Manual python clase por clase

## Clase 1: Introducción a Python

### Objetivos de la clase:

- Entender los conceptos básicos de Python.

- Aprender a instalar y configurar Python.

- Familiarizarse con el entorno de desarrollo.

- Escribir y ejecutar tu primer programa en Python.

### 1. Instalación y Configuración

#### Instalación de Python:

1. \*\*Descargar Python:\*\*

- Ve a la [página oficial de descargas de Python](https://www.python.org/downloads/).

- Descarga la última versión de Python para tu sistema operativo.

2. \*\*Instalar Python:\*\*

- Abre el archivo descargado y sigue las instrucciones de instalación.

- Asegúrate de marcar la opción "Add Python to PATH" durante la instalación.

#### Instalación de un Entorno de Desarrollo:

- \*\*Visual Studio Code (VS Code):\*\*

- Descarga e instala VS Code desde [su página oficial](https://code.visualstudio.com/).

- Instala la extensión de Python en VS Code.

### 2. Escribiendo tu Primer Programa

#### Hola Mundo:

1. Abre VS Code y crea un nuevo archivo con extensión `.py` (por ejemplo, `hola\_mundo.py`).

2. Escribe el siguiente código en el archivo:

```python

print("Hola, Mundo!")

```

3. Guarda el archivo y ejecútalo:

- Abre la terminal en VS Code (puedes usar el atajo `Ctrl + ~`).

- Navega hasta el directorio donde guardaste el archivo.

- Ejecuta el archivo escribiendo `python hola\_mundo.py` en la terminal.

### 3. Conceptos Básicos de Python

#### Variables y Tipos de Datos:

```python

# Variables

nombre = "Juan"

edad = 25

altura = 1.75

es\_estudiante = True

# Imprimir valores

print(nombre)

print(edad)

print(altura)

print(es\_estudiante)

```

#### Operadores:

```python

# Operadores aritméticos

suma = 10 + 5

resta = 10 - 5

multiplicacion = 10 \* 5

division = 10 / 5

# Operadores de comparación

es\_igual = 10 == 5

es\_diferente = 10 != 5

es\_mayor = 10 > 5

es\_menor = 10 < 5

```

#### Estructuras de Control:

```python

# Condicionales

if edad > 18:

print("Eres mayor de edad.")

else:

print("Eres menor de edad.")

# Bucles

# Bucle for

for i in range(5):

print(i)

# Bucle while

contador = 0

while contador < 5:

print(contador)

contador += 1

```

#### Funciones:

```python

# Definir una función

def saludar(nombre):

return f"Hola, {nombre}!"

# Llamar a la función

mensaje = saludar("Juan")

print(mensaje)

```

### Tarea

#### Parte 1:

1. Instala Python y VS Code en tu computadora.

2. Crea un archivo llamado `primer\_programa.py`.

3. Escribe un programa que:

- Solicite al usuario su nombre, edad y altura.

- Imprima un mensaje de saludo que incluya su nombre.

- Calcule cuántos años tendrá el usuario en 10 años y lo imprima.

- Calcule la altura del usuario en centímetros y la imprima.

#### Parte 2:

1. Crea un archivo llamado `calculadora.py`.

2. Escribe un programa que:

- Solicite al usuario dos números.

- Imprima la suma, resta, multiplicación y división de esos números.

### Ejemplo de solución para `primer\_programa.py`:

```python

# Solicitar información al usuario

nombre = input("¿Cuál es tu nombre? ")

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

altura = float(input("¿Cuál es tu altura en metros? "))

# Saludo

print(f"Hola, {nombre}!")

# Calcular edad en 10 años

edad\_futura = edad + 10

print(f"En 10 años tendrás {edad\_futura} años.")

# Calcular altura en centímetros

altura\_cm = altura \* 100

print(f"Tu altura en centímetros es {altura\_cm} cm.")

```

### Ejemplo de solución para `calculadora.py`:

```python

# Solicitar números al usuario

num1 = float(input("Ingresa el primer número: "))

num2 = float(input("Ingresa el segundo número: "))

# Realizar operaciones

suma = num1 + num2

resta = num1 - num2

multiplicacion = num1 \* num2

division = num1 / num2

# Imprimir resultados

print(f"Suma: {suma}")

print(f"Resta: {resta}")

print(f"Multiplicación: {multiplicacion}")

print(f"División: {division}")

Clase 2: Estructuras de Datos en Python

### Objetivos de la clase:

- Aprender sobre las principales estructuras de datos en Python.

- Entender cómo manipular listas, tuplas, conjuntos y diccionarios.

- Escribir programas prácticos utilizando estas estructuras de datos.

### 1. Listas

Las listas son colecciones ordenadas y mutables que permiten almacenar múltiples elementos.

#### Crear y Acceder a Listas:

```python

# Crear una lista

frutas = ["manzana", "banana", "cereza"]

# Acceder a elementos

print(frutas[0]) # manzana

print(frutas[1]) # banana

# Modificar elementos

frutas[0] = "kiwi"

print(frutas) # ['kiwi', 'banana', 'cereza']

```

#### Métodos de Listas:

```python

# Añadir elementos

frutas.append("naranja")

print(frutas) # ['kiwi', 'banana', 'cereza', 'naranja']

# Eliminar elementos

frutas.remove("banana")

print(frutas) # ['kiwi', 'cereza', 'naranja']

# Longitud de la lista

print(len(frutas)) # 3

# Recorrer una lista

for fruta in frutas:

print(fruta)

```

### 2. Tuplas

Las tuplas son colecciones ordenadas e inmutables.

#### Crear y Acceder a Tuplas:

```python

# Crear una tupla

punto = (10, 20)

# Acceder a elementos

print(punto[0]) # 10

print(punto[1]) # 20

# No se pueden modificar los elementos de una tupla

# punto[0] = 30 # Esto causaría un error

```

### 3. Conjuntos

Los conjuntos son colecciones desordenadas de elementos únicos.

#### Crear y Operar con Conjuntos:

```python

# Crear un conjunto

numeros = {1, 2, 3, 4, 5}

# Añadir y eliminar elementos

numeros.add(6)

numeros.remove(2)

print(numeros) # {1, 3, 4, 5, 6}

# Operaciones de conjuntos

pares = {2, 4, 6, 8}

impares = {1, 3, 5, 7}

union = pares.union(impares)

interseccion = pares.intersection(impares)

diferencia = pares.difference(impares)

print(union) # {1, 2, 3, 4, 5, 6, 7, 8}

print(interseccion) # set()

print(diferencia) # {8, 2, 4, 6}

```

### 4. Diccionarios

Los diccionarios son colecciones desordenadas de pares clave-valor.

#### Crear y Acceder a Diccionarios:

```python

# Crear un diccionario

alumno = {

"nombre": "Juan",

"edad": 21,

"carrera": "Ingeniería"

}

# Acceder a valores

print(alumno["nombre"]) # Juan

print(alumno["edad"]) # 21

# Modificar valores

alumno["edad"] = 22

print(alumno) # {'nombre': 'Juan', 'edad': 22, 'carrera': 'Ingeniería'}

# Añadir nuevos pares clave-valor

alumno["universidad"] = "UNAM"

print(alumno) # {'nombre': 'Juan', 'edad': 22, 'carrera': 'Ingeniería', 'universidad': 'UNAM'}

```

#### Recorrer un Diccionario:

```python

# Recorrer claves

for clave in alumno:

print(clave)

# Recorrer valores

for valor in alumno.values():

print(valor)

# Recorrer claves y valores

for clave, valor in alumno.items():

print(f"{clave}: {valor}")

```

### Tarea

#### Parte 1: Manipulación de Listas y Tuplas

1. Crea un archivo llamado `lista\_tupla.py`.

2. Escribe un programa que:

- Cree una lista con los nombres de tus cinco películas favoritas.

- Imprima la lista completa.

- Reemplace la segunda película con una nueva y vuelva a imprimir la lista.

- Cree una tupla con los nombres de tres ciudades que te gustaría visitar.

- Imprima cada ciudad en la tupla usando un bucle `for`.

#### Parte 2: Uso de Conjuntos y Diccionarios

1. Crea un archivo llamado `conjunto\_diccionario.py`.

2. Escribe un programa que:

- Cree un conjunto con los números del 1 al 10.

- Añada los números 11 y 12 al conjunto.

- Imprima el conjunto resultante.

- Cree un diccionario que almacene información sobre un libro (título, autor, año de publicación).

- Imprima cada clave y valor del diccionario en un formato legible.

### Ejemplo de solución para `lista\_tupla.py`:

```python

# Lista de películas favoritas

peliculas = ["Inception", "Interstellar", "The Dark Knight", "Memento", "Dunkirk"]

print("Lista de películas:", peliculas)

# Reemplazar la segunda película

peliculas[1] = "Tenet"

print("Lista modificada:", peliculas)

# Tupla de ciudades a visitar

ciudades = ("Tokyo", "Paris", "New York")

print("Ciudades a visitar:")

for ciudad in ciudades:

print(ciudad)

```

### Ejemplo de solución para `conjunto\_diccionario.py`:

```python

# Conjunto de números

numeros = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

numeros.add(11)

numeros.add(12)

print("Conjunto de números:", numeros)

# Diccionario de un libro

libro = {

"titulo": "Cien Años de Soledad",

"autor": "Gabriel García Márquez",

"año": 1967

}

print("Información del libro:")

for clave, valor in libro.items():

print(f"{clave}: {valor}")

**Clase 3: Funciones y Módulos en Python**

### Objetivos de la clase:

- Entender cómo definir y usar funciones en Python.

- Aprender sobre argumentos y valores de retorno en funciones.

- Conocer cómo organizar el código utilizando módulos.

- Aprender a importar y utilizar funciones desde otros archivos.

### 1. Definiendo y Usando Funciones

#### Crear una Función:

Las funciones en Python se definen utilizando la palabra clave `def`.

```python

# Definir una función sin parámetros

def saludar():

print("¡Hola, Mundo!")

# Llamar a la función

saludar()

```

#### Funciones con Parámetros:

Las funciones pueden aceptar parámetros para trabajar con datos que se pasan a la función.

```python

# Definir una función con parámetros

def saludar(nombre):

print(f"¡Hola, {nombre}!")

# Llamar a la función con un argumento

saludar("Juan")

```

#### Funciones con Valores de Retorno:

Las funciones pueden devolver valores utilizando la palabra clave `return`.

```python

# Definir una función que devuelve un valor

def suma(a, b):

return a + b

# Llamar a la función y almacenar el resultado

resultado = suma(5, 3)

print(f"La suma es: {resultado}")

```

### 2. Funciones con Argumentos por Defecto y Variables

#### Argumentos por Defecto:

Puedes definir valores predeterminados para los argumentos de una función.

```python

# Definir una función con un argumento por defecto

def presentar(nombre, edad=30):

print(f"Nombre: {nombre}, Edad: {edad}")

# Llamar a la función con y sin el argumento por defecto

presentar("Ana")

presentar("Luis", 25)

```

#### Argumentos Variables:

Puedes usar `\*args` y `\*\*kwargs` para aceptar un número variable de argumentos.

```python

# Usar \*args para argumentos variables

def sumar\_todos(\*args):

return sum(args)

print(sumar\_todos(1, 2, 3)) # 6

print(sumar\_todos(4, 5, 6, 7, 8)) # 30

# Usar \*\*kwargs para argumentos con nombre variables

def mostrar\_info(\*\*kwargs):

for clave, valor in kwargs.items():

print(f"{clave}: {valor}")

mostrar\_info(nombre="Carlos", edad=28, ciudad="Madrid")

```

### 3. Módulos en Python

#### Qué es un Módulo:

Un módulo es un archivo Python que contiene definiciones de funciones, clases y variables que puedes reutilizar en otros programas.

#### Creando un Módulo:

1. Crea un archivo llamado `mi\_modulo.py` con el siguiente contenido:

```python

# mi\_modulo.py

def sumar(a, b):

return a + b

def restar(a, b):

return a - b

PI = 3.14159

```

#### Importando un Módulo:

Puedes importar un módulo en tu programa usando la palabra clave `import`.

```python

# Importar el módulo

import mi\_modulo

# Usar funciones del módulo

resultado\_suma = mi\_modulo.sumar(5, 3)

resultado\_resta = mi\_modulo.restar(5, 3)

print(f"La suma es: {resultado\_suma}")

print(f"La resta es: {resultado\_resta}")

# Acceder a una variable del módulo

print(f"El valor de PI es: {mi\_modulo.PI}")

```

#### Importando Funciones Específicas:

También puedes importar funciones específicas desde un módulo.

```python

# Importar funciones específicas

from mi\_modulo import sumar, restar

# Usar las funciones importadas

print(sumar(10, 5))

print(restar(10, 5))

```

### Tarea

#### Parte 1: Funciones en Python

1. Crea un archivo llamado `funciones.py`.

2. Escribe un programa que:

- Defina una función llamada `convertir\_celsius\_a\_fahrenheit` que convierta grados Celsius a Fahrenheit.

- Defina una función llamada `es\_par` que determine si un número es par.

- Solicite al usuario un número en grados Celsius y lo convierta a Fahrenheit usando la función.

- Solicite al usuario un número y verifique si es par usando la función `es\_par`.

#### Parte 2: Módulos en Python

1. Crea un archivo llamado `matematica.py`.

2. Define un módulo que contenga las siguientes funciones:

- `multiplicar(a, b)`: Multiplica dos números.

- `dividir(a, b)`: Divide dos números.

3. Crea otro archivo llamado `main.py`.

4. En `main.py`, importa las funciones del módulo `matematica.py` y usa las funciones `multiplicar` y `dividir` con diferentes valores. Imprime los resultados.

