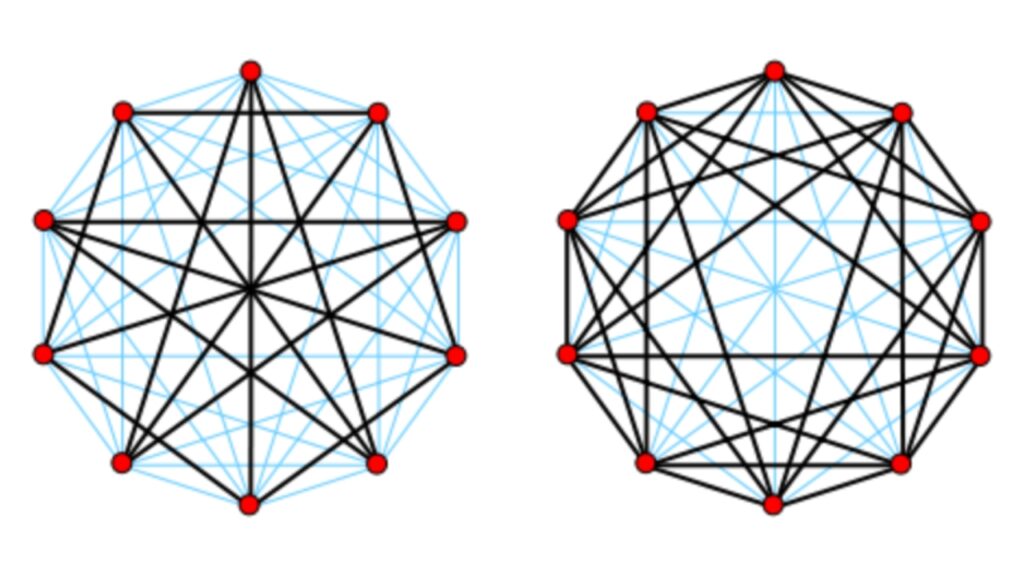
**TECNOLOGÍA DE LA PROGRAMACIÓN**

**2º DTIE MATEMÁTICAS Y FÍSICA UNIVERSIDAD DE MURCIA**



**SESIÓN 1: INTRODUCCIÓN A LA PROGRAMACIÓN EN PYTHON**

Dos tipos de datos:

1. Primitivos 🡪 carácter, numérico, booleano, enumerado
2. Compuestos 🡪 string, array, registro, lista etc.

Las variables son las ubicaciones de almacenamiento de los datos. Una constante será una variable que solo puede declararse una vez. En Python no hace falta declarar los tipos de datos de las variables, pero vamos a hacerlo por limpieza.

Por ejemplo:

x:int = 5

x:int = “hola” #funcionaría perfectamente pero vamos a evitarlo

En Python escribimos las variables separadas con guiones. Python no tiene por defecto **constantes** pero nosotros las escribiremos declarando el nombre en mayúscula para así distinguirlas. Además podemos asignar varias variables simultáneamente como:

x, y, z = 1, 2, 3

Python tampoco distingue entre comillas simples y comillas dobles para los strings.

**TIPOS DE DATOS COMPUESTOS**

Las **listas** en Python pueden cambiar el tamaño automáticamente

lista:list = [2,3,”hola”,3.14]

Para acceder a las primeras posiciones nos referiremos como:

lista[0], lista[1], …, lista[n]

También podemos acceder a la última posición utilizando

lista[-1]

Los **conjuntos** son una colección de elementos arbitrarios únicos.

conjunto:set = {1,2,3}

Si por ejemplo ahora añadiésemos el elemento tres, no nos dejaría (porque ya está en el conjunto). Un **diccionario** es un conjunto identificado por unas claves (keys) y unos parámetros (ítems).

diccionario:dict = {“a”:1, “b”:2, “c”:3}

Una **tupla** es una lista congelada, una lista que no puede cambiar una vez ha sido declarada.

tupla:tuple[int, str, int] = (1,”hola”,3)

El **rango** es una función que genera elementos de una tupla equiespaciados como :

range(START, STOP, salto) # EL SEGUNDO NO ESTÁ INCLUIDO, COMO EN PROCESSING, [START, STOP)

Un **conjunto congelado** (frozenset) es un conjunto que no puede cambiar una vez ha sido declarado.

conjunto\_congelado : frozenset[int, int, int] = {1,2,3}

Decimos que una variables es mutable si la puedo cambiar sin cambiar su reserva en memoria. Donde antes ponía un cuatro, ahora pongo un tres. Una variable es inmutable si para cambiarla he de crear un nuevo espacio en memoria. En Python podemos ver la dirección en memoria de una variable utilizando la función *id().*

x :str= “pablo”

print(f”El espacio reservado para la memoria de la variable x es: {id(x)}”)

**CASTING**

Python define funciones par alos cambios de tipo.

int(), float(), str(), list(), tuple(), set() etc.

Por ejemplo :

tupla :tuple[int, int, int] = (1,2,3)

tupla\_a\_lista = list(tupla)

**ESTRUCTURA SECUENCIAL Y BUILT IN FUNCTIONS**

Python permite estructuras algebráicas de toda la vida, y además:

- leer por pantalla: input(texto)

- escribir por pantalla: print(texto)

- tamaño de la lista: len(lista)

- tipo de una variable: type(variable)

Los condicionales funcionan como en cualquier lenguaje y mantienen la estructura sintáctica de las operaciones aritméticas, cambiando la sintaxis de las lógicas (and=&&, or=||, not=! y sucesivas).

if condicion:

B1

else:

B2

Que podemos escribir en una sola línea como (inline)

if condicion: B1

O con el else:

B1 if condicion else B2

Y condicionales anillados como:

if condicion1:

B1

elif condicion2:

B2

elif

...

else:

Bn

Utilizamos los bucles for cuando sabemos de dónde a dónde vamos y el bucle while cuando no sabemos dónde tenemos que acabar (más búsquedas que recorridos). **Así que EVITAMOS! for-if-return**

while variable\_de\_control:

B1

estructura que modifica variable\_de\_control

Así, la estructura del bucle for será:

for elemento in [a,b,c,...]:

B1(elemento)

O si queremos crear un bucle for en un rango numérico:

for num in range(10,20,3):

B1(num)

Despúes de un while podemos poner un else: NO se evalúa si se ejecuta un break.

while condicion:

B1

else:

Bloque que se ejecuta siempre despúes del while

**FUNCIONES Y PROCEDIMIENTOS**

Una función no debe tener más de una responsabilidad o un propósito. Declaramos funciones como:

def funcion(param1:tipo\_param1, param2:tipo\_param2, …, paramN:tipo\_paramN, \*args, \*\*kwargs) -> tipo\_return:

Expresión(param1, param2, … paramN)

return valor esperado

Una función puede devolver muchos valores, agrupados en una tupla

coords = space\_coordinates(time)

print(coords) # 🡪 (x,y,z)

Una función que no devuleve ningún valor se llama **procedimiento** (antiguo void). En Python podemos establecer valores predeterminados para los parámetros de las funciones.

def funcion(a:int, b:int, c:int = 3, d:int=5)

Podemos incluso invocar la función refiriendonos al nombre de los parámetros:

Funcion(a=7, c=2, b=4, d=9)

Por ejemplo:

def g(x:int, y:int, \*args, \*\*kwargs) -> int:

return x+y+args[0]+kwargs[“key”]

PROBLEMA 2. Crea un programa que reciba un valor numérico en formato string desde input, lo convierta a entero y luego a real y realice una operación matemática con él.

print(f”El resultado de la operación es: {1.5\*float(int(input(“Introduce: “)))}”)

PROBLEMA 3. Escribe un programa que reciba una frase del usuario y realice las siguientes operaciones:

1. Convertir toda la frase a mayúsculas
2. Convertir toda la frase a minúsculas
3. Aplicar capitalize a la frase y mostrar el resultado
4. Aplicar title() a la frase para capitalizar cada palabra
5. Solicitar al usuario una letra y mostrar el código ASCII de esa letra utilizando ord. Luego mostrar el caractér correspondiente a un có9digo ASCII ingresado por el usuario utilizando la función chr
6. Contar cuantas veces aparece una letra específica en la frase (la letra ha de ser introducida por el usuario).

frase = input("Introduce tu frase: ")

print(f"Frase en mayúsculas: {upper(frase)}")

print(f"Frase en minúsculas: {lower(frase)}")

print(f"Frase capitalizada: {capitalice(frase)}") # Primera letra en mayúsculas

print (f"Frase capitalizada por palabra: {title(frase)}") # Primera letra de cada palabra en mayúsculas

letra:str = input("Introduce una letra: ")

print(f"El código ASCII de la letra {letra} es: {ord(letra)}")

ascii:str = input("Introduce un código ascii: ")

print(f"La letra asociada al código ASCII {ascii} es: {chr(asci)}")

PROBLEMA 4. Listas (mutables). Define una lista con cinco números de forma manual. Por ejemplo, con los valores [10, 20, 30, 40, 50]. Luego, realiza las siguientes operaciones en el siguiente orden: Consultar el valor del tercer elemento de la lista, obtener una porción (slice) de la lista que contenga los elementos desde la posición 1 a la 3 (sin incluir la posición 3), agregar un nuevo número al final de la lista (por ejemplo, el 60), eliminar el segundo número de la lista, generar un número aleatorio y añadirlo como primera posición de la lista, ordenar la lista de mayor a menor.

def problema4 (lista):

print(lista[2])

print(lista[0:2])

lista.append(60)

print(lista)

lista.pop(1)

print(lista)

numero\_aleatorio:int = random.randint(1,100)

lista.insert(0, numero\_aleatorio)

print(lista)

lista.sort(reverse=True)

print(lista)

PROBLEMA 5. Define manualmente una tupla que contenga los nomnbres de cuatro ciudades. Luego realiza las siguientes operaciones: Accede al segundo valor de la tupla y muéstralo por pantalla, convierte la tupla en una lista para ser modificada, cambia el valor de la tercera ciudad por otro nombre de ciudad de tu elección. Convierte la lista modificada de nuevo en una tupla utilizando tuple y muestra por pantalla la nueva tupla.

ciudades:tuple[str, str, str] = (“Madrid”, “Barcelona”, “Málaga”)

print(f”El segundo elemento de la tupla es: {ciudades[1]}”)

ciudades\_lista:list = list(ciudades)

ciudades\_lista[2] = “Murcia”

ciudades\_nueva\_tupla:tuple[str, str, str] = tuple(ciudades\_lista)

print(f”La nueva tupla es: {ciudades\_nueva\_tupla}”)

PROBLEMA 6. Escribe un programa que manualmente defina dos conjuntos A, B y luego realice las siguientes operaciones: unión de ambos conjuntos,intersección de ambos conjuntos, elementos que están en el primer conjunto pero no en el segundo ie A-B, añade un nuevo número al primer conjunto, elimina un número del segundo conjunto.

**Para añadir un elemento a un conjunto NO podemos hacer .append( elemento) sino que tenemos que hacer .add (elemento)**

A:set = {1,2,3,4,5,6,7,8,9,10}  
B:set = {2,4,6, 8, 10, 12}  
def union(A:set, B:set) -> set:  
 C:set = A  
 for elemento in B:  
 C.add(elemento)  
 return C  
def interseccion(A:set, B:set) -> set:  
 C:set = set()  
 for elemento in A:  
 if elemento in B:  
 C.add(elemento)  
 for elemento in B:  
 if elemento in A:  
 C.add(elemento)  
 return C  
def resta(A:set, B:set) -> set:  
 C:set = set()  
 for elemento in A:  
 if elemento not in B:  
 C.add(elemento)  
 return C  
def anyade(A:set, elemento:int) -> set:  
 A.add(elemento)

return A  
def elimina(A:set, elemento:int) -> set:  
 A.remove(elemento)

return A  
  
print(union(A, B))  
print(interseccion(A,B))  
print(resta(A,B))  
print(anyade(A, 100))  
print(elimina(A, 4))

PROBLEMA 7. **Crea un diccionario donde las claves sean los nombres de cuatro estudiantes y los valores sean sus respectivas**

**notas en un examen. Luego, realiza las siguientes operaciones, mostrando el diccionario actualizado tras cada operación:**

* Muestra el diccionario original.
* Muestra el listado de estudiantes y el número total de estudiantes en el diccionario.
* Muestra el listado de las notas almacenadas, sin mostrar los nombres de los estudiantes.
* Añade un nuevo estudiante y su nota al diccionario.
* Modifica la nota de uno de los estudiantes.
* Elimina a un estudiante del diccionario.

estudiantes:dict[str:int] = {

"Paco Martínez": 8,

"Laura Norte": 5,

"Sara López": 10,

"Juan Pérez": 3

}

print(estudiantes)

nombres\_estudiantes = [e for e in estudiantes.keys()]

print(f"Nombres de estudiantes: {nombres\_estudiantes}")

print(f"Número total de estudiantes: {len(nombres\_estudiantes)}")

notas\_estudiantes = [n for n in estudiantes.values()]

print(f"Notas de los estudiantes: {notas\_estudiantes}")

estudiantes["Sara Lopez"] = 9

print(estudiantes)

estudiantes.pop("Juan Pérez")

print(estudiantes)

PROBLEMA 8. Escribe un programa que determine si un cliente de una tienda es elegible para un descuento especial. El programa debe solicitar al cliente dos datos: 1) La cantidad de dinero que ha gastado en la tienda durante el último mes (en euros), y 2) Si tiene una membresía premium (si/no). Nota: representar internamente la membresía como un booleano. El programa debe aplicar las siguientes reglas para determinar si el cliente recibe un descuento:

* Si el cliente ha gastado más de 100=C y tiene una membresía premium, se le otorga un descuento del 20 %.
* Si el cliente ha gastado más de 100=C o tiene una membresía premium, se le otorga un descuento del 10 %.
* Si no cumple ninguna de las condiciones anteriores, no se le otorga ningún descuento.

