max_salario[dept] = salario
            empleado_max[dept] = fila['Nombre']

print(empleado_max)
```

Explicación: Se usa `defaultdict` para llevar el máximo por grupo.

Ejercicio 5

Enunciado: Crear un CSV `ventas_por_mes.csv` que contenga total de ventas por producto y mes.

Solución:

```
import csv
from collections import defaultdict

totales = defaultdict(lambda: defaultdict(float))

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        producto = fila['Producto']
        mes = fila['Mes']
        totales[producto][mes] += float(fila['Cantidad']) *
float(fila['Precio'])

with open("ventas_por_mes.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Mes', 'Total']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    for producto in totales:
        for mes in totales[producto]:
            escritor.writerow({'Producto': producto, 'Mes': mes, 'Total':
totales[producto][mes]})
```

Explicación: Agrupamiento doble por producto y mes usando `defaultdict` anidado.

Ejercicio 6

Enunciado: Leer `productos.csv` y mostrar los 5 productos más caros.

Solución:

```
import csv

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    productos = sorted(lector, key=lambda x: float(x['Precio']), reverse=True)

for fila in productos[:5]:
    print(fila)
```

Explicación: Ordenamos por precio descendente y seleccionamos los primeros 5.

Ejercicio 7

Enunciado: Leer `ventas.csv` y calcular el promedio de ventas por producto.

Solución:

```
import csv
from collections import defaultdict

suma = defaultdict(float)
contador = defaultdict(int)

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        producto = fila['Producto']
        suma[producto] += float(fila['Cantidad'])
        contador[producto] += 1

promedio = {k: suma[k]/contador[k] for k in suma}
print(promedio)
```

Explicación: Calculamos promedio agrupando por producto.

Ejercicio 8

Enunciado: Filtrar `notas.csv` para estudiantes con nota superior al promedio general.

Solución:

```
import csv

notas = []

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        notas.append({'Estudiante': fila['Estudiante'], 'Nota':
float(fila['Nota'])})

promedio = sum(n['Nota'] for n in notas)/len(notas)

for n in notas:
    if n['Nota'] > promedio:
        print(n['Estudiante'], n['Nota'])
```

Explicación: Primero calculamos el promedio general, luego filtramos.

Ejercicio 9

Enunciado: Crear un CSV `ranking_notas.csv` con estudiantes ordenados por nota descendente.

Solución:

```
import csv

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    notas = sorted(lector, key=lambda x: float(x['Nota']), reverse=True)

with open("ranking_notas.csv", "w", newline='') as archivo:
    campos = ['Estudiante', 'Nota']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(notas)
```

Explicación: Ordenamiento avanzado y escritura de un nuevo CSV.

Ejercicio 10

Enunciado: Leer `empleados.csv` y crear CSV `edad_departamento.csv` con edad promedio por departamento.

Solución:

```
import csv
from collections import defaultdict

suma = defaultdict(int)
contador = defaultdict(int)

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        dept = fila['Departamento']
        suma[dept] += int(fila['Edad'])
        contador[dept] += 1

promedios = {d: suma[d]/contador[d] for d in suma}

with open("edad_departamento.csv", "w", newline='') as archivo:
    campos = ['Departamento', 'Edad_Promedio']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    for d in promedios:
        escritor.writerow({'Departamento': d, 'Edad_Promedio': promedios[d]})
```

Explicación: Agrupamiento por departamento y cálculo de promedio.

Ejercicio 11

Enunciado: Leer `ventas.csv` y mostrar productos cuya venta total supera 1000.

Solución:

```
import csv
from collections import defaultdict

totales = defaultdict(float)

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        totales[fila['Producto']] += float(fila['Cantidad']) *
float(fila['Precio'])

for producto, total in totales.items():
    if total > 1000:
        print(producto, total)
```

Explicación: Acumulamos totales por producto y filtramos por condición avanzada.

Ejercicio 12

Enunciado: Leer `productos.csv` y calcular cuántos productos hay por rango de precio: <10, 10–50, >50.

Solución:

```
import csv
from collections import defaultdict

rangos = defaultdict(int)

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        precio = float(fila['Precio'])
        if precio < 10:
            rangos['<10'] += 1
        elif precio <= 50:
            rangos['10-50'] += 1
        else:
            rangos['>50'] += 1

print(rangos)
```

Explicación: Clasificación avanzada usando rangos de valores.

Ejercicio 13

Enunciado: Leer `ventas.csv` y crear un CSV `top_vendedores.csv` con los 3 productos con mayores ventas por cantidad.

Solución:

```
import csv
from collections import defaultdict

totales = defaultdict(int)

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        totales[fila['Producto']] += int(fila['Cantidad'])

top3 = sorted(totales.items(), key=lambda x: x[1], reverse=True)[:3]

with open("top_vendedores.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Cantidad_Total']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    for producto, cantidad in top3:
        escritor.writerow({'Producto': producto, 'Cantidad_Total': cantidad})
```

Explicación: Agrupamiento, ordenamiento y selección de top 3.

Ejercicio 14

Enunciado: Crear un CSV `notas_categoria.csv` con estudiantes clasificados como “Alta”, “Media” o “Baja” según nota.

Solución:

```
import csv

clasificados = []

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        nota = float(fila['Nota'])
        if nota >= 8:
            categoria = 'Alta'
        elif nota >= 5:
            categoria = 'Media'
        else:
            categoria = 'Baja'
        fila['Categoria'] = categoria
        clasificados.append(fila)

with open("notas_categoria.csv", "w", newline='') as archivo:
    campos = ['Estudiante', 'Nota', 'Categoria']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(clasificados)
```

Explicación: Clasificación avanzada y escritura de nuevo CSV.

Ejercicio 15

Enunciado: Leer `empleados.csv` y mostrar la desviación estándar del salario.

Solución:

```
import csv
import math

salarios = []

with open("empleados.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        salarios.append(float(fila['Salario']))

promedio = sum(salarios)/len(salarios)
varianza = sum((x-promedio)**2 for x in salarios)/len(salarios)
desviacion = math.sqrt(varianza)
print(desviacion)
```

Explicación: Cálculo de estadística avanzada (desviación estándar).

Ejercicio 16

Enunciado: Crear CSV `ventas_filtradas.csv` solo con productos que tengan ventas mayores al promedio.

Solución:

```
import csv

ventas = []

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        ventas.append({'Producto': fila['Producto'], 'Cantidad':
float(fila['Cantidad'])})

promedio = sum(v['Cantidad'] for v in ventas)/len(ventas)
filtradas = [v for v in ventas if v['Cantidad'] > promedio]

with open("ventas_filtradas.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Cantidad']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(filtradas)
```

Explicación: Filtrado avanzado usando comprensión de listas.

Ejercicio 17

Enunciado: Crear CSV `promedio_notas.csv` con promedio de notas por inicial de estudiante.

Solución:

```
import csv
from collections import defaultdict

suma = defaultdict(float)
contador = defaultdict(int)

with open("notas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        inicial = fila['Estudiante'][0].upper()
        suma[inicial] += float(fila['Nota'])
        contador[inicial] += 1

promedios = [{'Inicial': k, 'Promedio': suma[k]/contador[k]} for k in suma]

with open("promedio_notas.csv", "w", newline='') as archivo:
    campos = ['Inicial', 'Promedio']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(promedios)
```

Explicación: Agrupamiento por inicial y escritura en CSV.

Ejercicio 18

Enunciado: Leer `productos.csv` y crear CSV `productos_mayor_precio.csv` con productos que superen el precio promedio.

Solución:

```
import csv

productos = []

with open("productos.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        productos.append(fila)

promedio = sum(float(p['Precio']) for p in productos)/len(productos)
mayores = [p for p in productos if float(p['Precio']) > promedio]

with open("productos_mayor_precio.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Precio', 'Categoria']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor.writeheader()
    escritor.writerows(mayores)
```

Explicación: Filtrado avanzado por promedio.

Ejercicio 19

Enunciado: Crear CSV `ventas_acumuladas.csv` con total acumulado de ventas por producto.

Solución:

```
import csv
from collections import defaultdict

acumulado = defaultdict(float)

with open("ventas.csv", newline='') as archivo:
    lector = csv.DictReader(archivo)
    for fila in lector:
        acumulado[fila['Producto']] += float(fila['Cantidad'])

with open("ventas_acumuladas.csv", "w", newline='') as archivo:
    campos = ['Producto', 'Cantidad_Total']
    escritor = csv.DictWriter(archivo, fieldnames=campos)
    escritor
    .writeheader()
    for producto, total in acumulado.items():
        escritor.writerow({'Producto': producto, 'Cantidad_Total': total})

**Explicación:** Cálculo acumulado y escritura de resultados.
```

****Ejercicio 20****

****Enunciado:**** Usando `pandas`, leer `ventas.csv` y mostrar los 5 productos con mayor venta total.

****Solución:****

```
` ``python
import pandas as pd

df = pd.read_csv("ventas.csv")
df['Total'] = df['Cantidad'] * df['Precio']
top5 = df.groupby('Producto')
['Total'].sum().sort_values(ascending=False).head(5)
```

```
print(top5)
```

Explicación: Uso de pandas para agregación avanzada y ordenamiento.

Ejercicio 20

Enunciado: Usando pandas, leer `ventas.csv` y mostrar los 5 productos con mayor venta total.

Solución:

```
import pandas as pd

df = pd.read_csv("ventas.csv")
df['Total'] = df['Cantidad'] * df['Precio']
top5 = df.groupby('Producto')
['Total'].sum().sort_values(ascending=False).head(5)
print(top5)
```

Explicación: Uso de pandas para agregación avanzada y ordenamiento.

Epílogo

Llegar hasta aquí no ha sido un camino corto, y si has recorrido cada página de este libro, significa que has dado un paso importante en tu aprendizaje de la programación con Python. Lo que comenzó como un primer acercamiento a los conceptos básicos ha ido creciendo poco a poco hasta adentrarse en temas más avanzados, mostrándote no solo cómo escribir código, sino cómo pensar como un programador.

El conocimiento que has adquirido es una herramienta poderosa. Python no es únicamente un lenguaje de programación, es una puerta hacia múltiples oportunidades: el desarrollo de aplicaciones, la automatización de tareas, el análisis de datos, la inteligencia artificial, el aprendizaje automático, la creación de videojuegos o la experimentación artística. Las posibilidades son tan amplias como tu imaginación y tu deseo de seguir aprendiendo.

Este libro ha sido escrito con la convicción de que programar no debe ser visto como una tarea inaccesible ni reservada para unos pocos, sino como una habilidad abierta a todos, que combina lógica, creatividad y pasión por crear. Si este viaje te ha inspirado, entonces el objetivo está cumplido.

Quiero recordarte que este es solo el comienzo. La programación es un camino de aprendizaje continuo, y cada proyecto que emprendas será una oportunidad para crecer. Mantén la curiosidad viva, no tengas miedo a equivocarte y, sobre todo, disfruta del proceso.

Y una vez más, deseo dedicar unas últimas palabras de gratitud a mi mujer, Marisa. Su apoyo ha sido la base sobre la que pude construir este trabajo. A ella le debo no solo la realización de este libro, sino también la motivación constante para seguir adelante.