### Ejemplo de solución para `funciones.py`:

```python

# Definir la función para convertir Celsius a Fahrenheit

def convertir\_celsius\_a\_fahrenheit(celsius):

return (celsius \* 9/5) + 32

# Definir la función para verificar si un número es par

def es\_par(numero):

return numero % 2 == 0

# Solicitar al usuario la temperatura en Celsius

celsius = float(input("Introduce la temperatura en grados Celsius: "))

fahrenheit = convertir\_celsius\_a\_fahrenheit(celsius)

print(f"La temperatura en Fahrenheit es: {fahrenheit}")

# Solicitar al usuario un número y verificar si es par

numero = int(input("Introduce un número: "))

if es\_par(numero):

print(f"{numero} es un número par.")

else:

print(f"{numero} es un número impar.")

```

### Ejemplo de solución para `matematica.py`:

```python

# matematica.py

def multiplicar(a, b):

return a \* b

def dividir(a, b):

if b != 0:

return a / b

else:

return "Error: División por cero."

```

### Ejemplo de solución para `main.py`:

```python

# main.py

from matematica import multiplicar, dividir

# Usar las funciones del módulo

resultado\_multiplicacion = multiplicar(4, 5)

resultado\_division = dividir(20, 4)

print(f"La multiplicación es: {resultado\_multiplicacion}")

print(f"La división es: {resultado\_division}")

**Clase 4: Estructuras de Datos en Python (Listas, Tuplas, Diccionarios y Conjuntos)**

### Objetivos de la clase:

- Entender cómo usar listas, tuplas, diccionarios y conjuntos en Python.

- Aprender a realizar operaciones comunes en estas estructuras de datos.

- Practicar la manipulación y organización de datos utilizando estas estructuras.

### 1. Listas

Las listas son colecciones ordenadas y modificables de elementos.

#### Crear una Lista:

```python

# Crear una lista

frutas = ["manzana", "banana", "cereza"]

print(frutas)

```

#### Acceso y Modificación de Elementos:

```python

# Acceder a un elemento

print(frutas[0]) # manzana

# Modificar un elemento

frutas[1] = "kiwi"

print(frutas) # ['manzana', 'kiwi', 'cereza']

```

#### Métodos Comunes de Listas:

```python

# Agregar un elemento al final

frutas.append("naranja")

print(frutas) # ['manzana', 'kiwi', 'cereza', 'naranja']

# Eliminar un elemento

frutas.remove("kiwi")

print(frutas) # ['manzana', 'cereza', 'naranja']

# Ordenar la lista

frutas.sort()

print(frutas) # ['cereza', 'manzana', 'naranja']

```

### 2. Tuplas

Las tuplas son colecciones ordenadas e inmutables de elementos.

#### Crear una Tupla:

```python

# Crear una tupla

numeros = (1, 2, 3, 4, 5)

print(numeros)

```

#### Acceso a Elementos:

```python

# Acceder a un elemento

print(numeros[2]) # 3

# Las tuplas son inmutables, no se pueden modificar elementos.

# Intentar modificar un elemento causará un error:

# numeros[2] = 10 # Esto dará un error

```

#### Desempaquetado de Tuplas:

```python

# Desempaquetar una tupla en variables

a, b, c, d, e = numeros

print(a, b, c, d, e) # 1 2 3 4 5

```

### 3. Diccionarios

Los diccionarios son colecciones desordenadas de pares clave-valor.

#### Crear un Diccionario:

```python

# Crear un diccionario

estudiante = {"nombre": "Ana", "edad": 22, "curso": "Matemáticas"}

print(estudiante)

```

#### Acceso y Modificación de Valores:

```python

# Acceder a un valor

print(estudiante["nombre"]) # Ana

# Modificar un valor

estudiante["edad"] = 23

print(estudiante) # {'nombre': 'Ana', 'edad': 23, 'curso': 'Matemáticas'}

```

#### Métodos Comunes de Diccionarios:

```python

# Agregar un nuevo par clave-valor

estudiante["nota"] = 9.5

print(estudiante) # {'nombre': 'Ana', 'edad': 23, 'curso': 'Matemáticas', 'nota': 9.5}

# Eliminar un par clave-valor

del estudiante["curso"]

print(estudiante) # {'nombre': 'Ana', 'edad': 23, 'nota': 9.5}

# Obtener todas las claves

print(estudiante.keys()) # dict\_keys(['nombre', 'edad', 'nota'])

# Obtener todos los valores

print(estudiante.values()) # dict\_values(['Ana', 23, 9.5])

```

### 4. Conjuntos

Los conjuntos son colecciones desordenadas de elementos únicos.

#### Crear un Conjunto:

```python

# Crear un conjunto

colores = {"rojo", "verde", "azul"}

print(colores)

```

#### Operaciones con Conjuntos:

```python

# Agregar un elemento

colores.add("amarillo")

print(colores) # {'rojo', 'verde', 'azul', 'amarillo'}

# Eliminar un elemento

colores.remove("rojo")

print(colores) # {'verde', 'azul', 'amarillo'}

# Operación de unión

otros\_colores = {"negro", "blanco"}

todos\_los\_colores = colores.union(otros\_colores)

print(todos\_los\_colores) # {'verde', 'amarillo', 'blanco', 'negro', 'azul'}

```

### Tarea

#### Parte 1: Listas y Tuplas

1. Crea un archivo llamado `listas\_tuplas.py`.

2. Escribe un programa que:

- Cree una lista de nombres de cinco amigos.

- Agregue un nuevo nombre a la lista y luego lo elimine.

- Cree una tupla con tres colores favoritos.

- Imprima la lista y la tupla, y muestre el primer nombre en la lista y el segundo color en la tupla.

#### Parte 2: Diccionarios y Conjuntos

1. Crea un archivo llamado `diccionarios\_conjuntos.py`.

2. Escribe un programa que:

- Cree un diccionario para almacenar información sobre un automóvil (marca, modelo, año).

- Agregue un nuevo par clave-valor para el color del automóvil.

- Cree un conjunto con tres destinos de viaje que te gustaría visitar.

- Imprima el diccionario y el conjunto, y luego realiza una operación de unión con otro conjunto de destinos.

### Ejemplo de solución para `listas\_tuplas.py`:

```python

# Listas

amigos = ["Carlos", "Ana", "Luis", "Pedro", "María"]

amigos.append("Lucía")

print(amigos) # ['Carlos', 'Ana', 'Luis', 'Pedro', 'María', 'Lucía']

amigos.remove("Lucía")

print(amigos) # ['Carlos', 'Ana', 'Luis', 'Pedro', 'María']

# Tuplas

colores = ("rojo", "verde", "azul")

print(colores) # ('rojo', 'verde', 'azul')

print(amigos[0]) # Carlos

print(colores[1]) # verde

```

### Ejemplo de solución para `diccionarios\_conjuntos.py`:

```python

# Diccionarios

automovil = {"marca": "Toyota", "modelo": "Corolla", "año": 2020}

automovil["color"] = "rojo"

print(automovil) # {'marca': 'Toyota', 'modelo': 'Corolla', 'año': 2020, 'color': 'rojo'}

# Conjuntos

destinos = {"París", "Tokio", "Nueva York"}

otros\_destinos = {"Londres", "Berlín"}

todos\_los\_destinos = destinos.union(otros\_destinos)

print(destinos) # {'París', 'Tokio', 'Nueva York'}

print(todos\_los\_destinos) # {'Londres', 'Berlín', 'París', 'Tokio', 'Nueva York'}

**Clase 5: Funciones en Python**

### Objetivos de la clase:

- Aprender a definir y utilizar funciones en Python.