El programa debe mostrar el porcentaje de descuento que recibe el cliente.

dinero\_gastado:float = float(input("Dinero gastado en la tienda: "))  
premium:str = input("Tiene premium? (si/no): ")  
membresia:bool = True if premium == "si" else False  
  
descuento:float = 0  
if (dinero\_gastado > 100 and membresia):  
 descuento = 0.2  
elif (dinero\_gastado > 100 or membresia):  
 descuento = 0.1  
  
print(f"El cliente recibe un descuento del {descuento\*100}%")

PROBLEMA 9. **Bucle while**. Escribe un programa que solicite al usuario un número entero positivo. El programa debe sumar los números desde 1 hasta ese número, pero se detendrá si la suma acumulada supera un valor límite, que también debe ser proporcionado por el usuario. Utiliza un bucle while con una expresión booleana en la condición para controlar el proceso. Reglas: el bucle debe continuar mientras el número actual sea menor o igual al número proporcionado y la suma acumulada sea menor o igual al límite establecido

x:int = int(input("Introduzca un número entero positivo: "))  
limite:int = int(input("Introduzca el valor límite de la suma: "))  
  
suma:int = 0  
numero\_actual=0  
while((suma + numero\_actual) < limite and numero\_actual <= x):  
 suma+=numero\_actual  
 numero\_actual+=1  
  
print(f"La suma final es: {suma}")

PROBLEMA 10. Bucle for con range(). Escribe un programa que solicite al usuario un número entero positivo que representará el tamaño de una palabra. El programa debe componer un string de dicho tamaño utilizando la función range(). En cada iteración del bucle for, se generará un número aleatorio del 0 al 9, se mostrará la posición actual, y se concatenará el número al string resultante. Finalmente, el programa deberá mostrar el string generado.

def problema10():  
 palabra:str =""  
 size:int = int(input("Tamaño de la palabra: "))  
 for i in range(0,size):  
 random\_number:int = random.randint(0,9)  
 print(f"Posicion actual {i}")  
 palabra+=str(random\_number)  
 print(palabra)

PROBLEMA 11. Bucle for con listas. Escribe un programa que genere una lista de tamaño aleatorio entre 3 y 7. Luego, usando un bucle for, llena la lista con números aleatorios entre 0 y 9 (ambos inclusive). Finalmente, mostraremos el contenido dela lista de tres formas diferentes Hacer print() dire tamente sobre la variable de tipo lista creada. Usar un bucle for con range() para mostrar el contenido indicando el índice de la posición y su valor. Nota: considerar el uso de len(). Como tercera forma, utilizar la iteración del for como contador.

def problema11():  
 size:int = random.randint(3,7)  
 lista = []  
 for i in range(size):  
 lista.append(random.randint(0,9))  
 print(lista)  
 for pos in range(len(lista)):  
 print(lista[pos])  
 for elem in lista:  
 print(elem)

PROBLEMA 12. Funciones con parámetros obligatorios. Escribe una función que reciba dos números enteros como parámetros y  
retorne el mayor de ellos. El programa debe generar aleatoriamente cinco pares de números, y para cada par, utilizar  
la función para encontrar y mostrar el mayor número. ¿Sabrías documentar la función con Docstring?

def problema12(x:int, y:int)->None:  
 print(f"El mayor número entre {x} e {y} es: {x if x > y else y}")

PROBLEMA 13.Funciones con parámetros opcionales. Crea una función que reciba el nombre de una persona y su edad, siendo la edad un parámetro opcional (por defecto 18). La función debe imprimir un mensaje del tipo: "Hola [nombre], tienes [edad] años". Si no se proporciona la edad, debe imprimir el valor por defecto.

def problema13(name:str, edad:int=18)->None:  
 print(f"Hola {name}, tienes {edad} años")

PROBLEMA 14. Funciones con argumentos empaquetados. Escribe una función que reciba una cantidad variable de números (\*args) y devuelva el promedio de todos ellos. Además, la función debe aceptar argumentos con nombre (\*\*kwargs) para mostrar mensajes personalizados al usuario (por ejemplo, un mensaje de bienvenida y otro de despedida). Ejemplo de salida: ¡Hola! Bienvenido a la calculadora de promedios. Gracias por usar el programa. El promedio es: 30.0

def problema14(\*args, \*\*kwargs):  
 suma:float = 0  
 for n in args:  
 suma+=n  
 promedio:float = suma/len(args)  
  
 if "bienvenida" in kwargs.keys() and "despedida" in kwargs.keys():  
 print(f"{kwargs["bienvenida"]}")  
 print(promedio)  
 print(f"{kwargs["despedida"]}")  
 else:  
 print(promedio)

PROBLEMA 15. Define una función que reciba como argumento obligatorio el nombre deuna persona, un parámetro opcional que represente el saldo (por defecto 0), y una serie de transacciones utilizando \*args para sumar al saldo. Por ejemplo, las transacciones a realizar se pueden expresar como [100, -50, 200,-30], donde números positivos indicarían depósitos a cuenta (aumentar el valor de la cuenta), y números negativos retiradas de efectivo (valores negativos). Además, recibe \*\*kwargs con información adicional sobre la persona (como dirección o número de cuenta). La función debe devolver dos valores: el saldo final después de aplicar lastransacciones y un string con un mensaje ilustrativo detallando los aspectos adicionales de la persona.

def problema15(nombre:str, saldo:float = 0, \*args, \*\*kwargs)-> (int, str):  
 saldo\_final:float = saldo  
 print(args)  
 for transaccion in args[0]:  
 saldo\_final += transaccion  
 string\_final = ""  
 for key, argument in kwargs.items():  
 string\_final += f"{key} --> {argument}"  
 return saldo\_final, string\_final

En Python un **enumerado** es una colección de nombres fijos (llamados miembros) asociados a valores constantes. Se utiliza para tener un referencial de nombre-valor estructurado, sin utilizar muchas variables sueltas que actúen como claves. Para utilizarlo hemos de importar la clase *Enum* del módulo enum.

from enum import Enum

# Definicion de un enumerado

class ClaseQueHeredaDeUnEnumerado(Enum):

VALOR1:str = “Valor referido al nombre VALOR1”

VALOR2:str = “Valor referido al nombre VALOR2”

…

VALORN:str = “Valor referido al nombre ValorN”

valorElegido:ClaseQueHeredaDeUnEnumerado = ClaseQueHeredaDeUnEnumerado.VALOR2

print(valorElegido.name) # 🡪 VALOR2

PROBLEMA 16. Enumerados. Define un enumerado Color para gestionar colores. El programa tendrá disponibles los siguientes colores:

* ROJO: Representado por el número 1.
* VERDE: Representado por el número 2.
* AZUL: Representado por el número 3.
* AMARILLO: Representado por el número 4.

Tras ello, programa la siguiente funcionalidad: Define una variable de tipo Color que represente tu color favorito de entre los disponibles, y muestra su valor por pantalla. Crea una función mostrar\_color que reciba un Color y muestre por pantalla su nombre y su valor. Crea una función capaz de mostrar todos los colores disponibles en el enumerado utilizando un bucle for

Nota: podemos usar un bucle for para iterar las opciones del enumerado con for color in Color. Para acceder al nombre utilizaremos Class.VALOR.name y para acceder al valor asociado a esa clave utilizaremos Class.VALOR.value, si hacemos simplemente Class.VALOR obtendremos un string “class.VALOR”

from enum import Enum  
  
class Color(Enum):  
 ROJO:int = 1  
 VERDE:int = 2  
 AZUL:int = 3  
 AMARILLO:int = 4  
  
color\_escogido:Color = Color.AMARILLO  
print(f"Valor: {color\_escogido.value}")  
print(f"Nombre: {color\_escogido.name}") # ES UN ATRIBUTO, NO UN MÉ***TODO***def mostrar\_color(color:Color) -> None:  
 print(f"Valor: {color.value}, Nombre: {color.name}")  
  
def mostrar\_todos\_colores() -> None:  
 for color in Color:  
 mostrar\_color(color)  
  
mostrar\_color(Color.AZUL)  
mostrar\_todos\_colores()

PROBLEMA DESAFÍO: Crear un programa que dado un número real x, muestre su representación en base b:

import math  
  
def calcular\_siguiente\_digito(x:int, b:int, digito\_actual) -> int:  
digito = math.floor(x / pow(b, digito\_actual))  
return digito  
  
def calcular\_repr(x:int, b:int) -> int:  
digitos:list = []  
digito\_actual = math.floor(math.log(x, b))  
while x >= b and digito\_actual > 0:  
digito:int = calcular\_siguiente\_digito(x, b, digito\_actual)  
digitos.append(digito)  
x -= digito \* pow(b, digito\_actual)  
digito\_actual -= 1  
digitos.append(x)  
  
repr:str = ""  
for d in digitos:  
repr += str(d)  
return int(repr)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
x:int = int(input("INTRODUZCA EL NÚMERO : "))  
b:int = int(input("INTRODUZCA LA BASE: "))  
if (x < 0):  
print("El número ha de ser un entero positivo")  
elif (b < 2):  
print("La base ha de ser mayor o igual que dos")  
else:  
repr:int = calcular\_repr(x, b)  
print(f"La representación del número {x} en base ({b}) es: {repr}")

**SESIÓN 2: MIEMBROS, SOBRECARGA Y VISIBILIDAD**

Cada clase representa a un conjunto de objetos con la misma estructura. La información que contiene en las clases se llama miembro (por ejemplo, un miembro puede ser un atributo, un método).

* Atributo 🡪 Una característica (tipo variable) que tiene la clase (ej: la clase Persona tendría un atributo “ojo”). Se define edsde el constructor:

class NombreDeLaClase:

def \_\_init\_\_(self, param1:tipo param1, …, paramN: tipo\_param\_N): # Constructor de la clase

self.param1 = param1

self.paramN = paramN

En general, habra tantos objetos como instancias se hayan creado de la clase

objeto1 = NombreDeLaClase(param1, …, paramN)

objeto2 = NombreDeLaClase(param1’, …, paramN’)

**En Python puedo crear un nuevo atributo a la instancia aunque no lo haya declarado desde la clase**. Esto es importante porque no funciona en otros lenguajes. Por ejemplo:

class ClaseDeEjemplo:

def \_\_init\_\_(self, parametro1:int):

self.parametro1 = parametro1

objeto:ClaseDeEjemplo = ClaseDeEjemplo(5) # objeto.parametro1 = 5

objeto.parametro2:int = 2 # Ahora objeto es una instancia de ClaseDeObjeto con un segundo atributo que no tendrían el resto de objetos de esa clase

Para proteger a los objetos de erratas o de declaraciones no deseadas fueras del constructor, utilizamos la instrucción *\_\_slots\_\_*¸esto fija los identificadores de atributos de instancia permitidos (PARA EVITAR LA CREACIÓN DE ATRIBUTOS EXCLUSIVOS DE UNA INSTANCIA)

class ClaseDeEjemplo:

\_\_slots\_\_ = [“parametro1”]

def \_\_init\_\_(self, parametro1:int):

self.parametro1 = parametro1

objeto:ClaseDeEjemplo = ClaseDeEjemplo(5) # objeto.parametro1 = 5

objeto.parametro1 = 4 # Error de atributo

En Python hay dos tipos de atributos, los **Atributos de clase** y los **Atributos de instancias**, un atributo de clase es uno que se refiere a todas las instancias. Los atributos de clase son, esencialmente, atributos que comparten ente todas las instancias de la misma clase. No es un atributo asociado a un objeto en específico (por ejemplo, polígono 1, polígono 2) sino que es un atributo asociado a TODOS los objetos (instancias) de una misma clase (no necesitaré entonces el *self* para nada).

class ClaseDeEjemplo:

atributo\_de\_clase: tipo\_atributo\_de\_clase = valor\_inicial\_parametro\_de\_clase

def \_\_init\_\_(self, parametro1:int):

self.atributo\_de\_instancia : tipo\_atributo\_de\_instancia = parametro1

ClaseDeEjemplo.parametro\_de\_clase += 1 #Ejemplo de uso de atributo de calse.

objeto:ClaseDeEjemplo = ClaseDeEjemplo(5) # objeto.parametro1 = 5

ClaseDeEjemplo.parametro\_de\_clase # ACCESO AL PARÁMETRO DE CLASE

Objeto.parametro\_de\_clase # ACCESO NO RECOMENDADO

**ÁMBITO DE LAS VARIABLES (LOCAL, GLOBAL)**

**Si dentro de una función quiero modificar una variable global, tengo que utilizar el término *global variable. No necesito utilizar el término global si dentro de la función solo quiero LEER la variablle decarada globalmente.***

variable\_declarada\_como\_global:tipo\_variable\_global = valor\_variable\_global

def funcion(params:tipos) -> tipo\_de\_retorno:

global variable\_declarada\_como\_global **# SI NO, ESTARÍA CREANDO UNA VARIABLE LOCAL CON ESTE NOMBRE**

variable\_declarada\_como\_global = NUEVO\_valor\_variable\_global

PROBLEMA DE EJEMPLO: Diseñe un programa que cree una clase Estudiante que acceda a variables globales como el nombre del planeta donde viven los estudiantes o el aire que respiran (la tierra no es una característica exclusiva de los estudiantes. Por ejemplo, una clase “Animales” también tendría que acceder al nombre del planeta donde habitan, lo mismo ocurre con el aire que respiran, otra clase podría acceder a qué tipo de aire utilizan).

