TALLER PROGRAMACIÓN ORIENTADA A OBJETOS CON PYTHON

ACTIVIDADES POR DESARROLLAR:

- Comprender Programación Orientada a Objetos en **PYTHON**
- Aplicar Programación Orientada a Objetos en **PYTHON**

EVIDENCIA(S) A ENTREGAR:

EV1 Desarrollar la actividad a desarrollar propuesta en el taller

CONTROL DEL DOCUMENTO

	Nombre	Cargo	Dependencia	Fecha
Autor (es)	JOSE FERNANDO GALINDO SUAREZ	INSTRUCTOR	CGMLTI	13/04/2024

CONTROL DE CAMBIOS (diligenciar únicamente si realizan ajustes al taller)

	Nombre	Cargo	Dependencia	Fecha	Razón del Cambio
Autor (es)					

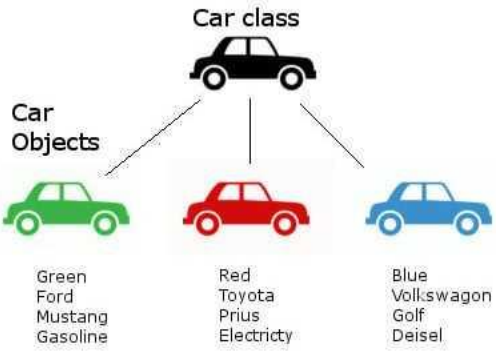
INTRODUCCIÓN

Después de realizar la lectura "programación orientada a objetos", podrá desarrollar los ejercicios dispuestos en este taller.

La programación orientada a objetos es un paradigma de la programación y es el más actualizado en la actualidad, aumentando así la modularidad y la reutilización de código en la programación, rompiendo el paradigma de la programación estructurada, aunque se puede combinar las dos al desarrollar aplicaciones.

Se acerca a la manera como se tratan los objetos en la realidad empezando con la fase de análisis, luego su implementación será la forma más adecuada para su desarrollo.

Clases y objetos.



- **Los objetos son representaciones (simples/complejas) (reales/imaginarias) de cosas: reloj, avión, coche.**
- **No todo puede ser considerado como un objeto, algunas cosas son simplemente características atributos de los objetos: color, velocidad, nombre**



Abstracción funcional

Hay cosas que sabemos que los coches hacen, pero no cómo lo hacen:

- avanzar

- parar
- girar a la derecha
- girar a la izquierda

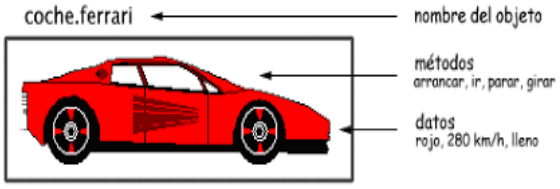
Abstracción de datos

Un coche tiene además ciertos atributos:

- color
- velocidad
- tamaño
- etc.

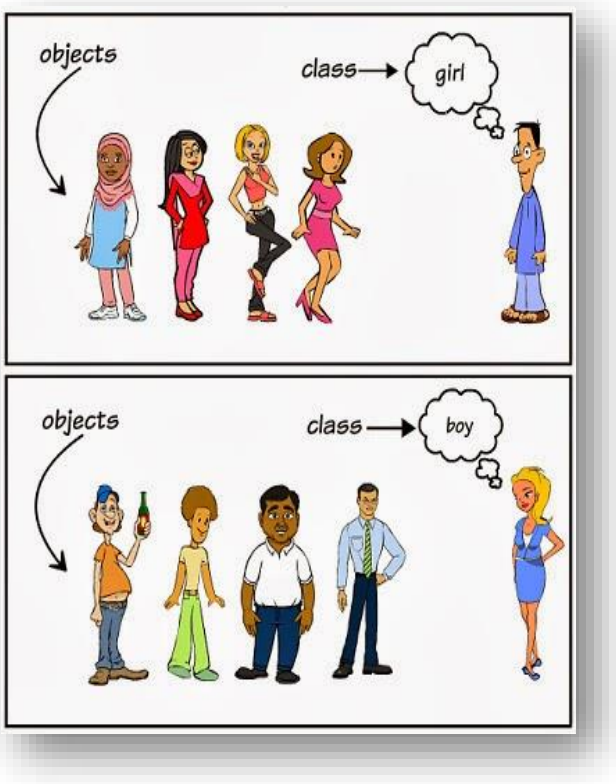
- **Es una forma de agrupar un conjunto de datos (estado) y de funcionalidad (comportamiento) en un mismo bloque de código que luego puede ser referenciado desde otras partes de un programa**

- **La clase a la que pertenece el objeto puede considerarse como un nuevo tipo de datos**



Los objetos permiten tener un control total sobre 'quién' o 'qué' puede acceder a sus variables y métodos, es decir, pueden tener componentes públicos, a los que podrán acceder otros objetos, o componentes privados, a los que únicamente puede acceder él.