- Entender los conceptos de parámetros, argumentos, valores de retorno y alcance de variables.

- Practicar la creación de funciones reutilizables para organizar mejor el código.

### 1. Definir una Función

Una función es un bloque de código que solo se ejecuta cuando es llamado. Las funciones permiten reutilizar código y hacer el código más limpio y modular.

#### Sintaxis Básica de una Función:

```python

def nombre\_funcion():

print("Hola, esta es una función")

```

#### Llamar a una Función:

```python

nombre\_funcion() # Llama a la función y ejecuta su código

```

### 2. Funciones con Parámetros

Las funciones pueden aceptar parámetros que actúan como variables dentro de la función.

#### Función con Parámetros:

```python

def saludar(nombre):

print(f"Hola, {nombre}")

```

#### Llamar a la Función con un Argumento:

```python

saludar("Carlos") # Imprime "Hola, Carlos"

```

### 3. Valores de Retorno

Una función puede devolver un valor utilizando la palabra clave `return`. Esto es útil cuando quieres que la función realice un cálculo y te devuelva un resultado.

#### Función con Valor de Retorno:

```python

def sumar(a, b):

return a + b

```

#### Llamar a la Función y Almacenar el Valor Devuelto:

```python

resultado = sumar(5, 3)

print(resultado) # Imprime 8

```

### 4. Funciones con Múltiples Parámetros y Valores por Defecto

Puedes definir funciones con varios parámetros e incluso establecer valores predeterminados.

#### Función con Múltiples Parámetros:

```python

def multiplicar(a, b=2):

return a \* b

```

#### Llamar a la Función:

```python

print(multiplicar(3, 4)) # Imprime 12

print(multiplicar(5)) # Imprime 10 (usa el valor predeterminado de b)

```

### 5. Ámbito de Variables (Scope)

Las variables definidas dentro de una función tienen un ámbito local, lo que significa que solo existen dentro de esa función.

#### Ejemplo de Ámbito:

```python

def ejemplo\_scope():

x = 10 # Variable local

print(x)

ejemplo\_scope() # Imprime 10

# print(x) # Esto dará un error porque 'x' no existe fuera de la función

```

### 6. Funciones Anidadas y Recursión

Las funciones pueden estar anidadas (definidas dentro de otras funciones), y las funciones también pueden llamarse a sí mismas (recursión).

#### Función Anidada:

```python

def funcion\_externa():

def funcion\_interna():

print("Soy la función interna")

funcion\_interna()

funcion\_externa() # Llama a la función interna desde la externa

```

#### Recursión:

```python

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n-1)

print(factorial(5)) # Imprime 120

```

### Tarea

#### Parte 1: Funciones Básicas

1. Crea un archivo llamado `funciones\_basicas.py`.

2. Escribe un programa que defina una función llamada `calcular\_area\_rectangulo(ancho, alto)` que calcule el área de un rectángulo.

- Llama a la función con diferentes valores de `ancho` y `alto` y muestra los resultados.

#### Parte 2: Funciones con Parámetros y Retorno

1. Crea un archivo llamado `funciones\_parametros.py`.

2. Escribe una función llamada `es\_par(numero)` que tome un número como parámetro y devuelva `True` si el número es par, y `False` si es impar.

- Usa la función para verificar varios números e imprime si son pares o impares.

#### Parte 3: Recursión

1. Crea un archivo llamado `funcion\_recursiva.py`.

2. Escribe una función recursiva llamada `fibonacci(n)` que devuelva el enésimo número en la secuencia de Fibonacci.

- Llama a la función con un valor de `n` e imprime el resultado.

### Ejemplo de Solución para `funciones\_basicas.py`:

```python

def calcular\_area\_rectangulo(ancho, alto):

return ancho \* alto

# Llamada a la función

area1 = calcular\_area\_rectangulo(5, 10)

print(f"El área del rectángulo es: {area1}")

area2 = calcular\_area\_rectangulo(7, 3)

print(f"El área del rectángulo es: {area2}")

```

### Ejemplo de Solución para `funciones\_parametros.py`:

```python

def es\_par(numero):

return numero % 2 == 0

# Verificar si los números son pares o impares

print(es\_par(4)) # True

print(es\_par(7)) # False

```

### Ejemplo de Solución para `funcion\_recursiva.py`:

```python

def fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n-1) + fibonacci(n-2)

# Imprimir el 6º número de Fibonacci

print(fibonacci(6)) # Imprime 8

**Clase 6: Manejo de Archivos en Python**

### Objetivos de la clase:

- Aprender a trabajar con archivos en Python.

- Practicar la lectura y escritura de archivos de texto y CSV.

- Comprender cómo manejar errores al trabajar con archivos.

### 1. Abrir y Leer Archivos

Python permite abrir y leer archivos utilizando la función `open()`. Existen varios modos de apertura: lectura (`'r'`), escritura (`'w'`), y agregar contenido (`'a'`).

#### Leer un archivo:

```python

# Abre el archivo en modo lectura ('r')

archivo = open('archivo.txt', 'r')

# Lee el contenido del archivo

contenido = archivo.read()

print(contenido)

# Cierra el archivo

archivo.close()

```

#### Leer línea por línea:

```python

archivo = open('archivo.txt', 'r')

# Leer línea por línea

for linea in archivo:

print(linea.strip()) # Elimina saltos de línea al final de cada línea

archivo.close()

```

### 2. Escribir en Archivos

Puedes escribir en archivos usando el modo `'w'` (escribir) o `'a'` (agregar).

#### Escribir en un archivo:

```python

archivo = open('archivo\_nuevo.txt', 'w')

# Escribir una línea en el archivo

archivo.write("Esta es una nueva línea de texto.\n")

# Cerrar el archivo

archivo.close()

```

#### Agregar contenido a un archivo existente:

```python

archivo = open('archivo\_nuevo.txt', 'a')

# Agregar otra línea al archivo

archivo.write("Esta es una línea adicional.\n")

archivo.close()

```

### 3. Usar `with` para Manejar Archivos

Es recomendable utilizar la instrucción `with` al manejar archivos, ya que garantiza que el archivo se cierre automáticamente después de terminar de usarlo, incluso si ocurre un error.

#### Ejemplo con `with`:

```python

with open('archivo.txt', 'r') as archivo:

contenido = archivo.read()

print(contenido)

# No es necesario llamar a archivo.close(), se cierra automáticamente

```

### 4. Manejo de Errores al Trabajar con Archivos

Cuando trabajamos con archivos, es posible que ocurran errores (como que el archivo no exista). Para manejar estos errores, podemos utilizar bloques `try` y `except`.

#### Ejemplo de manejo de errores:

```python

try:

with open('archivo\_que\_no\_existe.txt', 'r') as archivo:

contenido = archivo.read()

print(contenido)

except FileNotFoundError:

print("El archivo no fue encontrado.")

```

### 5. Trabajar con Archivos CSV

Los archivos CSV (Comma-Separated Values) son comunes para almacenar datos tabulares. Python tiene un módulo integrado llamado `csv` para trabajar con ellos.

#### Leer un archivo CSV:

```python

import csv

with open('datos.csv', 'r') as archivo\_csv:

lector\_csv = csv.reader(archivo\_csv)

for fila in lector\_csv:

print(fila)

```

#### Escribir en un archivo CSV:

```python

import csv

with open('datos\_nuevos.csv', 'w', newline='') as archivo\_csv:

escritor\_csv = csv.writer(archivo\_csv)

# Escribir la cabecera

escritor\_csv.writerow(['Nombre', 'Edad', 'Ciudad'])

# Escribir datos

escritor\_csv.writerow(['Carlos', '30', 'Bogotá'])

escritor\_csv.writerow(['María', '25', 'Medellín'])

```

### 6. Lectura y Escritura de Archivos de Texto y CSV en una Solución

#### Combinar lectura y escritura:

```python

import csv

# Leer un archivo CSV y copiarlo a otro archivo

with open('datos.csv', 'r') as archivo\_entrada:

lector\_csv = csv.reader(archivo\_entrada)

with open('copia\_datos.csv', 'w', newline='') as archivo\_salida:

escritor\_csv = csv.writer(archivo\_salida)

for fila in lector\_csv:

escritor\_csv.writerow(fila)

```

### Tarea

#### Parte 1: Leer y Escribir Archivos de Texto

1. Crea un archivo llamado `manejo\_archivos.py`.

2. Escribe un programa que abra un archivo llamado `entrada.txt`, lea su contenido y lo muestre en la consola.

- Si el archivo no existe, imprime un mensaje adecuado.

3. Luego, escribe un nuevo archivo llamado `salida.txt` que contenga el mismo contenido, pero en mayúsculas.

#### Parte 2: Trabajar con Archivos CSV

1. Crea un archivo llamado `manejo\_csv.py`.

2. Escribe un programa que lea un archivo CSV llamado `estudiantes.csv`, que contiene los datos de estudiantes con las columnas: `Nombre`, `Edad`, `Nota`.

3. Luego, escribe un nuevo archivo CSV llamado `estudiantes\_aprobados.csv` que contenga solo los estudiantes cuya nota sea mayor o igual a 3.0.

- Ejemplo de contenido de `estudiantes.csv`:

```

Nombre,Edad,Nota

Ana,21,3.5

Juan,22,2.8

María,20,3.7

```

### Ejemplo de Solución para `manejo\_archivos.py`:

```python

try:

with open('entrada.txt', 'r') as archivo:

contenido = archivo.read()

print(contenido)

with open('salida.txt', 'w') as archivo\_salida:

archivo\_salida.write(contenido.upper())

except FileNotFoundError:

print("El archivo no fue encontrado.")

```

### Ejemplo de Solución para `manejo\_csv.py`:

```python

import csv

with open('estudiantes.csv', 'r') as archivo\_csv:

lector\_csv = csv.reader(archivo\_csv)

encabezados = next(lector\_csv) # Leer la primera fila (encabezados)

with open('estudiantes\_aprobados.csv', 'w', newline='') as archivo\_salida:

escritor\_csv = csv.writer(archivo\_salida)

escritor\_csv.writerow(encabezados) # Escribir los encabezados

# Filtrar estudiantes aprobados

for fila in lector\_csv:

nombre, edad, nota = fila

if float(nota) >= 3.0:

escritor\_csv.writerow(fila)

**## Clase 7: Programación Orientada a Objetos (POO) en Python**

### Objetivos de la clase:

- Introducir los conceptos básicos de la Programación Orientada a Objetos (POO).

- Aprender a definir clases y objetos en Python.

- Entender conceptos clave como herencia, encapsulamiento, y métodos.

- Practicar la creación de clases y objetos para organizar el código.

### 1. Introducción a la POO

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza "objetos" para representar datos y métodos que operan sobre esos datos. En Python, los objetos son instancias de clases, que actúan como "plantillas" para crear objetos.

### 2. Definir una Clase y Crear un Objeto

Una clase en Python se define utilizando la palabra clave `class`, seguida del nombre de la clase.

#### Ejemplo básico:

```python

class Perro:

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre

self.edad = edad

def ladrar(self):

print(f"{self.nombre} está ladrando.")

# Crear un objeto (instancia de la clase Perro)

mi\_perro = Perro("Rex", 5)

print(mi\_perro.nombre) # Imprime "Rex"

mi\_perro.ladrar() # Imprime "Rex está ladrando."

```

### 3. Atributos y Métodos

- \*\*Atributos\*\*: Son variables que pertenecen a una clase o a un objeto.

- \*\*Métodos\*\*: Son funciones que pertenecen a una clase y operan sobre sus atributos.

#### Ejemplo de atributos y métodos:

```python

class Coche:

def \_\_init\_\_(self, marca, modelo, anio):

self.marca = marca

self.modelo = modelo

self.anio = anio

def arrancar(self):

print(f"El {self.marca} {self.modelo} del {self.anio} está arrancando.")

# Crear un objeto de la clase Coche

mi\_coche = Coche("Toyota", "Corolla", 2020)

mi\_coche.arrancar() # Imprime "El Toyota Corolla del 2020 está arrancando."

```

### 4. Encapsulamiento

El encapsulamiento es una de las características principales de la POO y se refiere a la restricción del acceso directo a algunos de los componentes de un objeto. En Python, se puede encapsular un atributo haciéndolo privado (con un guion bajo `\_`).

#### Ejemplo de encapsulamiento:

```python

class CuentaBancaria:

def \_\_init\_\_(self, titular, saldo):

self.titular = titular

self.\_saldo = saldo # Atributo encapsulado (privado)

def depositar(self, cantidad):

self.\_saldo += cantidad

def retirar(self, cantidad):

if cantidad <= self.\_saldo:

self.\_saldo -= cantidad

else:

print("Fondos insuficientes")

def mostrar\_saldo(self):

print(f"Saldo actual: {self.\_saldo}")

# Crear un objeto de la clase CuentaBancaria

mi\_cuenta = CuentaBancaria("Juan", 1000)

mi\_cuenta.depositar(500)

mi\_cuenta.mostrar\_saldo() # Imprime "Saldo actual: 1500"

mi\_cuenta.retirar(2000) # Imprime "Fondos insuficientes"

```

### 5. Herencia

La herencia permite que una clase (clase hija) herede atributos y métodos de otra clase (clase padre).

#### Ejemplo de herencia:

```python

class Animal:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

def hacer\_sonido(self):

print("Este animal hace un sonido.")

class Gato(Animal):

def hacer\_sonido(self):

print(f"{self.nombre} dice miau.")

# Crear un objeto de la clase Gato

mi\_gato = Gato("Misi")

mi\_gato.hacer\_sonido() # Imprime "Misi dice miau."

```

### 6. Polimorfismo

El polimorfismo permite que diferentes clases utilicen el mismo método de diferentes maneras.