# VARIABLES GLOBALES

PLANETA : str = “Tierra”

AIRE : str = “Oxígeno”

Class Estudiante:

def \_\_init\_\_(self, nombre:str, edad:int, altura:float):

self.\_nombre : str = nombre

self.\_edad : int = edad  
 self.\_altura : float = altura

def información\_global(self) -> str:

return f”{self.\_nombre} vive en {PLANETA} y respira {AIRE}”

PROBLEMA DE EJEMPLO: Diseñe un programa que incluya la clase estudiante, atributos y variables.

CIUDAD : str = “Murcia” # VARIABLE GLOBAL

class Estudiante:

numero\_estudiantes:int = 0 # ATRIBUTO DE CLASE

def \_\_init\_\_(self, calificaciones:list[float]): # CONSTRUCTOR DE CLASE

self.\_calificaciones = calificaciones # ATRIBUTO DE LA INSTANCIA

Estudiante.numero\_estudiantes += 1 # MODIFICA EL ATRIBUTO DE CLASE

def calificacion\_media(self) -> float: # MÉTODO DE LA CLASE

sum:float = 0

for calificacion in range(0, len(self.\_calificaciones)):

sum += self.\_calificaciones[calificacion]

return sum / len(self.\_calificaciones)

estudiante : Estudiante = Estudiante([7, 5, 10]) # OBJETO DE LA CLASE ESTUDIANTE

estudiante.calificaciones # LLAMADA DEL ATRIBUTO DE LA INSTANCIA

Estudiante.numero\_estudiantes # LLAMADA AL ATRIBUTO DE LA CLASE

estudiante.numero\_estudiantes # EVITAR LLAMADA DE ATRIBUTO DE CLASE DESDE LA INSTANCIA DEL MÉTODO

**En Python los parámetros de una fución se reconocen solo a nivel local (fuera de la función no se reconocen) pero se pasan por referencia, es decir la función puede modificar las posiciones de memoria originales. Como solo se pueden modificar los tipos de datos mutables, las funciones pueden modificar directamente (aliasing funcional):**

* **Listas**
* **Conjuntos**
* **Diccionarios**
* **Objetos**

**MÉTODOS**

Los métodos representan el COMPORTAMIENTO asociado a los obbjetos. Está asociado con una acción que puede realizar un objeto. Hay tres tipos de métodos:

* Métodos de instancia (uno por cada objeto) 🡪 Tienen self como primer parámetro
* Métodos de clase (uno para toda la clase y común a todos los objetos) 🡪 Tienen el decorador *@classmethod*
* Métodos estáticos (independientes de clases y objetos)

class Rueda:

@classmethod # MÉTODO DE CLASE

def descripcion(cls) -> str:

return “Una rueda es un elemento circular que gira sobre un eje”

def \_\_init\_\_(self, radio:float): # CONSTRUCTOR (MÉTODO INST).

self.\_radio : float = radio

0tro ejemplo (más útil) de métodos de clase sería:

class Estudiante:

numero\_estudiantes:int = 0

@classmethod

def get\_numero\_estudiantes(cls) -> int:

return cls.numero\_estudiantes

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

self.\_nombre = nombre

**Los métodos de clase (@classmethod) PUEDEN acceder a las variables de clase pero NO a las variables de instancia. (Sí a número de estudiantes, pero no al nombre del estudiante)**

**MÉTODOS ESTÁTICOS**

Los métodos estáticos (@staticmethod) no están asociados ni a una clase ni a un objeto y NO PUEDEN acceder ni a las variables de clase ni a las variables de isntancia. ESENCIALMENTE SON COMO FUNCIONES QUE AGRUPAMOS (POR ORDEN) DENTRO DE CLASES.

class Coche:

@staticmethod

def convertir\_a\_galon(litros:float) -> float: # EJEMPLO DE MÉTODO ESTÁTICO

return litros \* 0.264172

**MÉTODOS MÁGICOS**

* **\_\_str\_\_** : Representación de los objetos cuando haga *print(objeto)*
* **\_\_lt\_\_:** Less than (<)
* **\_\_le\_\_:** Less or equal than (<=)
* **\_\_gt\_\_:** Greater than (>)
* **\_\_ge\_\_:** Greater or equal than (>=)
* **\_\_eq\_\_:** Equal (==) **🡪** No confundir con *is* (True sii objeto1 y objeto2 son el mismo objeto). Por defecto *\_\_eq\_\_ es lo mismo que is*
* **\_\_ne\_\_:** Not equal (not =)

PROBEMA 1. Creación de clases. Crea una clase en Python que represente un Polígono, dentro de un fichero llamado clases\_geometria.py. En este primer ejercicio, el constructor (método \_\_init\_\_ ) no recibirá ningún parámetro de inicialización (salvo el self).

La clase debe contener los siguientes atributos de instancia:

* numero\_lados: de tipo entero, y que representa el número de lados que tiene el polígono. Se establecerá a 3.
* color: de tipo string, y que representa el color del polígono. Se establece a “BLANCO”.
* forma: este atributo será de tipo enumerado (de string) llamado TipoForma (recuerda que para ello se crea una clase class TipoForma(Enum)). El enumerado TipoForma tendrá tres opciones posibles (CONVEXO, CONCAVO o COMPLEJO) cuyos valore será 1, 2 o 3, respectivamente. El atributo se establecerá a CONVEXO en el momento de la construcción del objeto.
* relación\_lados: este atributo será de tipo enumerado (de string) llamado TipoRelacionLados (para ello, definir class TipoRelacionLados(Enum)). Sólo lo podría tener uno de los siguientes valores: REGULAR o IRREGULAR (como en el caso anterior, cada opción tendrá un entero asociado). El atributo se establecerá a REGULAR

Crea un objeto de dicha clase, llamado mi\_poligono, y muestra por pantalla sus cuatro atributos desde el main, accediendo directamente a los atributos de la instancia (RECUERDA: ¡esto último no es una buena práctica, por eso PyCharm lanza advertencias!).

from enum import Enum

class TipoForma(Enum): # **EN LOS ENUMERADOS NO HACE FALTA TIPAR :int**

CONVEXO = 1

CONCAVO = 2

COMPLEJO = 3

class TipoRelacionLados(Enum):

REGULAR:int = 1

IRREGURAL:int = 2

class Poligono:

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

self.\_numero\_lados:int = 3

self.\_color:str = "BLANCO"

self.\_forma:TipoForma = TipoForma.CONVEXO

self.\_relacion\_lados:TipoRelacionLados = TipoRelacionLados.REGULAR

if \_\_name\_\_ == "\_\_main\_\_":

pol1:Poligono = Poligono()

print(f”El polígono pol1 tiene {pol1.\_numero\_de\_lados} lados.”) **# MALA PRÁCTICA DESDE LA FUNCIÓN MAIN**

print(f”El polígono pol1 tiene forma {pol1.\_forma.name}”) **# TAMBIÉN ES MALA PRÁCTICA**

PROBLEMA 2. Inicialización de la clase. Modificar el constructor (método \_\_init\_\_ ) de la clase para que reciba como parámetros.

from enum import Enum

class TipoForma(Enum):

CONVEXO = 1

CONCAVO = 2

COMPLEJO = 3

class TipoRelacionLados(Enum):

REGULAR:int = 1

IRREGURAL:int = 2

class Poligono:

def \_\_init\_\_(self, numero\_lados:int, color:str, forma:TipoForma, relacion\_lados:TipoRelacionLados):

self.\_numero\_lados:int = numero\_lados

self.\_color:str = color

self.\_forma:TipoForma = forma

self.\_relacion\_lados:TipoRelacionLados = relacion\_lados

if \_\_name\_\_ == "\_\_main\_\_":

pol1:Poligono = Poligono(5, "VERDE", TipoForma.CONVEXO, TipoRelacionLados.IRREGURAL)

pol2:Poligono = Poligono(8, "AZUL", TipoForma.CONCAVO, TipoRelacionLados.REGULAR)

print(f"POL 1 {pol1.\_numero\_lados} lados")

print(f"POL 2 {pol2.\_numero\_lados} lados")

print(f”POL 1 tipo de relaccion de lados: {pol1.\_relacion\_lados.name}”)

print(f”POL 2 tipo de relaccion de lados: {pol2.\_relacion\_lados.name}”)

PROBLEMA 2’: Modifique la clase para que el numero de lados sea un parámetro opcional con valor tres por defecto e incluye un nuevo color que sea opcional con valor blanco (string) por defecto.

from enum import Enum

class TipoForma(Enum):

CONVEXO = 1

CONCAVO = 2

COMPLEJO = 3

class TipoRelacionLados(Enum):

REGULAR:int = 1

IRREGURAL:int = 2

class Poligono:

def \_\_init\_\_(self, forma:TipoForma,relacion\_lados:TipoRelacionLados, numero\_lados:int=3,color:str=”BLANCO”):

self.\_numero\_lados:int = numero\_lados

self.\_color:str = color

self.\_forma:TipoForma = forma

self.\_relacion\_lados:TipoRelacionLados = relacion\_lados

if \_\_name\_\_ == "\_\_main\_\_":

pol:Poligono = Poligono(TipoForma.CONVEXO, TipoREelacionLados.REGULAR)

print(f”Poligono color: {pol.\_color}”) # -> blanco

PROBLEMA 3. Métodos adicionales. Añadir los siguientes métodos de instancia a la clase Polígono:

* Un método que permita obtener el número de lados del polígono.
* Un método que permita obtener el color del polígono.
* Un método que permita obtener la forma del polígono.
* Un método que permita obtener la relación entre los lados del polígono. Un método llamado establecer\_numero\_lados que permita modificar el número de lados del polígono. método recibirá un parámetro con los lados, que debería ser mayor o igual que 3, devolviendo la cadena "OK, se han establecido X lados en el polígono", siendo X el parámetro introducido. Si es menor que 3, el método devolverá la cadena de texto “ERROR: el número de lados de un polígono no puede ser menor de 3".
* Implementar el método mágico \_\_str\_\_ en la clase Polígono, que permita mostrar por pantalla el estado del objeto. Es decir, el valor de todos sus atributos en el momento de su invocación. Este método debe utilizar los cuatro métodos anteriores para evitar acceder directamente a los atributos de la clase.

Tras la creación del objeto mi\_poligono, establece ahora que el polígono definido tiene 4 lados a través del primer método creado. Evita ahora el acceso directo a los atributos haciendo uso de los métodos de consulta de atributos.

from enum import Enum  
class TipoForma(Enum):  
 CONVEXO = 1  
 CONCAVO = 2  
 COMPLEJO = 3  
  
class TipoRelacionLados(Enum):  
 REGULAR = 1  
 IRREGURAL = 2  
  
class Poligono:  
 def \_\_init\_\_(self, numero\_lados:int, color:str, forma:TipoForma, relacion\_lados:TipoRelacionLados):  
 self.\_numero\_lados:int = numero\_lados  
 self.\_color:str = color  
 self.\_forma:TipoForma = forma  
 self.\_relacion\_lados:TipoRelacionLados = relacion\_lados  
  
 def \_\_str\_\_(self) -> str:  
 return f"Lados {self.\_numero\_lados}, color {self.\_color}, forma {self.\_forma.name}, relación {self.\_relacion\_lados.name}"  
  
 def obtener\_numero\_lados(self) -> int:  
 return self.\_numero\_lados  
  
 def obtener\_color(self) -> str:  
 return self.\_color  
  
 def obtener\_forma(self) -> TipoForma:  
 return self.\_forma  
  
 def obtener\_relacion\_lados(self) -> TipoRelacionLados:  
 return self.\_relacion\_lados  
  
 def modificar\_numero\_lados(self, nuevo\_numero\_lados:int) -> str:

**# LOS MÉTODOS PARA MODIFICAR ATRIBUTOS SON ÚTILES PARA CONTROLAR LOS ERRORES ANTES DE LA ASIGNACIÓN**  
 if nuevo\_numero\_lados < 3:  
 return "Error: el número de lados ha de ser mayor o igual que 3"  
 self.\_numero\_lados = nuevo\_numero\_lados

**# AUNQUE EN LA CLASE SIEMPRE PUEDO ACCEDER AL ATRIBUTO, ES MEJOR IDEA CONSULTAR EL MÉTODO** return f"Ok, se han establecido { self.obtener\_numero\_lados() } en el polígono"   
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pol1:Poligono = Poligono(5, "VERDE", TipoForma.CONVEXO, TipoRelacionLados.IRREGURAL)  
 print(pol1)  
 pol1.modificar\_numero\_lados(10)  
 print(pol1)

PROBLEMA 4. Crea un atributo de clase que permita obtener el número de objetos que hay de tipo polígono. Para ello añadir el atributo de clase llamado número de polígonos (inicializado a cero) y realizar las modificaciones que consideremos necesarias.