Los componentes pueden ser tanto las variables como los métodos de ese objeto.



La programación orientada a objetos (Object Oriented Programming OOP) es un modelo de lenguaje de programación organizado por objetos constituidos por datos y funciones, entre los cuales se pueden crear relaciones como herencia, cohesión, abstracción, polimorfismo y encapsulamiento.

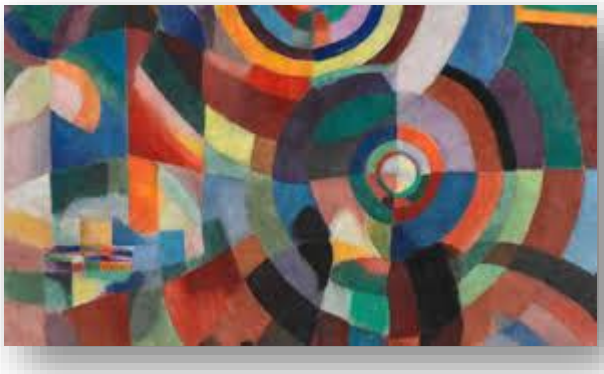
Esto permite que haya una gran flexibilidad y se puedan crear objetos que pueden heredarse y transmitirse sin necesidad de ser modificados continuamente.

JERARQUÍA DE COMPOSICIÓN

El objeto está compuesto por otros objetos con comportamientos distintos.

Esto sirve para representar uno varios objetos que están dentro de otro que los contiene.

DEFINICIÓN DE ABSTRACCIÓN.



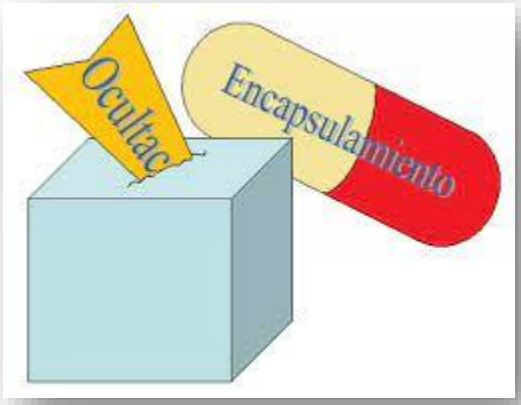
Nos da una visión simplificada de una realidad de la que sólo consideramos determinados aspectos esenciales:

¿qué entendemos por ...?
¿... color de un semáforo?
¿... estado de una cuenta bancaria?
¿... estado de una bombilla?
¿qué necesitamos conocer de un coche para utilizarlo?

La abstracción como técnica de programación

La programación es una tarea compleja.
Mediante la abstracción es posible elaborar software que permita solucionar problemas cada vez más grandes.

DEFINICIÓN DE ENCAPSULAMIENTO



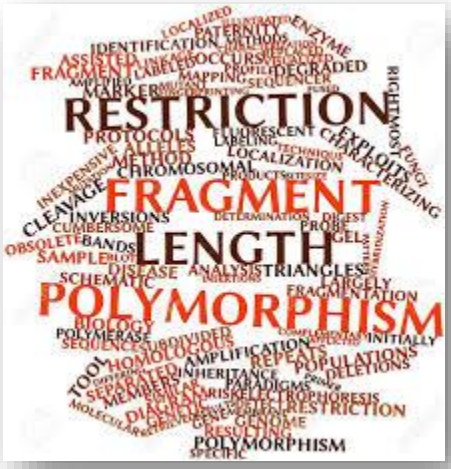
Proceso de ocultamiento de todos los detalles de una entidad que no contribuyen a sus características esenciales.

Abstracción nos centramos en la visión externa.
Encapsulamiento nos centramos en la visión interna.
El acceso a los datos y las operaciones se realiza
mediante una interfaz bien definida.

DEFINICIÓN DE POLIMORFISMO

El polimorfismo se refiere al hecho que un método adopte múltiples formas.

Esto se consigue por medio de la sobrecarga de métodos:
un mismo nombre de método para distintas funcionalidades.
a = Sumar (c,d); a = Sumar (c,d,5);
Sobrecarga de operadores:
un mismo operador con distintas funcionalidades.



En la sobrecarga de funciones se desarrollan distintas funciones con un mismo nombre, pero distinto código.
Las funciones que comparten un mismo nombre deben tener una relación en cuanto a su funcionalidad.
Aunque comparten el mismo nombre, deben tener distintos parámetros. Éstos pueden diferir en:

- El número
- El tipo
- El orden
- El tipo del valor de retorno de una función no es válido como distinción.