#### Ejemplo de polimorfismo:

```python

class Perro(Animal):

def hacer\_sonido(self):

print(f"{self.nombre} dice guau.")

# Crear una lista de diferentes animales

animales = [Gato("Misi"), Perro("Rex")]

# Llamar al método hacer\_sonido() en cada objeto

for animal in animales:

animal.hacer\_sonido()

```

### Tarea

#### Parte 1: Creación de Clases y Objetos

1. Crea un archivo llamado `clases\_y\_objetos.py`.

2. Define una clase llamada `Libro` con los atributos `titulo`, `autor` y `paginas`.

3. Añade un método llamado `resumen()` que imprima un resumen con el título y el autor.

4. Crea al menos dos objetos de la clase `Libro` y llama al método `resumen()` para cada uno.

#### Parte 2: Herencia y Polimorfismo

1. Crea un archivo llamado `herencia\_polimorfismo.py`.

2. Define una clase `Vehiculo` con los atributos `marca` y `modelo`. Añade un método llamado `describir()` que imprima una descripción general.

3. Define dos clases hijas `Coche` y `Moto` que hereden de `Vehiculo`. Sobrescribe el método `describir()` en ambas clases para que proporcionen una descripción más específica.

4. Crea objetos de las clases `Coche` y `Moto`, y llama al método `describir()` para cada uno.

### Ejemplo de Solución para `clases\_y\_objetos.py`:

```python

class Libro:

def \_\_init\_\_(self, titulo, autor, paginas):

self.titulo = titulo

self.autor = autor

self.paginas = paginas

def resumen(self):

print(f"'{self.titulo}' escrito por {self.autor}")

# Crear objetos de la clase Libro

libro1 = Libro("1984", "George Orwell", 328)

libro2 = Libro("Cien años de soledad", "Gabriel García Márquez", 417)

# Llamar al método resumen

libro1.resumen() # Imprime "'1984' escrito por George Orwell"

libro2.resumen() # Imprime "'Cien años de soledad' escrito por Gabriel García Márquez"

```

### Ejemplo de Solución para `herencia\_polimorfismo.py`:

```python

class Vehiculo:

def \_\_init\_\_(self, marca, modelo):

self.marca = marca

self.modelo = modelo

def describir(self):

print(f"Vehículo de marca {self.marca} y modelo {self.modelo}")

class Coche(Vehiculo):

def describir(self):

print(f"Coche de marca {self.marca} y modelo {self.modelo}")

class Moto(Vehiculo):

def describir(self):

print(f"Moto de marca {self.marca} y modelo {self.modelo}")

# Crear objetos de las clases Coche y Moto

mi\_coche = Coche("Toyota", "Corolla")

mi\_moto = Moto("Yamaha", "R1")

# Llamar al método describir

mi\_coche.describir() # Imprime "Coche de marca Toyota y modelo Corolla"

mi\_moto.describir() # Imprime "Moto de marca Yamaha y modelo R1"

**### Clase 8: Python - Estructuras de Datos Avanzadas**

\*\*1. Estructuras de Datos en Python\*\*

Python ofrece varias estructuras de datos avanzadas, más allá de las listas y diccionarios básicos:

- \*\*Conjuntos (`set`)\*\*: Una colección no ordenada de elementos únicos. Es útil para eliminar duplicados de una lista o para realizar operaciones de conjuntos como uniones, intersecciones y diferencias.

```python

conjunto = {1, 2, 3, 4, 5}

conjunto.add(6)

print(conjunto) # {1, 2, 3, 4, 5, 6}

```

- \*\*Tuplas (`tuple`)\*\*: Son similares a las listas, pero inmutables, lo que significa que no pueden cambiarse una vez creadas. Son útiles cuando necesitas una secuencia de elementos que no debería modificarse.

```python

tupla = (1, 2, 3)

print(tupla[1]) # 2

```

- \*\*Colas (`queue`)\*\*: En Python, se pueden implementar colas usando módulos como `collections.deque`. Las colas son útiles cuando se necesita acceder a los elementos en un orden específico, como el primero en entrar, primero en salir (FIFO).

```python

from collections import deque

cola = deque([1, 2, 3])

cola.append(4)

print(cola) # deque([1, 2, 3, 4])

cola.popleft()

print(cola) # deque([2, 3, 4])

```

\*\*2. Diccionarios y Conjuntos Anidados\*\*

Los diccionarios y conjuntos pueden ser anidados, lo que permite representar estructuras de datos más complejas:

- \*\*Diccionarios anidados\*\*: Se pueden usar para representar estructuras jerárquicas, como un árbol o un grafo.

```python

diccionario\_anidado = {

"frutas": {"manzana": 10, "naranja": 20},

"verduras": {"zanahoria": 30, "brocoli": 40}

}

print(diccionario\_anidado["frutas"]["manzana"]) # 10

```

- \*\*Conjuntos anidados\*\*: Aunque no se pueden tener conjuntos dentro de otros conjuntos (ya que los conjuntos deben ser inmutables), se pueden usar otros tipos de estructuras como frozenset (un conjunto inmutable) para lograr algo similar.

```python

conjunto\_anidado = {frozenset({1, 2}), frozenset({3, 4})}

print(conjunto\_anidado) # {frozenset({1, 2}), frozenset({3, 4})}

```

\*\*3. Comprensión de Listas, Diccionarios y Conjuntos\*\*

Las comprensiones son una manera concisa de crear listas, diccionarios y conjuntos a partir de iterables:

- \*\*List Comprehensions\*\*:

```python

lista = [x \* 2 for x in range(5)]

print(lista) # [0, 2, 4, 6, 8]

```

- \*\*Dictionary Comprehensions\*\*:

```python

diccionario = {x: x \* 2 for x in range(5)}

print(diccionario) # {0: 0, 1: 2, 2: 4, 3: 6, 4: 8}

```

- \*\*Set Comprehensions\*\*:

```python

conjunto = {x \* 2 for x in range(5)}

print(conjunto) # {0, 2, 4, 6, 8}

```

\*\*Tarea para Practicar:\*\*

1. \*\*Creación de una Lista de Pares\*\*: Crea una lista de los primeros 20 números enteros y luego utiliza una comprensión de lista para crear una nueva lista que contenga solo los números pares.

2. \*\*Operaciones con Conjuntos\*\*: Crea dos conjuntos de números enteros. Realiza las operaciones de unión, intersección y diferencia entre estos conjuntos.

3. \*\*Diccionario Anidado\*\*: Crea un diccionario anidado que represente un inventario de una tienda con diferentes categorías (por ejemplo, frutas y verduras) y subcategorías. Luego, escribe un código que te permita actualizar la cantidad de un artículo específico.

4. \*\*Implementación de una Cola\*\*: Usa `collections.deque` para implementar una cola que permita agregar elementos al final y eliminar elementos del principio. Añade al menos 5 elementos a la cola y luego elimina 2, mostrando el estado de la cola antes y después de las eliminaciones.

**### Clase 9: Python - Manejo de Excepciones y Archivos**

\*\*1. Manejo de Excepciones en Python\*\*

Las excepciones son errores que ocurren durante la ejecución de un programa. Python ofrece una forma robusta de manejar estos errores usando el bloque `try-except`.

- \*\*Estructura básica\*\*:

```python

try:

# Código que podría causar una excepción

resultado = 10 / 0

except ZeroDivisionError:

# Código que se ejecuta si ocurre una excepción

print("No se puede dividir por cero.")

```

- \*\*Múltiples excepciones\*\*: Puedes manejar diferentes tipos de excepciones con múltiples bloques `except`.