(...)  
class Poligono:  
 numero\_de\_poligonos : int = 0  
   
 def \_\_init\_\_(self, parámetros):  
 (...)  
 numero\_de\_poligonos += 1  
  
 (...)

if \_\_name\_\_ == “\_\_main\_\_”:

pol1 = Poligono(…)

pol2 = Poligono(…)

pol3 = Poligono(…)

pol4 = Poligono(…)

print(f”Número de polígonos: { Poligono.numero\_poligonos }”) **# MALA PRÁCTICA vs. PROBLEMA 4’**

**# HACER pol1.numero\_poligonos = 5 NO CREA UN ATRIBUTO DE INSTANCIA, SINO QUE MODIFICA EL ATRIBUTO DE**

**# CLASE. ESTO NO SE DEBE HACER (NO DEBEMOS CONSULTAR NI MODIFICAR EL NÚMERO DE POLÍGONOS A TRAVÉS DE**

**# INSTANCIAS**

PROBLEMA 4’ Modifique la clase para obtener el número de polígonos creando así un método de clase.

(...)  
class Poligono:  
 numero\_de\_poligonos : int = 0  
   
 def \_\_init\_\_(self, parámetros):  
 (...)  
 Poligono.numero\_de\_poligonos += 1

@classmethod

def obtener\_numero\_poligonos(cls) -> int:

return cls.numero\_poligonos  
  
 (...)

if \_\_name\_\_ == “\_\_main\_\_”:

pol1 = Poligono(…)

pol2 = Poligono(…)

pol3 = Poligono(…)

pol4 = Poligono(…)

print(f”Número de polígonos: { Poligono.obtener\_numero\_poligonos() }”)

PROBLEMA 5. Crea una clase dentro del fichero que represente un punto definida por los siguientes atributos: coordenada x, coordenada y. Adicionalmente crea métodos para obtener las coordenadas, para obtener un valor numérico asignado al cuadrante. Crea cuatro puntos y muestra a qué cuadrantes pertenecen.

from enum import Enum  
  
class Cuadrante(Enum):  
 EJE = 0  
 PRIMER\_CUADRANTE = 1  
 SEGUNDO\_CUADRANTE = 2  
 TERCER\_CUADRANTE = 3  
 CUARTO\_CUADRANTE = 4  
  
class Punto:  
  
 def \_\_init\_\_(self, x:float, y:float):  
 self.\_coordenada\_x:float = x  
 self.\_coordenada\_y:float = y  
  
 def \_\_str\_\_(self):  
 return f"({self.obtener\_coordenada\_x()}, {self.obtener\_coordenada\_y()}) CUAD {self.obtenerCuadrante().name}"  
 def obtener\_coordenada\_x(self) -> float:  
 return self.\_coordenada\_x  
  
 def obtener\_coordenada\_y(self) -> float:  
 return self.\_coordenada\_y  
  
 def obtenerCuadrante(self) -> Cuadrante:  
 if self.obtener\_coordenada\_x() == 0 or self.obtener\_coordenada\_y() == 0:  
 return Cuadrante.EJE  
 elif self.obtener\_coordenada\_x() > 0:  
 if self.obtener\_coordenada\_y() > 0:  
 return Cuadrante.PRIMER\_CUADRANTE  
 return Cuadrante.CUARTO\_CUADRANTE  
 else:  
 if self.obtener\_coordenada\_y() > 0:  
 return Cuadrante.SEGUNDO\_CUADRANTE  
 return Cuadrante.TERCER\_CUADRANTE  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 A:Punto = Punto(1,1)  
 B:Punto = Punto(-1, 1)  
 C:Punto = Punto(-1, -1)  
 D:Punto = Punto(1, -1)  
 E:Punto = Punto(0, 1)  
 F:Punto = Punto(1, 0)  
  
 print(f"{A}, {B}, {C}, {D}, {E}, {F}")

PROBLEMA 6. Crea una clase en el fichero que represente una línea. La clase debe contener dos atributos: punto inicio, punto fin. Adicionalmente crea métodos para obtener los puntos de inicio y de fin, para calcular la longitud así como para mostrar los puntos de inicio y de fín.

import math  
  
class Linea:  
  
 def \_\_init\_\_(self, punto\_inicio:Punto, punto\_fin:Punto) -> None:  
 self.\_punto\_inicio:Punto = punto\_inicio  
 self.\_punto\_fin:Punto = punto\_fin  
  
 def \_\_str\_\_(self) -> str:  
 return f"LÍNEA ({self.obtener\_punto\_fin()}, {self.obtener\_punto\_fin()}) LONGITUD {self.calcular\_longitud()}."  
 def obtener\_punto\_inicio(self) -> Punto:  
 return self.\_punto\_inicio  
  
 def obtener\_punto\_fin(self) -> Punto:  
 return self.\_punto\_fin  
  
 def calcular\_longitud(self) -> float:  
 x1:float = self.obtener\_punto\_inicio().obtener\_coordenada\_x()  
 y1:float = self.obtener\_punto\_inicio().obtener\_coordenada\_y()  
 x2:float = self.obtener\_punto\_fin().obtener\_coordenada\_x()  
 y2:float = self.obtener\_punto\_fin().obtener\_coordenada\_y()  
 return math.sqrt(pow(x2-x1, 2) + pow(y2-y1, 2))  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 origen:Punto = Punto(0,0)  
 linea:Linea = Linea(origen, Punto(1,1))  
 print(linea)

PROBLEMA 7. Amplía la clase polígono con un atributo “lados” que contenga elementos de tipo línea y que contendrá cada una de las líneas que sirve de lado del poligono. Añade los métodos:

* Calcular perímetro
* Establecer lados: que recibirá como parámetro una lista de líneas que representan los nuevos lados del polígono. Debe comprobar que el número de líneas introducidas equivale al número de lados, retornando True en caso de que se pueda establecer la lista de líneas y False en caso contrario

En el método main añade al objeto poligono una lista con cuatro líneas que formen un cuadrado y actualiza el método mágico \_\_str\_\_ de la clase Poligono para que también muestre el cálculo del perímetro del mismo.

from enum import Enum  
import math  
  
class Cuadrante(Enum):  
 EJE = 0  
 PRIMER\_CUADRANTE = 1  
 SEGUNDO\_CUADRANTE = 2  
 TERCER\_CUADRANTE = 3  
 CUARTO\_CUADRANTE = 4  
  
class Punto:  
  
 def \_\_init\_\_(self, x:float, y:float):  
 self.\_coordenada\_x:float = x  
 self.\_coordenada\_y:float = y  
  
 def \_\_str\_\_(self):  
 return f"({self.obtener\_coordenada\_x()}, {self.obtener\_coordenada\_y()}) CUAD {self.obtenerCuadrante().name}"  
 def obtener\_coordenada\_x(self) -> float:  
 return self.\_coordenada\_x  
  
 def obtener\_coordenada\_y(self) -> float:  
 return self.\_coordenada\_y  
  
 def obtenerCuadrante(self) -> Cuadrante:  
 if self.obtener\_coordenada\_x() == 0 or self.obtener\_coordenada\_y() == 0:  
 return Cuadrante.EJE  
 elif self.obtener\_coordenada\_x() > 0:  
 if self.obtener\_coordenada\_y() > 0:  
 return Cuadrante.PRIMER\_CUADRANTE  
 return Cuadrante.CUARTO\_CUADRANTE  
 else:  
 if self.obtener\_coordenada\_y() > 0:  
 return Cuadrante.SEGUNDO\_CUADRANTE  
 return Cuadrante.TERCER\_CUADRANTE  
  
class Linea:  
  
 def \_\_init\_\_(self, punto\_inicio:Punto, punto\_fin:Punto) -> None:  
 self.\_punto\_inicio:Punto = punto\_inicio  
 self.\_punto\_fin:Punto = punto\_fin  
  
 def \_\_str\_\_(self) -> str:  
 return f"LÍNEA ({self.obtener\_punto\_fin()}, {self.obtener\_punto\_fin()}) LONGITUD {self.calcular\_longitud()}."  
 def obtener\_punto\_inicio(self) -> Punto:  
 return self.\_punto\_inicio  
  
 def obtener\_punto\_fin(self) -> Punto:  
 return self.\_punto\_fin  
  
 def calcular\_longitud(self) -> float:  
 x1:float = self.obtener\_punto\_inicio().obtener\_coordenada\_x()  
 y1:float = self.obtener\_punto\_inicio().obtener\_coordenada\_y()  
 x2:float = self.obtener\_punto\_fin().obtener\_coordenada\_x()  
 y2:float = self.obtener\_punto\_fin().obtener\_coordenada\_y()  
 return math.sqrt(pow(x2-x1, 2) + pow(y2-y1, 2))  
  
  
from enum import Enum  
class TipoForma(Enum):  
 CONVEXO = 1  
 CONCAVO = 2  
 COMPLEJO = 3  
  
class TipoRelacionLados(Enum):  
 REGULAR = 1  
 IRREGURAL = 2  
  
class Poligono:  
 def \_\_init\_\_(self, numero\_lados:int, color:str, forma:TipoForma, relacion\_lados:TipoRelacionLados):  
 self.\_numero\_lados:int = numero\_lados  
 self.\_color:str = color  
 self.\_forma:TipoForma = forma  
 self.\_relacion\_lados:TipoRelacionLados = relacion\_lados  
  
 self.\_lados:list[Linea] = list  
  
 def \_\_str\_\_(self) -> str:  
 return f"Lados {self.\_numero\_lados}, color {self.\_color}, forma {self.\_forma.name}, relación {self.\_relacion\_lados.name}, perímetro{self.calcular\_perimetro()}"  
  
 def obtener\_numero\_lados(self) -> int:  
 return self.\_numero\_lados  
  
 def obtener\_color(self) -> str:  
 return self.\_color  
  
 def obtener\_forma(self) -> TipoForma:  
 return self.\_forma  
  
 def obtener\_relacion\_lados(self) -> TipoRelacionLados:  
 return self.\_relacion\_lados  
  
 def modificar\_numero\_lados(self, nuevo\_numero\_lados:int) -> str:  
# LOS MÉTODOS PARA MODIFICAR ATRIBUTOS SON ÚTILES PARA CONTROLAR LOS ERRORES ANTES DE LA ASIGNACIÓN  
 if nuevo\_numero\_lados < 3:  
 return "Error: el número de lados ha de ser mayor o igual que 3"  
 self.\_numero\_lados = nuevo\_numero\_lados  
# AUNQUE EN LA CLASE SIEMPRE PUEDO ACCEDER AL ATRIBUTO, ES MEJOR IDEA CONSULTAR EL MÉ***TODO*** return f"Ok, se han establecido { self.obtener\_numero\_lados() } en el polígono"  
  
 def calcular\_perimetro(self) -> float:  
 perimetro:float = 0  
 for lado in self.\_lados:  
 perimetro+=lado.calcular\_longitud()  
 return perimetro  
  
 def establecer\_lados(self, lineas:list[Linea]) -> bool:  
 if len(lineas) != self.obtener\_numero\_lados():  
 print("No se han podido establecer las líneas como lados del polígono.")  
 return False  
 print(f"Lineas establecidas. El polígono tiene {self.obtener\_numero\_lados}")  
 self.\_lados = lineas

**SESIÓN 3: ESTRUCTURA DE PROYECTOS, VISIBILIDAD, SOBRECARGA DE MÉTODOS, UML**

Para estudiar la estructura de un proyecto en Python vamos a simular un juego (como el primer proyecto de las sesiones de teoría)

* Archivo .py 🡪 Módulo (empaqueta la funcionalidad de una clase) ej: mascotas.py, personajes.py Por regla general cada clase debe tener un módulo (a menos que varias clases estén íntimamente relaccionadas). Un módulo debe ser
  + Cohesivo: el propósito ha de estar bien definido
  + Poco acoplado: Lo ideal es que las clases sean independientes entre sí, es mejor que no haya muchas referencias entre clases en diferentes módulos
* Agrupación de módulos similares 🡪 Paquete ej: entes

Entre módulos y paquetes podemos importar archivos. Por ejemplo, supongamos que tenemos la siguiente estructura

calculadora/  
 \_\_init\_\_.py  
 calculadora.py  
 operaciones/   
 \_\_init\_\_.py  
 suma.py  
 resta.py  
 mult.py  
 division.py

Para importar paquetes y módulos utilizamos la siguiente notación.

