**HERENCIA MULTIPLE**

Se conoce como herencia multiple al hecho de que una clase “herede” de dos o más clases.

Ejemplo: Una clase Cuadradro, que herede los atributos **base y altura** de la clase FiguraGeometrica y además, herede el atributo color de la clase Colores

class Cuadrado(FiguraGeometrica, Colores):

    def \_\_init\_\_(self, lado, nombre, hexa) -> None:

        FiguraGeometrica.\_\_init\_\_(self, lado, lado)

        Colores.\_\_init\_\_(self, nombre, hexa)

    def calcular\_area(self):

        return f' Area del cuadrado: {self.\_altura \* self.\_base}'

    def get\_hexa(self):

        return self.\_hexadecimal

En la clase cuadrado, se especifica por() las clases de la cual heredará, o a cuales extenderá.

Es importante diferenciar cada \_\_init\_\_ en cada caso:

Este es el de la clase cuadrado, que recibe los atributos **lado, nombre y Hexa.** Esta simplificación se hizo para entender la diferencia, además de que sería inoportuno “pedir” valores de base y altura para un cuadrado cuando todos sus lados son =

def \_\_init\_\_(self, lado, nombre, hexa) -> None:

Cuando heredamos de dos clases no podemos llamar al método super() porque tenemos 2 clases padres, por tanto, para inicializar los \_\_init\_\_ desde clases hijas se usa:

        FiguraGeometrica.\_\_init\_\_(self, lado, lado)

        Colores.\_\_init\_\_(self, nombre, hexa)

Lo que se hace con estas líneas es acceder a la clase padre y llamar a su método \_\_init\_\_ para pasar los parámetros desde ahí.

En este caso se pasa al \_\_init\_\_ de Figura geométrica, además de self, los valores lado y lado, esto debido a que el \_\_init\_\_ declarado en la clase padre es de la siguiente forma:

    def \_\_init\_\_(self, base, altura) -> None:

        self.\_base = base

        self.\_altura = altura

En el caso de **Colores** se pasan los valores de nombre y Hexa, los mismos que “se piden” en el \_\_init\_\_ de la clase padre:

    def \_\_init\_\_(self, nombre, hexadecimal) -> None:

        self.\_nombre = nombre

        self.\_hexadecimal = hexadecimal

Con esta declaración completa, ahora es posible utilizar instancias de la clase Cuadrado y acceder a los atributos de sus clases padre.

**ACCEDIENDO A METODOS**

Se presentan 2 ejemplos. Accediendo a métodos de clase padre, y haciendo que métodos de la clase Cuadrado accedan a atributos de clases padre.

* En la clase Colores definimos el siguiente método:

    def get\_hexa(self):

        return self.\_hexadecimal

Ahora podríamos acceder al mismo desde una instancia de la clase hijo sin ninguna complicación.

asd = Cuadrado(4, 'blanco', 'ffffff')

print(asd.get\_hexa())

* En la clase Cuadrado definimos el siguiente método:

    def calcular\_area(self):

        return f' Area del cuadrado: {self.\_altura \* self.\_base}'

Este método retorna una multiplicación de los valores **\_altura y \_base** a pesar de que en esta clase no están definidos dichos attr, lo están en la clase padre FiguraGeometrica.

Mediante self podemos acceder a los mismos. Cuando inicializamos el objeto Cuadrado, por más que los argumentos lleven otro nombre en realidad estamos inicializando los atributos de la clase Padre, por tanto debemos acceder a los mismos.

Si en el método calcular\_area, en lugar de utilizar self.\_altura y self.\_base utilizásemos self.\_lado (por el parámetro del constructor) obtendríamos un error.

**MRO – METHOD RESOLUTION ORDER**

El método mro especifica el orden en que “se buscan” los métodos o los atributos cuando se invocan desde una clase.

Por ejemplo, en nuestra clase cuadrado accederíamos de la forma:

print(Cuadrado.mro())

y su resultado es:

[<class '\_\_main\_\_.Cuadrado'>, <class 'FiguraGeometrica.FiguraGeometrica'>, <class 'Colores.Colores'>, <class 'object'>]

Podríamos modificar el orden, cambiando el orden en la definición de la clase

class Cuadrado(FiguraGeometrica, Colores):

si en lugar de este código tuviésemos:

class Cuadrado(Colores, FiguraGeometrica):

la clase Cuadrado “buscaría” primero en la clase Cuadrado, luego en la clase Colores y por ultimo en FiguraGeometrica.

**EJERCICIO**

Continuando con el ejercicio se pide que se cree una clase Rectangulo que herede de FiguraGeometrica y Colores.

Además se pide que se encapsulen los atributos de las clases Padres, con sus respectivos setters y getters.

Una funcionalidad importante, tratandose de figuras geométricas es la de la validación de valores negativos, ya que sería ilógico que una figura tenga valores negativos.

Esto puede realizarse con un método (encapsulado) que valide los valores. En este caso se declara dentro de la clase FiguraGeometrica antes del \_\_init\_\_ para utilizarse en el mismo.

    def \_validar(self, valor):

        return True if valor > 0 else False

Para utilizarlo en el init o en los demás métodos, debería anteponer **self**. Y el nombre del método, pasando por parámetro el valor a validar. De este modo, podemos operar en base a su valor True o False.

    def \_\_init\_\_(self, base, altura) -> None:

        if self.\_validar(base):

            self.\_base = base

        else:

            print('Valor de base incorrecto')

        if self.\_validar(altura):

            self.\_altura = altura

        else:

            print('valor de altura incorrecto')

    @base.setter

    def base(self, base):

        if self.\_validar(base):

            self.\_base = base

        else:

            print('valor de base incorrecto')

    @altura.setter

    def altura(self, altura):

        if self.\_validar(altura):

            self.\_altura = altura

        else:

            print('valor altura incorrecto ')