```python

try:

resultado = 10 / 0

except ZeroDivisionError:

print("No se puede dividir por cero.")

except TypeError:

print("Error de tipo de datos.")

```

- \*\*Bloque `else`\*\*: Se ejecuta si no ocurre ninguna excepción.

```python

try:

resultado = 10 / 2

except ZeroDivisionError:

print("No se puede dividir por cero.")

else:

print("El resultado es:", resultado)

```

- \*\*Bloque `finally`\*\*: Se ejecuta independientemente de si ocurre una excepción o no, útil para liberar recursos.

```python

try:

resultado = 10 / 2

finally:

print("Esta línea se ejecuta siempre.")

```

\*\*2. Manejo de Archivos en Python\*\*

Python permite trabajar fácilmente con archivos para leer, escribir y actualizar información.

- \*\*Abrir y leer archivos\*\*:

```python

archivo = open('archivo.txt', 'r')

contenido = archivo.read()

print(contenido)

archivo.close()

```

Es importante cerrar el archivo después de trabajar con él para liberar recursos. Una mejor práctica es usar el bloque `with`:

```python

with open('archivo.txt', 'r') as archivo:

contenido = archivo.read()

print(contenido)

```

- \*\*Escribir en archivos\*\*:

```python

with open('archivo.txt', 'w') as archivo:

archivo.write("Nueva línea de texto")

```

El modo `'w'` sobrescribe el archivo si ya existe. Para añadir contenido sin sobrescribir, usa `'a'` (modo de adición):

```python

with open('archivo.txt', 'a') as archivo:

archivo.write("\nOtra línea de texto")

```

- \*\*Leer líneas de un archivo\*\*:

```python

with open('archivo.txt', 'r') as archivo:

for linea in archivo:

print(linea.strip()) # `strip()` elimina espacios en blanco al inicio y final de la línea

```

\*\*Tarea para Practicar:\*\*

1. \*\*Manejo de Excepciones\*\*: Escribe un programa que pida al usuario que ingrese dos números y realice la división entre ellos. Maneja las excepciones para evitar divisiones por cero y errores de tipo de datos (por ejemplo, si el usuario ingresa texto en lugar de un número).

2. \*\*Lectura y Escritura de Archivos\*\*: Crea un archivo de texto que contenga una lista de productos y precios. Luego, escribe un programa que lea el archivo, muestre los productos y precios, y permita al usuario añadir un nuevo producto y precio al archivo.

3. \*\*Lectura de Archivos con Excepciones\*\*: Modifica el programa de lectura de archivos para manejar la excepción que ocurre si el archivo no existe, mostrando un mensaje de error adecuado.

4. \*\*Operaciones de Archivos y Excepciones\*\*: Escribe un programa que lea un archivo de texto con números en cada línea, calcule la suma de todos los números y maneje cualquier excepción que pueda ocurrir si una línea no contiene un número válido.

**### Clase 10: Python - Programación Orientada a Objetos (POO)**

\*\*1. Fundamentos de la Programación Orientada a Objetos (POO)\*\*

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el uso de "objetos", los cuales son instancias de "clases". Las clases actúan como un plano o molde para crear objetos.

- \*\*Clase\*\*: Una plantilla que define las propiedades y comportamientos de los objetos.

```python

class Persona:

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre

self.edad = edad

def saludar(self):

print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

```

- \*\*Objeto\*\*: Una instancia de una clase.

```python

persona1 = Persona("Juan", 30)

persona1.saludar() # Salida: Hola, mi nombre es Juan y tengo 30 años.

```

- \*\*Atributos\*\*: Variables que pertenecen a una clase o a los objetos creados a partir de ella.

- \*\*Métodos\*\*: Funciones que pertenecen a una clase y definen los comportamientos de los objetos.

\*\*2. Encapsulamiento\*\*

El encapsulamiento es el principio de restringir el acceso a ciertos detalles de un objeto y proteger la integridad de sus datos. En Python, se utilizan guiones bajos para indicar que un atributo o método es privado (aunque el acceso no está estrictamente restringido).

```python

class Persona:

def \_\_init\_\_(self, nombre, edad):

self.\_nombre = nombre # Atributo "protegido"

self.\_edad = edad

def \_detalles(self): # Método "protegido"

print(f"Nombre: {self.\_nombre}, Edad: {self.\_edad}")

def saludar(self):

print(f"Hola, soy {self.\_nombre}.")

```

\*\*3. Herencia\*\*

La herencia permite crear una nueva clase basada en una clase existente, heredando atributos y métodos de la clase base.

```python

class Empleado(Persona):

def \_\_init\_\_(self, nombre, edad, salario):

super().\_\_init\_\_(nombre, edad) # Llama al constructor de la clase base

self.salario = salario

def detalles\_empleado(self):

print(f"Nombre: {self.\_nombre}, Edad: {self.\_edad}, Salario: {self.salario}")

```

- \*\*Clase base\*\* (superclase): La clase original que proporciona propiedades y métodos a las clases derivadas.

- \*\*Clase derivada\*\* (subclase): Una clase que hereda de otra clase.

\*\*4. Polimorfismo\*\*

El polimorfismo permite que diferentes clases utilicen métodos con el mismo nombre de manera diferente.

```python

class Animal:

def hablar(self):

pass

class Perro(Animal):

def hablar(self):

print("Guau!")

class Gato(Animal):

def hablar(self):

print("Miau!")

def hacer\_hablar(animal):

animal.hablar()

perro = Perro()

gato = Gato()

hacer\_hablar(perro) # Guau!

hacer\_hablar(gato) # Miau!

```

\*\*Tarea para Practicar:\*\*

1. \*\*Crea una Clase `Coche`\*\*: Define una clase `Coche` con atributos como `marca`, `modelo`, y `año`. Incluye un método para mostrar la información del coche. Luego, crea varios objetos `Coche` y muestra su información.

2. \*\*Encapsulamiento y Métodos Privados\*\*: Modifica la clase `Coche` para incluir un atributo privado `\_kilometraje` y un método para actualizar este valor de manera controlada. Implementa métodos para aumentar y mostrar el kilometraje.

3. \*\*Herencia\*\*: Crea una clase derivada `CocheEléctrico` que herede de `Coche`, agregando un atributo `bateria` y un método para mostrar la capacidad de la batería. Crea un objeto de `CocheEléctrico` y muestra su información, incluida la capacidad de la batería.

4. \*\*Polimorfismo\*\*: Implementa dos clases adicionales, `CocheDeportivo` y `CocheFamiliar`, que hereden de `Coche`. Cada una debe tener un método `tipo\_coche` que indique si es un coche deportivo o familiar. Crea una función que reciba un objeto `Coche` y muestre su tipo utilizando polimorfismo.

**### Clase 11: Python - Módulos y Paquetes**

\*\*1. Módulos en Python\*\*

Un módulo en Python es un archivo que contiene definiciones y declaraciones de Python. Este archivo puede incluir funciones, clases y variables que puedes importar y reutilizar en otros scripts.

- \*\*Importar un Módulo\*\*: Puedes importar un módulo utilizando la palabra clave `import`.