**# Usa un modulo del mismo paquete**

**import unmodulo**

**# Usa un modulo de otro paquete**

**import Paquete.unmodulo**

**# Usa un modulo de un paquete interno (paquete dentro de paquete)**

**import Paquete . otropaquete . otromodulo**

**# Usa la función del mismo paquete**

**unmodulo.funcA ()**

**# Usa la función de otro paquete**

**Paquete.unmodulo.funcB ()**

**# Usa la función del modulo del paquete interno**

**Paquete.otropaquete.otromodulo.funcC ()**

También puedo importar paquetes con nombres completos:

import math as m

print(m.pi) # -> 3.14159…

O incluso importar SOLO lo que me interesa del paquete:

from math import pi, cos

**VISIBILIDAD**

En muchos casos interesa que desde el main no se pueda acceder a todos los parámetros de una clase. Tenemos entonces tres tipos de visibilidad. No debemos permitir que desde un objeto (o desde el mismo main) se modifiquen atributos de otro objeto directamente (podría inducir a muchos errores). Para modificar la visibilidad de los atributos y los métodos utilizamos los MODIFICADORES DE ACCESO:

* Pública 🡪 Accesible desde el main sin problema
* Protegida 🡪 Solo va a ser accesible desde la clase Y DESDE LAS SUBCLASES (herencia) (NO DESDE FUERA DE LA CLASE)
* Privada 🡪 Solo se va a obtener la información desde DENTRO DE ESTA CLASE

main:

objeto1.atributo\_publico # FORMA DE ACCEDER A UN MIEMBRO PÚBLIC

objeto1.\_atributo\_protegido # NO DEBERÍA PODER, PYTHON SE LO PASA POR EL FORRO (GITANADA)

objeto1.\_\_atributo\_privado # NO PUEDO. La única forma de gitanearlo es p1. \_ Clase \_ \_ atributo

A Python se la pelan los modificadores de acceso. SI podemos acceder desde fuera a atributos protegidos o privados (es una auténtica gitanada)

Al conjunto de TODOS LOS MÉTODOS PÚBLICOS DE UNA CLASE se le llama **contrato público de la clase**,

Por defecto siempre crearemos atributos protegidos.

Desde el main, no es buena idea (ni buena práctica) acceder a los atributos directamente. A estos métodos se les llama get y set. Lo ideal es definir los mínimos métodos get y set, para que desde fuera solo se pueda hacer uso de las cosas estrictamente imprescindibles.

def get\_variable(self) -> tipo\_variable:

return self.\_variable

Método para asignar el valor a una variable (Por convenio, set)

def set\_variable(self, valor:tipo\_valor) -> None:

COMPROBACIONES CON valor

self.\_variable = valor

**SOBRE CARGA DE MÉTODOS Y SOBRE CARGA DEL CONSTRUCTOR**

Python no permite la sobre carga de métodos (diferentes versiones del mismo método). Tenemos entonces que utilizar funciones con parámetros opcionales

def f(p1:float, p2:float = None) -> None

print(f“Función con {p1} y {p2}”)

También podemos tener sobrecarga utilizando *\*args* y \*\*kwargs.

Si no pongo método \_\_init\_\_ en una clase, por defecto python creará un objeto SIN ATRIBUTOS (un poco raro, mejor no hacer).

**UML**

UML es un lenguaje estándar para modelar sistemas orientados a objetos. Expresa las clases y la relacción entre sus clases.

PRIMERA CAJA: nombre de la clase

SEGUNDA CAJA: atributos

Atributos de clase se subrayan

* Atributo público: **+**
* Atributo protegido: **#**
* Atributo privado: **-**

TERCERA CAJA: métodos

Métodos de clase se subrayan

* Método público: **+**
* Método protegido: **#**
* Método privado: **-**

En estos diagramas no solemos poner los métodos get y set.

Diagrama

El contenido generado por IA puede ser incorrecto.

PROBLEMA 1. Sobrecarga de métodos. Trabajaremos con la clase Polígono para practicar la sobrecarga de constructores y métodos. Sigue los siguientes pasos: a) Actualmente, el constructor de la clase Polígono recibe dos parámetros obligatorios (forma y relación\_lados) y dos parámetros opcionales (numero\_lados y color). Transforma el constructor para que se puedan crear polígonos de las siguientes formas: Especificado sólo el número de lados. Especificando el número de lados y la forma. Especificando el número de lados, la forma y la relación de lados. Especificando el número de lados, la forma, la relación de lados y el color

# CONSTRUCTOR SOBRE CARGADO  
def \_\_init\_\_(self,   
 numero\_lados:int,  
 forma:TipoForma = None,  
 relacion\_lados:TipoRelacionLados = None,  
 color:str = None):  
 self.\_numero\_lados:int = numero\_lados  
 self.\_color:str = color  
 self.\_forma:TipoForma = forma  
 self.\_relacion\_lados:TipoRelacionLados = relacion\_lados  
  
 self.\_lados:list[Linea] = list()  
   
   
# COMO CONSECUENCIA DIRECTA HAY QUE MODIFICAR EL MÉ***TODO STR***def \_\_str\_\_(self) -> str:  
 msg:str = f"Lados {self.obtener\_numero\_lados()} "  
 if self.obtener\_forma(): # equivalente a is not None  
 msg += f"Forma: {self.obtener\_forma().name}"  
 if self.obtener\_color():  
 msg += f"Color: {self.obtener\_color().name} "  
 if self.obtener\_relacion\_lados():  
 msg += f"Relacion lados: {self.obtener\_relacion\_lados().name} "  
 return msg

PROBLEMA 1’: Crea un método llamado escalar\_poligono capaz de aumentar el número de lados de un polígono, devolviendo si ha sido posible el escalado. Podremos usarlo de dos formas diferentes: La primera versión recibe un parámetro factor, que multiplica el número de lados del polígono por ese valor. La segunda versión recibe el parámetro ajuste\_lados que sumará (o restará) un número determinado de lados. 9 ¿Podríamos apoyarnos en algún método ya existente en la clase Polígono? Si es así, ¿sería conveniente realizar alguna modificación?

def modificar\_numero\_lados(self, nuevo\_numero\_lados: int) -> bool:  
 # LOS MÉTODOS PARA MODIFICAR ATRIBUTOS SON ÚTILES PARA CONTROLAR LOS ERRORES ANTES DE LA ASIGNACIÓN  
 if nuevo\_numero\_lados < 3:  
 print("No se han podido establecer el número de lados")  
 return False  
 self.\_numero\_lados = nuevo\_numero\_lados  
 # AUNQUE EN LA CLASE SIEMPRE PUEDO ACCEDER AL ATRIBUTO, ES MEJOR IDEA CONSULTAR EL MÉ***TODO*** print(f"Ok, se han establecido {self.obtener\_numero\_lados()} en el polígono")  
 return True

( . . . )

def escalar(self, factor:int = None, ajuste\_lados:int = None) -> bool:  
 resultado:bool = False  
 # PRIMERO MULTIPLICA Y LUEGO SUMA  
   
 if factor:  
 resultado = self.modificar\_numero\_lados(self.obtener\_numero\_lados() \* factor)  
 if ajuste\_lados:  
 resultado = self.modificar\_numero\_lados(self.obtener\_numero\_lados() + ajuste\_lados)  
 return resultado

PROBLEMA 2. Visibilidad de atributos en este ejercicio continuaremos trabajando con la clase polígono. Para profundizar en los conceptos de visibilidad de atributos y el uso de modificadores de acceso.

1. Revisa la visibilidad actual de los atributos y determina cuales deberían cambiar y cuales mantener su visibilidad actual
2. Añade un atributo privado contraseña base de datos que será exclusivamente accedido por la propia clase para dotar de protección extra.

self.\_\_contrasenya:str = contrasenya

PROBLEMA 3. Métodos Getter/Setter para acceder a los atributos. Para el tercer apartado, trabajaremos con getters y setters en el código de la clase Polígono. En particular, vamos a analizar si para cada atributo tiene sentido crear tanto un getter como un setter (ahora mismo con visibilidad pública), o si hay casos donde solo uno sería suficiente, o ninguno de ellos.

Recuerda cambiar el nombre de los métodos obtener() y establecer() ya existentes a get() y set() por convención. Además, amplía el main para que incluya la siguiente funcionalidad:

1. Tras crear y escalar ambos polígonos (ya implementado en el main en el apartado anterior), modifica el primer polígono para que tenga color AZUL, tenga forma convexa y sea un polígono regular. Haz uso de los métodos setter previamente definidos. ¿Podríamos modificar directamente al atributo contraseña\_base\_datos?

def get\_numero\_lados(self) -> int:

return self.\_numero\_lados

def get\_color(self) -> str:

return self.\_color

def get\_forma(self) -> TipoForma:

return self.\_forma

def get\_relaccion\_lados(self) -> TipoRelacionLados:

return self.\_relacion\_lados

def set\_numero\_lados(self, nuevo\_numero\_lados:int) -> bool:

# LOS MÉTODOS PARA MODIFICAR ATRIBUTOS SON ÚTILES PARA CONTROLAR LOS ERRORES ANTES DE LA ASIGNACIÓN

if nuevo\_numero\_lados < 3:

print("Error: el número de lados ha de ser mayor o igual que 3")

return False

self.\_numero\_lados = nuevo\_numero\_lados

# AUNQUE EN LA CLASE SIEMPRE PUEDO ACCEDER AL ATRIBUTO, ES MEJOR IDEA CONSULTAR EL MÉTODO

print(f"Ok, se han establecido { self.obtener\_numero\_lados() } en el polígono")

return True

def set\_color(self, color:str) -> bool:

self.\_color = color

print(f"Se ha cambiado el color del polígono a {self.get\_color()}")

return True

def set\_forma(self, forma:TipoForma) -> bool:

self.\_forma = forma

print(f"Se ha cambiado la forma del polígono a {self.get\_forma().name}")

return True

def set\_relacion\_lados(self, relacion\_lados:TipoRelacionLados) -> bool:

self.\_relacion\_lados = relacion\_lados

print(f"Se ha cambiado la relación de lados del polígono a {self.get\_relaccion\_lados().name}")

return True

def set\_lados(self, lineas:list[Linea]) -> bool:

if len(lineas) != self.get\_numero\_lados():

print("No se han podido establecer las líneas como lados del polígono.")

return False

print(f"Lineas establecidas. El polígono tiene {self.obtener\_numero\_lados}")

self.\_lados = lineas

b) Consulta los atributos modificados del primer polígono de forma adecuada haciendo uso de métodos getter. ¿Podemos leer directamente al atributo contraseña\_base\_datos? c) Muestra por pantalla el contenido de ambos objetos mediante el uso del método mágico \_\_str\_\_.

print(mi\_poligono)  
mi\_poligono.set\_color("azul")  
mi\_poligono.set\_forma(TipoForma.CONVEXO)  
mi\_poligono.set\_relacion\_lados(TipoRelacionLados.REGULAR)  
print(mi\_poligono)

PROBLEMA 4. . Revisión de la visibilidad de los métodos. En este apartado, ajustaremos la visibilidad de los métodos en la clase Polígono, considerando su funcionalidad y uso. Considera ahora que sólo es posible modificar el número de lados a través del método escalar\_poligono():

def \_set\_numero\_lados(self, nuevo\_numero\_lados:int) -> bool:

a) Evalúa si los métodos get/set deben ser públicos, protegidos o privados, teniendo en cuenta el acceso externo que se espera para cada atributo. Haz los cambios que consideres.

b) Revisa la visibilidad de los métodos restantes de la clase Polígono (por ejemplo, los métodos de clase y otros métodos de instancia) y determina si deben ser públicos, protegidos o protegidos.

c) Define un método para cambiar la contraseña de un polígono desde el programa principal. Se devolverá verdadero cuando se realice con éxito, falso en caso contrario. No se puede repetir la contraseña que haya en ese momento y la nueva contraseña debe tener más de 8 caracteres.

def set\_contrasenya(self, contrasenya:str) -> bool:  
 if self.\_\_contrasenya != contrasenya and len(contrasenya) > 8:  
 self.\_\_contrasenya = contrasenya  
 print("Contraseña cambiada con éxito.")  
 return True  
 else:   
 print("Contraseñas diferentes, al menos ocho caracteres")  
 return False