Define los datos y el comportamiento, llamado atributos y métodos.

```
'''
nombre #atributo público
_nombre #atributo protegido
__nombre #atributo privado
'''
class Alumno:
  def __init__(self, nombre, nota):
    self.__nombre = nombre
    self.__nota = nota
    print ("Estoy dentro del constructor")

  def mostrar_info(self, maximo=5):
    print ("nombre", self.__nombre, "nota:", self.__nota, "entre ", maximo)
    print ("Objeto actual ", self)
########################################
a1 = Alumno ("Juan", 3)

a1.mostrar_info()


a2 = Alumno ("Margarita", 3)
a2.mostrar_info(4)
print(a2)
print (a2)
```

Character	Icon for field	Icon for method	Visibility
-			private
#			protected
~			package private
+			public

La visibilidad de una propiedad, un método o una constante se puede definir anteponiendo a su declaración una de las palabras reservadas public, protected o private.

- public la variable/función se accede desde cualquier lugar y otras instancias de esa misma clase.
- private la variable/función solamente se accede desde la misma clase que la define.
- protected la variable/función se accede desde la clase que las define y las clases que se herede de ella.

Cuando un método tiene dos guiones bajos al principio y al final de sus nombres se denomina un método mágico. Los métodos mágicos no se pueden llamar directamente desde el código como lo hacemos con los demás métodos, son "mágicos" porque Python los llama automáticamente en determinadas situaciones.

```
__metodo_magico__()   #el doble guión bajo antes y después del nombre del método lo identifica como método mágico
```

El parámetro self

Cada clase da origen a diferentes objetos. El parámetro **self** que se recibe en los constructores o métodos de una clase hace referencia al objeto en particular que está accediendo al comportamiento definido en la clase en un momento determinado.

Todos los métodos en Python deben recibir como primer parámetro el parámetro **self** de esta manera se tiene una referencia al objeto actual que está haciendo uso de los miembros de la clase.

```
obtener_nombre(self):
    return self.nombre      #se devuelve el nombre del objeto actual
```

Ejemplo:

```
class Persona:
      def saludar(self, nombre):
    print("hola ", nombre)
    print(self)
#################################################
p = Persona()
p.saludar("Juan")
p1 = Persona()
p1.saludar("Lina")

print("estoy por fuera y soy :", p)
```

El constructor

Si en Python no se especifica un constructor, de igual manera cada clase tendrá un constructor por defecto que permite instanciar objetos de esa clase.

Así:

```
class Persona:
      pass

juan = Persona()
print (juan)
```

En el código anterior, se creó un objeto de la clase Persona, el objeto se llama **juan** y no hace nada, ya que la clase Persona no tiene código. Pero el objeto existe y tiene memoria asignada para su almacenamiento.

```
class Persona:
      pass

juan = Persona()
print ("memoria juan:", juan) #muestra la posición de memoria
print ("tipo juan:", type(juan)) #muestra de qué tipo es juan

pedro = Persona()
print ("memoria pedro:", pedro) #muestra la posición de memoria
print ("tipo pedro:", type(pedro)) #muestra de qué tipo es juan
```

__ init__ ()

Para definir el constructor de una clase en Python, se usa el método mágico **init**

```
class Persona:
      def __init__(self):
    print ("Me estoy creando ", self)
################################################

juan = Persona()   #creamos objetos de la clase Persona
pedro = Persona()
```

Complementando:

```
class Persona:
      def __init__(self, ced, nom, ed):
    print ("Me estoy creando ", self)
```

```
    self.cedula = ced
    self.nombre = nom
    self.edad = ed
################################################



obj1 = Persona(123, "Luis", 35)   #creamos objetos de la clase Persona
obj2 = Persona(345, "Lina", 20)
print (obj1.cedula, obj1.nombre, obj1.edad)
print(obj1)
print (obj2.cedula, obj2.nombre, obj2.edad)
print(obj2)
```

En Python no es necesario declarar explícitamente los atributos de una clase. Ellos van a estar siempre almacenados en el parámetro self

```
class Persona:
      def __init__(self, n, e, pepito):
    self.nombre = n
    self.edad = e
    self.ciudad = pepito

  def mostrar_informacion(self):  #mediante el self el método reconoce al
objeto que está invocando el método
    print(self.nombre, self.edad, self.ciudad)
################################################

juan = Persona("Juan Villa", 23, "Medellín")
pedro = Persona("Pedro Valencia", 40, "Pereira")

pedro.pasaporte = "AX4743"
pedro.novia = "Florencia"

juan.alergia = "mariscos"

print (juan.ciudad)
print (pedro.ciudad)

print (juan.alergia)

#juan.mostrar_informacion()
#pedro.mostrar_informacion()
```