```python

import math

print(math.sqrt(16)) # 4.0

```

- \*\*Importar Funciones Específicas\*\*: Si solo necesitas algunas funciones del módulo, puedes importarlas directamente.

```python

from math import sqrt, pi

print(sqrt(25)) # 5.0

print(pi) # 3.141592653589793

```

- \*\*Renombrar un Módulo\*\*: Puedes usar la palabra clave `as` para dar un alias a un módulo.

```python

import math as m

print(m.sqrt(9)) # 3.0

```

- \*\*Crear un Módulo Propio\*\*: Puedes crear tus propios módulos escribiendo funciones y guardándolas en un archivo `.py`.

```python

# archivo: operaciones.py

def sumar(a, b):

return a + b

def restar(a, b):

return a - b

```

Luego, puedes importar este módulo en otro script:

```python

from operaciones import sumar, restar

print(sumar(5, 3)) # 8

print(restar(10, 4)) # 6

```

\*\*2. Paquetes en Python\*\*

Un paquete es una colección de módulos organizados en un directorio. Para que un directorio sea considerado un paquete, debe contener un archivo `\_\_init\_\_.py`, que puede estar vacío o ejecutar algún código de inicialización.

- \*\*Estructura de un Paquete\*\*:

```

mi\_paquete/

\_\_init\_\_.py

modulo1.py

modulo2.py

```

- \*\*Importar desde un Paquete\*\*:

```python

from mi\_paquete.modulo1 import funcion1

funcion1()

```

\*\*3. Uso de Paquetes Estándar\*\*

Python incluye una gran cantidad de paquetes estándar que se pueden usar para diversas tareas, como `os` para operaciones del sistema, `sys` para manipulación de la consola, y `datetime` para trabajar con fechas y horas.

- \*\*Ejemplo con `os`\*\*:

```python

import os

print(os.getcwd()) # Muestra el directorio de trabajo actual

```

- \*\*Ejemplo con `datetime`\*\*:

```python

import datetime

hoy = datetime.date.today()

print(hoy) # Muestra la fecha actual

```

\*\*Tarea para Practicar:\*\*

1. \*\*Crear un Módulo\*\*: Escribe un módulo llamado `calculadora.py` que contenga funciones para sumar, restar, multiplicar y dividir. Luego, crea un script separado para importar y utilizar estas funciones, realizando algunas operaciones básicas.

2. \*\*Crear un Paquete\*\*: Crea un paquete llamado `geometria` que contenga dos módulos: `area.py` y `perimetro.py`. El módulo `area.py` debe incluir funciones para calcular el área de un cuadrado y un círculo, mientras que `perimetro.py` debe contener funciones para calcular el perímetro de un cuadrado y un círculo. Escribe un script separado que importe estos módulos y realice algunos cálculos.

3. \*\*Uso de Módulos del Sistema\*\*: Utiliza el módulo `os` para listar todos los archivos en un directorio específico. Usa el módulo `datetime` para obtener y mostrar la fecha y hora actual en el mismo script.

Estas tareas te permitirán practicar la creación y el uso de módulos y paquetes en Python, así como trabajar con algunos módulos estándar del lenguaje.

**Clase 12: Python - Comprehensions y Expresiones Generadoras**

\*\*1. List Comprehensions\*\*

Una `list comprehension` es una forma concisa de crear listas en Python. Permite generar una nueva lista aplicando una expresión a cada elemento de una secuencia o iteración, de forma más compacta que con un bucle tradicional.

- \*\*Sintaxis Básica\*\*:

```python

[expresion for item in iterable]

```

- \*\*Ejemplo\*\*: Crear una lista con los cuadrados de los números del 1 al 5.

```python

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

print(cuadrados) # [1, 4, 9, 16, 25]

```

- \*\*Condiciones en List Comprehensions\*\*:

Puedes agregar una condición para filtrar los elementos que se incluirán en la nueva lista.

- \*\*Ejemplo\*\*: Crear una lista con los números pares del 1 al 10.

```python

pares = [x for x in range(1, 11) if x % 2 == 0]

print(pares) # [2, 4, 6, 8, 10]

```

\*\*2. Dictionary Comprehensions\*\*

Al igual que con las listas, puedes crear diccionarios de manera compacta utilizando `dictionary comprehensions`.

- \*\*Sintaxis Básica\*\*:

```python

{key: value for item in iterable}

```

- \*\*Ejemplo\*\*: Crear un diccionario con números del 1 al 5 como claves y sus cuadrados como valores.

```python

cuadrados = {x: x\*\*2 for x in range(1, 6)}

print(cuadrados) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

```

\*\*3. Set Comprehensions\*\*

De manera similar, puedes crear conjuntos (`sets`) usando `set comprehensions`.

- \*\*Sintaxis Básica\*\*:

```python

{expresion for item in iterable}

```

- \*\*Ejemplo\*\*: Crear un conjunto con los caracteres únicos en una cadena.

```python

caracteres = {letra for letra in "abracadabra"}

print(caracteres) # {'a', 'b', 'r', 'c', 'd'}

```

\*\*4. Expresiones Generadoras\*\*

Las expresiones generadoras permiten crear iteradores en una forma similar a las `list comprehensions`, pero en lugar de crear una lista en memoria, generan los elementos sobre la marcha.

- \*\*Sintaxis Básica\*\*:

```python

(expresion for item in iterable)

```

- \*\*Ejemplo\*\*: Crear un generador que devuelva los cuadrados de los números del 1 al 5.

```python

cuadrados = (x\*\*2 for x in range(1, 6))

for numero in cuadrados:

print(numero)

```

A diferencia de las listas, los generadores no almacenan todos los valores en memoria, lo que los hace más eficientes en términos de uso de memoria cuando se trabaja con grandes volúmenes de datos.

\*\*Tarea para Practicar:\*\*

1. \*\*List Comprehension con Condición\*\*: Crea una lista de los primeros 50 números, pero solo incluye aquellos que sean divisibles por 5. Usa `list comprehension` para lograrlo.

2. \*\*Dictionary Comprehension\*\*: Escribe un `dictionary comprehension` que cree un diccionario a partir de una lista de palabras, donde la clave sea la palabra y el valor sea la longitud de esa palabra.

3. \*\*Set Comprehension\*\*: Crea un `set comprehension` para generar un conjunto de las vocales que aparecen en una frase dada.

4. \*\*Expresión Generadora\*\*: Escribe una expresión generadora que devuelva los primeros 100 números cuadrados. Recorre el generador con un bucle para mostrar estos números.

Para comparar tu respuesta, asegúrate de que las `comprehensions` (listas, diccionarios, sets) y la expresión generadora estén correctamente implementadas con la sintaxis adecuada. Verifica que las operaciones se realicen de forma eficiente y que los resultados sean los esperados.