PROBLEMA 5. Módulos y paquetes. Define tu propia estructura de proyecto usando módulos y paquetes, de forma que modularicemos lo que hasta ahora tenemos sólo en un módulo. Algunas consideraciones importantes:

a) En primer lugar, asegúrate que tienes un módulo .py para el main y un módulo por cada clase (o enumerado) de tu proyecto.

b) Recuerda que tenemos las clases Polígono, Punto y Línea definidas de la sesión anterior y que debemos incluir en nuestra estructura de proyecto.

c) Tras ello, ten en cuenta que además de la aplicación principal, se ofrece funcionalidad centrada en geometría para diferentes tipos de polígonos (podríamos tener diferentes tipos de polígonos como círculos, cuadrados o triángulos, entre otros muchos). Para definir nuestro proyecto, crea clases para Círculo, Cuadrado y Triángulo en módulos nuevos que estarán vacíos de código.

d) Tras crear la estructura de paquetes y módulos, escribe código en el módulo main.py para que el programa principal pueda hacer uso de las clases implementadas. En particular, debe ejecutar las siguientes acciones: Crea un polígono rojo de cuatro lados, convexo e irregular. Además, crea un segundo polígono de cinco lados con forma convexa (el resto de propiedades no se aportan para este polígono). Accede a algunas de las propiedades de los polígonos creados mediante el uso de métodos get. Modifica o establece algunas propiedades de los polígonos creados con métodos set.

mi\_poligono3:Poligono = Poligono(4, TipoForma.CONVEXO, TipoRelacionLados.REGULAR, "rojo")  
mi\_poligono4:Poligono = Poligono(5, TipoForma.CONVEXO)  
  
print(f"Ejemplo de consulta usando un método getter polígono 3, numero\_lados: {mi\_poligono3.get\_numero\_lados()}")  
print(f"Ejemplo de consulta usando un método getter polígono 3, forma: {mi\_poligono3.get\_forma().name}")  
print(f"Ejemplo de consulta usando un método getter polígono 3, relacion\_lados: {mi\_poligono3.get\_relaccion\_lados().name}")  
print(f"Ejemplo de consulta usando un método getter polígono 3, color: {mi\_poligono3.get\_color()}")  
  
mi\_poligono3.set\_color("verde")  
mi\_poligono4.set\_color("lila")  
mi\_poligono4.set\_forma(TipoForma.CONCAVO)  
  
print(mi\_poligono3)  
print(mi\_poligono4)

PROBLEMA 6. Mejora de las clases Punto y Línea. Modificar las clases Punto y Línea para mejorar los siguientes aspectos, previamente definidos en el ejercicio 5 como módulos de nuestro proyecto:

* Sobrecarga de métodos: Revisar si sería necesario aplicar sobrecarga de métodos a los constructores y métodos de ambas clases.
* Visibilidad de atributos: Comprobar la visibilidad actual de los atributos de ambas clases y modificarlas de forma conveniente.
* Métodos getter/setter: Definir los métodos get/set adecuados para cada clase, sustituyendo aquellos métodos de consulta y modificación previamente existentes en la sesión 2. Modificación del main: Actualiza el main para crear, consultar y modificar puntos y líneas, además de polígonos.

# EN LA CLASE LÍNEA:

def get\_punto\_inicio(self) -> Punto:

return self.\_punto\_inicio

def get\_punto\_fin(self) -> Punto:

return self.\_punto\_fin

def set\_punto\_inicio(self, punto:Punto) -> bool:  
 self.\_punto\_inicio = punto  
 print(f"Punto de inicio actualizado con éxito a {self.get\_punto\_inicio()}")  
 return True  
  
def set\_punto\_fin(self, punto:Punto) -> bool:  
 self.\_punto\_fin = punto  
 print(f"Punto de fin actualizado con éxito a {self.get\_punto\_fin()}")  
 return True

# EN LA CLASE PUNTO:

def get\_coordenada\_x(self) -> float:  
 return self.\_coordenada\_x  
  
def get\_coordenada\_y(self) -> float:  
 return self.\_coordenada\_y  
  
def get\_cuadrante(self) -> Cuadrante:  
 if self.get\_coordenada\_x() == 0 or self.get\_coordenada\_y() == 0:  
 return Cuadrante.EJE  
 elif self.get\_coordenada\_x() > 0:  
 if self.get\_coordenada\_y() > 0:  
 return Cuadrante.PRIMER\_CUADRANTE  
 return Cuadrante.CUARTO\_CUADRANTE  
 else:  
 if self.get\_coordenada\_y() > 0:  
 return Cuadrante.SEGUNDO\_CUADRANTE  
 return Cuadrante.TERCER\_CUADRANTE  
   
def set\_coordenada\_x(self, x:float) -> bool:  
 self.\_coordenada\_x = x   
 print(f"Coordenada X actualizada con éxito a {self.get\_coordenada\_x()}")  
 return True  
  
def set\_coordenada\_y(self, y:float) -> bool:  
 self.\_coordenada\_y = y   
 print(f"Coordenada Y actualizada con éxito a {self.get\_coordenada\_y()}")  
 return True

# EN EL MAIN AÑADIMOS ALGO COMO:

ejemplo\_punto:Punto = Punto(3,4)  
print(f"Ejemplo punto: {ejemplo\_punto}")  
ejemplo\_punto.set\_coordenada\_x(6)  
ejemplo\_punto.set\_coordenada\_y(-2)  
print(ejemplo\_punto)

**SESIÓN 4: RELACIONES DE CLASE, DELEGACIÓN**

Podemos establecer comunicaciones entre objetos de diferentes clases.

1. **RELACIONES DE ASOCIACIÓN**

Como clase, puedo tener atributos que sean instancias de otra clase. Suelen ser relaciones estables en el tiempo. Información que queremos mantener (desde ClaseA) durante mucho tiempo.

Por ejemplo, el listado de amigos de un personaje en un videojuego tendría que ser un atributo, queremos acceder a él durante todo el tiempo de ejecución del videojuego.

Es decir, relación de asociación 🡪 **uso continuado, el objeto es una instancia de la clase.**

class ClaseB:  
 ( . . . )

class ClaseA:

(. . .)

self.objeto\_claseB: ClaseB = ClaseB()

1. **RELACIONES DE AGREGACIÓN**

(has-a). Un tipo específico de relación de asociación es la relación de agregación. Cuando como clase me “asocio” algo (en el sentido de pertenencia) esta relación es una relación de agregación. Por ejemplo los hombres tienen mujeres y los niños tienen mascotas. Se asume que es una relación relativamente estable en el tiempo (tener una pertenencia no cambia mucho).

Es decir, relación de agregación 🡪 **uso continuado, sentido de pertenencia. Subtipo de asociación.**

1. **RELACIONES DE COMPOSICIÓN**

(part-of) Es el caso de asociación más estricto de todos. A nivel de composición, cuando se destruye una clase que tiene un atributo relacionado por composición, este atributo se destruye también con la case. La composición se refiere a atributos míos de tal forma que cuando yo desaparezca, ellos desapareceran. Por ejemplo, como humano tengo brazos. Estos brazos son atributos en relación de composición.

Por ejemplo, en un videojuego cada personaje podría tener un “collar” especial que le da super poderes. Si este collar es único, cuando uno de los personajes desaparezca, el collar desaparecería también. O si suponemos que las armas del videojuego son exclusivas del personaje que las crea (nadie sabe usar mi arma, yo no sé usar el arma del resto). Esta relación sería de composición. El atributo de la clase B suele crearse en el constructor de la clase A.

Es decir, relación de composición 🡪 **uso continuado, muy restrictiva. Cuando me destruyo, se destruye.**

1. **RELACIONES DE USO**

Como clase, puedo también tener un método que reciba un parámetro para modificar un objeto de otra clase. En general implementamos este tipo de relación cuando queremos que la “interacción” con el objeto de clase B sea breve. (Solo recibe un parámetro, lo usa brevemente y esta será la relación con el objeto de la segunda clase.

Por ejemplo, si queremos usar un objeto en un videojuego, pero no guardarlo en nuestro inventario, deberíamos utilizar este segundo tipo de relación entre objetos.

Es decir, relación de uso 🡪 **uso breve, esporádico, el objeto se recibe como parámetro.**

class ClaseB:

def usar():

(. . .)

class ClaseA:

(. . .)

def metodo\_que\_usa\_claseB(self, objeto\_clase\_B:ClaseB): # BREVE

objeto\_clase\_B.usar()

El estándar UML incluye nomenclatura para las relaciones de clase.

* Relación de Asociación bidirecional 🡪 Línea.
* Relación de Asociación unidireccional 🡪 Línea dirigida (flecha oscura).
* Relación de Agregación 🡪 Línea con rombo blanco en el extremo del poosedor.
* Relación de Composición 🡪 Línea con rombo negro en el extremo del poosedor.
* Relación de uso 🡪 Línea discontínua

Diagrama

El contenido generado por IA puede ser incorrecto.

Diagrama

El contenido generado por IA puede ser incorrecto.

Además, podemos definir **cardinalidad** como el número de personas como el número de relaciones que se establecen entre clases. Por ejemplo, una tarjeta de crédito está asociada a una única persona, pero una persona puede estár asociadas a N tarjetas de crédito.

Diagrama

El contenido generado por IA puede ser incorrecto.

**DELEGACIÓN**

Muchas veces tenemos métodos que llaman simultáneamente a otros métodos de una clase diferente. Este comportamiento se llama delegación de funcionalidad. Tenemos que EVITAR las “clases Dios”, clases demasiado grandes que abarcan demasiadas funcionalidades. Por ejemplo, en la clase Personaje de un videojuego, no deberíamos conocer la funcionalidad de las mascotas o de sus armas, sino delegarlas en la clase Mascota, Arma.

class claseB:

def metodoB():

( MÉTODO B)

class ClaseA:

( . . .)

def metodo\_que\_llama\_al\_metodoB(self):

( COMPROBACIONES PREVIAS A LA LLAMADA DEL MÉTODO B )

self.objeto\_claseB.metodoB()

**COPIAR OBJETOS**

Imaginemos que dado objeto1 queremos crear una copia en tiempo real de objeto2. Esto se denomina “clonar un objeto”. Supongamos que objeto1 tiene relaciones entre clases de cualquier tipo (atributos que son listas, atributos que son de clase). Entonces al copiar este objeto no copio una copia idéntica de los atributos, sino una referencia de memoria.

1. **Copia superficial**: los tipos inmutables se copian sin problema pero los tipos mutables se copian la dirección de memoria (ALIASING). Por ejemplo, si objeto1 tiene un coche “COCHE 1”, al copiar objeto1 en objeto2, compartirían el coche, es decir, el coche de objeto2 sería “COCHE 1”

import copy

objeto2 = copy.copy(objeto1) # ¡TIENE ALIASING!

1. **Copia profunda** Se generan dos objetos completamente independientes (sin aliasing) que puede ser muy compleja.

import copy

objeto2 = copy.deepcopy(objeto2) # OBJETOS ABSOLUTAMENTE INDEPENDIENTES, SIN ALIASING

**Para que una clase tenga versión de *copy* ý *deepcopy*, definimos los métodos mágicos *\_\_copy\_\_( )* ý *\_\_deepcopy\_\_( )***

PROBLEMA 1. Relación de uso.

a) Implementa el método calcular\_distancia\_puntos que calcule la distancia de Manhattan entre dos puntos, ¿en qué clase debes hacerlo?

# LA CLASE PUNTO NO EXISTE TODAVÍA, LA ESTOY DEFINIENDO. POR ESO NO PUEDO USAR SIMPLEMENTE

# other:Punto

def distancia\_manhatan(self, other:'Punto') -> float:

distancia:float = abs(self.get\_coordenada\_x() - other.get\_coordenada\_x()) + abs(self.get\_coordenada\_y() - other.get\_coordenada\_y())

return distancia

def distancia\_manhatan\_a\_poligono(self, poligono: 'Poligono' ) -> float:

"""

Dado un polígono devuelve la distancia entre el punto y el polígono, que se define como la menor distancia entre

el punto y los vértices del polígono.

"""

distancias:list[float] = list()

for lado in poligono.get\_lados():

distancias.append(self.distancia\_manhatan(lado.get\_punto\_inicio()))

return min(distancias)

1. ¿Podríamos aprovechar este nuevo método para reutilizarlo en funcionalidad ya existente? ¿Podríamos hacer uso de delegación de métodos entre clases en este ejercicio? Si la respuesta a alguna de estas preguntas es sí, implementa los cambios necesarios.

# EN LA CLASE LINEA, CONCRETAMENTE EN EL MÉTODO CALCULAR LONGITUD

def calcular\_longitud(self) -> float:

return self.get\_punto\_inicio().distancia\_manhatan(self.get\_punto\_fin())

PROBLEMA 2. Relación de asociación. En este ejercicio, se trabajará con la idea de polígonos vecinos. Un polígono se considera vecino de otro si existe al menos un vértice de uno de los polígonos que se encuentra a una distancia menor o igual a un valor dado (distancia máxima) de cualquier vértice del otro polígono.

1. Define el atributo distancia\_maxima\_vecinos que representa el valor de la distancia a partir de la cual dos polígonos ya no se considerarían vecinos. El valor de la distancia máxima es un valor común a todos los polígonos existentes. Por simplicidad, le asignaremos un valor de 10.

class Poligono:

distancia\_maxima\_vecinos:float = 10

@classmethod

def get\_distancia\_maxima\_vecinos(cls) -> float:

return cls.distancia\_maxima\_vecinos

@classmethod

def set\_distancia\_maxima\_vecinos(cls, distancia) -> bool:

if distancia < 0:

print("La distancia ha de ser positiva o cero")

return False

cls.distancia\_maxima\_vecinos = distancia

return True

( . . . )

1. Define el atributo poligonos\_vecinos que representa un listado de polígonos vecinos asociados al polígono actual (inicialmente vacío).

self.\_poligonos\_vecinos:list[Poligono] = list()

1. Implementa el método es\_vecino() capaz de comparar si otro polígono es vecino del polígono actual.

def es\_vecino(self, otro\_poligono:'Poligono') -> bool:

es\_vecino:bool = False

posicion\_lado = 0

while not es\_vecino and posicion\_lado < len(self.get\_lados()):

punto\_actual:Punto = self.get\_lados()[posicion\_lado].get\_punto\_inicio()

distancia:float = punto\_actual.distancia\_manhatan\_a\_poligono(otro\_poligono)

if distancia < Poligono.get\_distancia\_maxima\_vecinos():

es\_vecino = True

posicion\_lado += 1

return es\_vecino

Que funciona con el siguiente método definido en la clase Punto()

def distancia\_manhatan\_a\_poligono(self, poligono: 'Poligono' ) -> float:

"""

Dado un polígono devuelve la distancia entre el punto y el polígono, que se define como la menor distancia entre

el punto y los vértices del polígono.

"""

distancias:list[float] = list()

for lado in poligono.get\_lados():

distancias.append(self.distancia\_manhatan(lado.get\_punto\_inicio()))

return min(distancias)

1. Implementa el método agregar\_vecino() para añadir un polígono a la lista de polígonos, siempre que se cumplan las condiciones para serlo

def agregar\_vecino(self, otro\_poligono:'Poligono') -> bool:

if not self.es\_vecino(otro\_poligono):

print(f"No se pudo agregar {otro\_poligono} a la lista de vecinos")

return False

# NECESARIO PARA EVITAR LA RECURSIVIDAD

if otro\_poligono in self.get\_poligonos\_vecinos():

return False

self.\_poligonos\_vecinos.append(otro\_poligono)

otro\_poligono.agregar\_vecino(self)

print(f"Se ha añadido {otro\_poligono} a la lista de vecinos")

return True

PROBLEMA 3. Relación de composición. Amplía la clase Poligono para que contenga un conjunto de vértices.

1. Añade el método get\_puntos() a la clase Linea para que devuelva los dos puntos que componen la línea.

self.\_vertices : set[Punto] = set()

self.\_vertices = self.get\_vertices()

1. Define el atributo vertices que representa la colección de vértices como un conjunto de puntos, creado a partir de las líneas que definen a un polígono.

def calcular\_vertices(self) -> set[Punto]:

vertices:set[Punto] = set()

for lado in self.get\_lados():

# AÑADO EL INICIAL, EL FINAL DE UN LADO SERÁ EL INICIAL DEL SIGUIENTE

punto\_inicial, \_ = lado.get\_puntos()

vertices.add(punto\_inicial)

return vertices

def get\_vertices(self) -> set[Punto]:

if not self.\_vertices: # NO HAY, LOS CALCULO

self.\_vertices = self.calcular\_vertices()

return self.\_vertices

1. ¿Debemos modificar métodos ya existentes en la clase Poligono?

def set\_lados(self, lineas:list[Linea]) -> bool:

# RECALCULAMOS LOS VÉRTICES CON LOS NUEVOS LADOS

if lineas != self.get\_lados():

self.set\_vertices(self.calcular\_vertices())

if len(lineas) != self.get\_numero\_lados():

print("No se han podido establecer las líneas como lados del polígono.")

return False

print(f"Lineas establecidas. El polígono tiene {self.get\_numero\_lados()}")

self.\_lados = lineas

return True

Nota: Este ejercicio asume que existe coherencia en las líneas que forman un polígono. Sin embargo, como no se iimplementan mecanismos de coherencia, podría pasar que las líneas establecidas no generen realmente un polígonoen el código actual.

4. Clonación. Este ejercicio hace uso de los métodos copy y deepcopy proporcionados por el módulo copy de Python (sin implementar métodos mágicos) sobre un objeto de la clase Poligono. En tu función main realiza:

a) Copia superficial: Clona de manera superficial un polígono. Modifica los atributos color y lados del polígonoclonado. ¿Se modifica el color en ambos objetos? ¿Y los lados? ¿Por qué?

b) Copia profunda: Clona de manera profunda un polígono. Modifica los atributos color y lados del polígono clonado. ¿Se modifica el color en ambos objetos? ¿Y los lados? ¿Por qué?

# PROBLEMA 4, CLONACIÓN

print(f"ANTES DE LA COPIA (POLIGONOS ORIGINALES): POLIGONO 1\n\t{mi\_poligono} POLIGONO 2\n\t{mi\_poligono2} ")

copia\_superficial\_poligono:Poligono = copy.copy(mi\_poligono)

copia\_profunda\_poligono:Poligono = copy.deepcopy(mi\_poligono2)

copia\_superficial\_poligono.set\_color("NARANJA")

copia\_profunda\_poligono.set\_color("LILA")

print(f"DESPUES DE LA COPIA (POLIGONOS ORIGINALES): POLIGONO 1\n\t{mi\_poligono} POLIGONO 2\n\t{mi\_poligono2}")

print(f"DESPUES DE LA COPIA (POLIGONOS COPIA): \n\t{copia\_superficial\_poligono} \n\t{copia\_profunda\_poligono}")

Nota: para hacer este ejercicio crea un método en la clase Polígono llamado eliminar\_lado\_aleatorio(). Este método seleccionará uno de los lados del polígono de forma aleatoria y lo eliminará. ¿Sería necesario actualizar el conjunto de vértices?

import random

def eliminar\_lado\_aleatorio(self) -> None:

lados = self.get\_lados()

self.\_set\_numero\_lados(self.get\_numero\_lados() - 1)

lados.remove(random.randint(0, len(lados)))

self.set\_lados(lados)

5. Relación de asociación bidireccional. Implementa los siguientes pasos:

a) Haz las modificaciones necesarias para que un punto almacene la línea a la que pertenece.

# ARCHIVO punto.py

def \_\_init\_\_(self, x:float, y:float):

self.\_coordenada\_x:float = x

self.\_coordenada\_y:float = y

self.\_linea\_pertenece:'Linea' = None

def get\_linea\_pertenece(self) -> ‘Linea’:

return self.\_linea\_pertenece

def set\_linea\_pertenece(self, linea: ‘Linea’) -> bool:

self.\_linea\_pertenece = linea

return True

# MODIFICAMOS EL CONSTRUCTOR DE LA LÍNEA PARA QUE ESTABLEZCAN LA RELACCIÓN EN LA INSTANCIA PUNTO, ES DECIR:

# ARCHIVO linea.py

class Linea:

def \_\_init\_\_(self, punto\_inicio:Punto, punto\_fin:Punto) -> None:

self.\_punto\_inicio:Punto = punto\_inicio

self.\_punto\_fin:Punto = punto\_fin

punto\_fin.set\_linea\_pertenece(self)

punto\_inicio.set\_linea\_pertenece(self)

1. Haz las modificaciones necesarias para que una línea almacene el polígono al que pertenece.

# ARCHIVO linea.py

class Linea:

def \_\_init\_\_(self, punto\_inicio:Punto, punto\_fin:Punto) -> None:

(. . . )

self.\_poligono\_pertenece = None

def get\_poligono\_pertenece(self) -> 'Poligono':

return self.\_poligono\_pertenece

def set\_poligono\_pertenece(self, poligono:'Poligono') -> bool:

self.\_poligono\_pertenece = poligono

return True

# ARCHIVO poligono.py, dentro del método set\_lados()

def set\_lados(self, lineas:list[Linea]) -> bool:

(. . .)

for linea in lineas:

linea.set\_poligono\_pertenece(self)

return True

**SESIÓN 5: HERENCIA**

Otro tipo de relacción que especifica tipo.

* Clase padre: tendra ciertos métodos principales 🡪 Vehículo (moverse, tanque\_gasolina)
* Clase hija: tiene todos los métodos y atributos de la clase padre + algunos propios 🡪 Autobús (todos los de vehículo + asientos)

En Python una clase puede heredar de más de una clase (**herencia múltiple**). Además cuando una clase (hija) hereda de una clase padre, hereda también los tipos de datos de las clases padres. Así los objetos de la clase *AUTOBÚS* (hereda de *VEHÍCULO*) son también objetos de tipo *VEHÍCULO*

Diagrama

El contenido generado por IA puede ser incorrecto.

Para declarar una clase hija:

class ClaseHija (clasePadre1, clasePadre2, . . ., clasePadreN):

Si declaro una clase hija SIN CONSTRUCTOR Python automáticamente llama al constructor DE LA CLASE PADRE.

class ClasePadre:  
 def \_\_init\_\_(self, parametros\_clase\_padre):

self.param\_padre = parametros\_clase\_padre

class ClaseHija(ClasePadre):

pass

**# LA CLASE HIJA, QUE NO TIENE CONSTRUCTOR SE CONSTRUYE EXÁCTAMENTE IGUAL QUE LA CLASE PADRE**

hija = ClaseHija(parametros\_clase\_padre)

En la clase Hija puedo crear un método *\_\_init\_\_( )* que cambiará el constructor de la calse padre POR COMPLETO. Es decir, me olvido de todas las propiedades de la clase padre porque no llamo en ningún momento al constructor de la clase padre.

class ClasePadre:  
 def \_\_init\_\_(self, parametros\_clase\_padre):

self.param\_padre = parametros\_clase\_padre

class ClaseHija(ClasePadre):

def \_\_init\_\_(self, parametros\_clase\_hija):

self.param\_hija = parametros\_clase\_hija

**# LA CLASE HIJA TIENE UN CONSTRUCTOR PROPIO (NADA QUE VER CON EL CONSTRUCTOR DE LA CLASE PADRE)**

hija = ClaseHija(parametros\_clase\_hija)

Para solucionar esto en el constructor de la clase hija puedo (debo) llamar al constructor de la clase padre (para que además de la clase hija se inicialicen los métodos y atributos que queremos que la clase hija herede de la clase padre). Para esto utilizo *super()*

class ClasePadre:  
 def \_\_init\_\_(self, parametros\_clase\_padre):

self.param\_padre = parametros\_clase\_padre

class ClaseHija(ClasePadre):

def \_\_init\_\_(self, parametros\_clase\_padre, parametros\_clase\_hija):

super().\_\_init\_\_(parametros\_clase\_padre)

self.param\_hija = parametros\_clase\_hija

**# LA CLASE HIJA TIENE EL CONSTRUCTOR DEL PADRE EXTENDIDO A SU PROPIO CONSTRUCTOR**

hija = ClaseHija(parametros\_clase\_padre, parametros\_clase\_hija)

Una clase hija puede hacer TRES cosas con los **MÉTODOS DE LA CLASE PADRE:**

1. USARLOS SIN MODIFICAR

hija = Hija(parametros\_clase\_padre, parametros\_clase\_hija)

hija.metodo\_definido\_en\_la\_clase\_padre()

1. SOBREESCRIBIRLOS

class ClasePadre:

def \_\_init\_\_(self, parametros\_clase\_padre):

self.param\_padre = parametros\_clase\_padre

def f(self):

print("EJECUTO MÉTODO CLASE PADRE")

class ClaseHija(ClasePadre):

def \_\_init\_\_(self, parametros\_clase\_hija):

super().\_\_init\_\_(parametros\_clase\_padre)

self.param\_hija = parametros\_clase\_hija

def f(self):

print("EJECUTO MÉTODO CLASE HIJA")

hija = Hija(parametros\_clase\_padre, parametros\_clase\_hija)

hija.f() # --> EJECUTO MÉTODO CLASE HIJA

1. EXTENDERLOS 🡪 Utilizar el método de la clase padre (con algunas cosas que añado desde la clase hija)

class ClasePadre:

def \_\_init\_\_(self, parametros\_clase\_padre):

self.param\_padre = parametros\_clase\_padre

def f(self, x:int):

print(f"EJECUTO MÉTODO CLASE PADRE x = {x}")

class ClaseHija(ClasePadre):

def \_\_init\_\_(self, parametros\_clase\_hija):

super().\_\_init\_\_(parametros\_clase\_padre)

self.param\_hija = parametros\_clase\_hija

def f(self):

super().f(x = 10)

print("EJECUTO MÉTODO CLASE HIJA")

hija = Hija(parametros\_clase\_padre, parametros\_clase\_hija)

hija.f() # --> EJECUTO MÉTODO CLASE PADRE x = 10 || EJECUTO MÉTODO CLASE HIJA

Si una clase Hija hereda un **ATRIBUTO DE LA CLASE PADRE** puede hacer dos cosas con él:

1. USARLO SIN MODIFICARLO
2. OCULTARLO 🡪 Definir el mismo artículo en la clase hija (nos la pela el de la clase padre)

Además desde la clase hija puedo tener nuevos atributos que no estuvieran previamente definidos en la clase padre.

class A:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

class B(A):

def \_\_init\_\_(self, x, y, z):

super().\_\_init\_\_(x, y)

self.y = y + 10

self.z = z

b = B(1,2,3)

print(b.x) # --> 1 (USO DE UN ATRIBUTO DE LA CLASE PADRE)

print(b.y) # --> 12 (OCULTACIÓN DE UN ATRIBUTO DE LA CLASE PADRE)

print(b.z) # --> 3 (DEFINICIÓN DE UN NUEVO ATRIBUTO QUE NO EXISTE EN LA CLASE PADRE)

a = A(4,5)

print(a.z) # ERROR (SOLO ESTÁ DEFINIDA z EN LA CLASE HIJA)

PROBLEMA 1. Partiendo de la definición de la clase Poligono, vamos a implementar tres nuevos tipos de polígonos: cuadrados, triángulos y círculos. Para ello, veamos cuáles son sus características principales.

1. Clase Triángulo.
   1. Atributos: Un triángulo se caracteriza por ser un polígono de tres lados. Además, un triángulo puede ser de un determinado tipo: escaleno, isósceles o equilátero. Esta propiedad la denominaremos tipo\_triangulo. Nota: recuerda que podemos heredar los atributos del padre, extenderlos u ocultarlos.

from enum import Enum

from week5.poligono import Poligono, TipoRelacionLados, TipoForma

class TipoTriangulo(Enum):

EQUILATERO = 1

ISOSCELES = 2

class Triangulo(Poligono):

def \_\_init\_\_(self,

tipo\_triangulo:TipoTriangulo,

relacion\_lados:TipoRelacionLados = None,

color:str = None,

lados:list['Linea'] = None):

super().\_\_init\_\_(numero\_lados =3,

forma = TipoForma.CONVEXO,

relacion\_lados=relacion\_lados,

color=color,

lados=lados)

self.\_tipo\_triangulo = tipo\_triangulo

* 1. Métodos:
     1. a) Para el cálculo del perímetro, un triángulo debe verificar que tiene exactamente tres lados. En caso contrario, no se podrá hacer uso de esta funcionalidad.

# EXTENDER UN MÉTODO DEL PADRE

def calcular\_perimetro(self) -> float:

# CON EL SELF SE VA AL MÉTODO DEFINIDO EN LA INSTANCIA

# SI ESTE MÉTODO NO ESTÁ DEFINIDO, SE VA AL MÉTODO DE LA CLASE PADRE

if self.get\_numero\_lados() == 3:

super().calcular\_perimetro()

else:

return -1

* + 1. b) Un triángulo debe permitir calcular su área de forma conveniente. Para ello, puedes hacer uso de la fórmula de Herón: A = p s(s − a)(s − b)(s − c), donde a, b y c representan las longitudes de cada uno de los tres lados del triángulo, y s representa el semiperímetro, calculado como s = a+b+c 2 .

# CREAR UN NUEVO MÉTODO DE LA CLASE HIJO

def calcular\_area(self) -> float:

a:float = self.get\_lados()[0].calcular\_longitud()

b:float = self.get\_lados()[1].calcular\_longitud()

c:float = self.get\_lados()[2].calcular\_longitud()

s:float = self.calcular\_perimetro() / 2

return math.sqrt(s \* (s-a) \* (s-b) \* (s-c))

* + 1. c) No se puede permitir eliminar un lado aleatorio de un triángulo, pues dejaría de ser un polígono. Haz los cambios necesarios para tener en cuenta esta restricción.

def eliminar\_lado\_aleatorio(self) -> None:

return False # PARA QUE NO SE PUEDAN ELIMINAR LOS LADOS DE UN TRIÁNGULO.

* + 1. d) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.

# SOBRE ESCRITURA DE MÉTODOS MÁGICOS DE LA CLASE PADRE.

def \_\_str\_\_(self) -> str:

descripcion\_basica:str = super().\_\_str\_\_()

descripcion\_triangulo:str = f"Tipo de triángulo: {self.get\_tipo\_triangulo().name}"

return f"{descripcion\_basica} \n {descripcion\_triangulo}"

def \_\_eq\_\_(self, otro:'Triangulo') -> bool:  
 return True if self.get\_tipo\_triangulo() == otro.get\_tipo\_triangulo() else False

def get\_tipo\_triangulo(self) -> TipoTriangulo:

return self.\_tipo\_triangulo

PROBLEMA 2. Clase Círculo.

1. Atributos: Aunque un círculo no es realmente un polígono, en esta sesión lo consideraremos como tal debido a fines didácticos. Así, un círculo tiene como propiedades su radio y su centro.

from enum import Enum

from week5.elementos\_geometria.triangulo import TipoTriangulo

from week5.poligono import Poligono, TipoRelacionLados, TipoForma

from week5.punto import Punto

class Circulo(Poligono):

def \_\_init\_\_(self,

radio: float,

centro: Punto,

color: str = None,

lados: list['Linea'] = None):

super().\_\_init\_\_(numero\_lados=0,

forma=TipoForma.CONVEXO,

relacion\_lados=TipoRelacionLados.REGULAR,

color=color,

lados=lados)

self.\_radio:float = radio

self.\_centro:Punto = centro

1. Métodos:
2. Un círculo debe permitir calcular su área y su perímetro de forma conveniente.

def get\_radio(self) -> float:

return self.\_radio

def get\_centro(self) -> Punto:

return self.\_centro

# SOBRE ESCRIBEN LOS MÉTODOS DE LA CLASE POLÍGONO

def calcular\_perimetro(self) -> float:

return 2\*pi\*self.get\_radio()

def calcular\_area(self) -> float:

return pi\*pow(self.get\_radio(), 2)

1. Adapta el método es\_vecinos() para que los cálculos se realicen en base a la distancia al centro del círculo (entre centros si se trata de dos círculos, o del centro del círculo a cualquier otro vértice del polígono en caso contrario).

def es\_vecino(self, otro:'Poligono') -> bool:

if otro is Circulo:

distancia\_centros:float = self.get\_centro().distancia\_manhatan(otro.get\_centro())

return True if distancia\_centros < Circulo.get\_distancia\_maxima\_vecinos() else False

else:

# EL OTRO ES UN POLÍGONO

vecinos:bool=False

posicion\_lado:int = 0

while not vecinos and posicion\_lado < len(otro.get\_lados()):

lado\_actual:'Linea' = otro.get\_lados()[posicion\_lado]

punto\_actual:Punto = lado\_actual.get\_punto\_inicio()

distancia:float = punto\_actual.distancia\_manhatan(self.get\_centro())

if distancia < otro.get\_distancia\_maxima\_vecinos():

vecinos = True

return vecinos

1. El escalado de un círculo consiste en modificar su radio en base a los valores de factor y ajuste, en lugar de su número de lados.

def escalar\_poligono(self, factor=None, ajuste\_lados=None) -> bool:

resultado\_escalada:bool = False

if factor:

resultado\_escalada =self.set\_radio(self.get\_radio() \* factor)

if ajuste\_lados:

resultado\_escalada = self.set\_radio(self.get\_radio() + ajuste\_lados)

return resultado\_escalada

d) Realiza los cambios necesarios en los métodos para que el código sea compatible con las restricciones con la clase Poligono analizadas el apartado de atributos. e) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que

def \_\_eq\_\_(self, otro:'Circulo') -> bool:  
 # ENTENDEMOS QUE DOS CIRCULOS SON IGUALES SI TIENEN EL MISMO RADIO   
 return True if self.get\_radio() == otro.get\_radio() else False

def \_set\_numero\_lados(self, nuevo\_numero\_lados:int) -> bool:

# NO SE PUEDE CAMBIAR EL NÚMERO DE LADOS DE UN CIRCULO

return False

def set\_forma(self, forma: TipoForma) -> bool:

# NO SE PUEDE CAMBIAR LA FORMA DE UN CIRCULO

return False

def set\_relacion\_lados(self, relacion\_lados: TipoRelacionLados) -> bool:

# NO SE PUEDE CAMBIAR LA RELACIÓN DE LADOS DE UN CÍRCULO

return False

def calcular\_vertices(self):

# NO SE PUEDEN CALCULAR LOS VÉRTICES DE UN CÍRCULO

pass

def get\_vertices(self):

pass

def eliminar\_lado\_aleatorio(self):

pass

PROBLEMA 3. Clase Cuadrado.

1. Atributos: Un cuadrado se caracteriza por ser un polígono de cuatro lados. Además, definiremos como propiedad de un cuadrado la longitud de su diagonal, denominada longitud\_diagonal, obtenida a partir de la fórmula longitud\_lado ∗ sqrt(2).

from week5.poligono import Poligono, TipoForma, TipoRelacionLados

from week5.linea import Linea

import math

class Cuadrado(Poligono):

def \_\_init\_\_(self, color:str = None, lados:list[Linea]):

super().\_\_init\_\_(numero\_lados = 4,

forma = TipoForma.CONVEXO,

relacion\_lados=TipoRelacionLados.REGULAR,

color = color,

lados=lados)

self.\_longitud\_diagonal:float = self.get\_longitud\_diagonal()

def get\_longitud\_diagonal(self) -> float:

if self.\_longitud\_diagonal:

return self.\_longitud\_diagonal

else:

if self.get\_lados():

return self.get\_lados()[0].calcular\_longitud() \* math.sqrt(2)

else:

print("No se puede calcular la longitud de la diagonal para un cuadrado sin lados establecidos.")

return -1

2. Métodos:

a) Para el cálculo del perímetro, un cuadrado debe verificar que tiene exactamente cuatro lados. En caso contrario, no se podrá hacer uso de esta funcionalidad.

def calcular\_perimetro(self) -> float:  
 if self.get\_numero\_lados() != 4:  
 return super().calcular\_perimetro()  
 else:  
 print("No se puede calcular el perímetro de un cuadrado cuyo número de lados no es cuatro")  
 return -1

b) Un cuadrado debe permitir calcular su área de forma conveniente.

def calcular\_area(self) -> float:  
 if self.get\_numero\_lados() == 4:  
 return pow(self.get\_lados()[0].calcular\_longitud(), 2)  
 else:  
 print("No se puede calcular el área de un cuadrado cuyo número de lados no es cuatro")  
 return -1

c) Revisa los métodos mágicos de esta clase y modifícalos en caso de que sea necesario. Si hay que modificarlos, considera si se trata de una extensión de funcionalidad o de una reescritura completa.

def \_\_str\_\_(self) -> str:

descripcion\_basica:str = super().\_\_str\_\_()

descripcion\_cuadrado:str = f"Cuadrado con longitud: {self.get\_longitud\_diagonal()}"

return f"{descripcion\_basica}\n{descripcion\_cuadrado}"

def \_\_eq\_\_(self, otro\_cuadrado:'Cuadrado') -> bool:

# Dos cuadrados serán iguales si tienen la misma diagonal, por ejemplo

return True if self.get\_longitud\_diagonal() == otro\_cuadrado.get\_longitud\_diagonal() else False

PROBLEMA 4. Definición de nuevas subclases. Define las clases TrianguloEscaleno, TrianguloIsosceles y TrianguloEquilatero para que podamos instanciar objetos de esos tipos, gestionando la herencia y haciendo las comprobaciones pertinentes para que estas clases sean coherentes.

class TrianguloEscaleno(Triangulo):

def \_\_init\_\_(self, color:str, lados:list[Linea] = None):

super().\_\_init\_\_(tipo\_triangulo=TipoTriangulo.ESCALENO, color=color, lados=lados)

def \_\_str\_\_(self) -> str:

return f"{super().\_\_str\_\_()}\n Triangulo escaleno"

class TrianguloEquilatero(Triangulo):

def \_\_init\_\_(self, color:str, lados:list[Linea] = None):

super().\_\_init\_\_(tipo\_triangulo=TipoTriangulo.EQUILATERO, color=color, lados=lados)

def \_\_str\_\_(self) -> str:

return f"{super().\_\_str\_\_()}\n Triángulo equilátero"

class TrianguloIsosceles(Triangulo):

def \_\_init\_\_(self, color:str, lados:list[Linea] = None):

super().\_\_init\_\_(TipoTriangulo.ISOSCELES, color=color, lados=lados)

def \_\_str\_\_(self) -> str:

return f"{super().\_\_str\_\_()}\n Triángulo isósceles"