En Python se pueden crear atributos "al vuelo", es decir, después de crear un objeto se le pueden asignar los atributos deseados y estos formarán parte de los miembros de este objeto, los cuales recordemos podemos acceder a través del parámetro **self**

```
class Persona:
      def mostrar_informacion(self):
    print(vars(self))      # devuelve un diccionario con los atributos y
valores del objeto actual

  def __str__(self):
     return "Hola soy el objeto con esta información: " + str(vars(self))
################################################

juan = Persona()
juan.altura = 180
juan.peso = 85

juan.mostrar_informacion()
print (juan)

obj = Persona()
obj.sexo = "masculino"
obj.salario = 1000000
obj.tipo_sangre = "o+"
obj.mostrar_informacion()

print (obj)
```

Otros métodos mágicos

Las clases tienen por defecto algunos métodos mágicos que podemos sobrescribir para alterar su funcionamiento por defecto.

__ str__ ()

El método mágico **str** en una clase tiene el comportamiento por defecto de retornar una cadena con la clase a la que pertenece el objeto y su dirección de memoria. Este método mágico es llamado cada vez que a la función **str()** le enviamos por parámetro un objeto.

```
class Lenguaje:
      def __init__(self, nombre):
    self.nombre = nombre

  def __str__(self):
```

```
    return self.nombre
##########
l1 = Lenguaje("Java")
print(l1)

l2 = Lenguaje("Python")
print(l2)

l3 = Lenguaje("C#")
print(l3)
```

De lo contrario, se imprime me la dirección de memoria

```
class Persona:
    def __init__(self):
      print (str(self))
################################################

juan = Persona()   #creamos objetos de la clase Persona
print(juan)
```

En toda clase se puede sobrescribir el método mágico **str** para obtener otro tipo de resultado al imprimir un objeto de dicha clase

```
class Persona:
      def __str__(self):
    return "Este es un objeto cool otra vez"


################################################

juan = Persona()   #creamos objetos de la clase Persona y nos ejecuta el
constructor init
print (juan)
```

Se imprime el nombre.

```
class Persona:
      def __init__(self, nombre, edad, direccion):
    self.nombre = nombre
    self.edad = edad
    self.direccion = direccion

  def __str__(self):
    return self.nombre
```

```
###############################################

objeto1 = Persona("Andres Julian Valencia", 23, "calle 3")
print (objeto1)
objeto2 = Persona("Ximena Diaz", 35, "calle 66")
print (objeto2)
```

Atributos de Instancia y de Clase

Los atributos de los ejemplos anteriores son atributos de instancia, esto quiere decir que cada instancia (objeto) tiene su propio conjunto de atributos con sus propios valores. Estos atributos son accedidos a través del parámetro **self** y el cambio de valor en un atributo de instancia no tiene efecto en los demás objetos. Incluso, gracias a los atributos dinámicos en Python, dos instancias de la misma clase pueden tener atributos diferentes con valores por supuesto diferentes.

```
class Persona:
      def __init__(self):
    pass
  def mostrar_informacion(self):
    print(vars(self))      # devuelve un diccionario con los atributos y
valores del objeto actual
###############################################

juan = Persona()
juan.altura = 180
juan.peso = 85

pedro = Persona()
pedro.altura = 170
pedro.telefono = "300232323"

print ("Información del objeto juan:")
juan.mostrar_informacion()
print ("Información del objeto pedro:")
pedro.mostrar_informacion()
print ("Cambiamos la altura de pedro")
pedro.altura = 200
print ("Información del objeto juan:")
juan.mostrar_informacion()
print ("Información del objeto pedro:")
pedro.mostrar_informacion()
```

Un atributo de clase por su parte, es un atributo cuyo valor es el mismo para todas las instancias de una clase pues pertenece a la clase y no a sus instancias, aunque estas lo puedan acceder. En el siguiente código vamos a usar un atributo de clase para llevar la cuenta del número total de personas creadas:

```
class Persona:
      personas_total = 0 #este es un atributo de clase
  def __init__(self):
    Persona.personas_total += 1  #cada vez que se crea un objeto se incrementa
el atributo de clase
################################################

juan = Persona()
print ("personas_total a través de juan: ", juan.personas_total)
pedro = Persona()
print ("personas_total a través de pedro: ", pedro.personas_total)
print ("personas_total a través de juan: ", juan.personas_total)
luis = Persona()print ("personas_total a través de luis: ",
luis.personas_total)print ("personas_total a través de juan: ",
juan.personas_total)

print ("personas_total a través de pedro: ", pedro.personas_total)
```

Métodos

Los métodos se definen igual que una función con la diferencia de siempre recibir el primer parámetro **self** que como vimos antes hace referencia al objeto actual.

Sintaxis:

```
def mi_metodo(self):
    cuerpo del método
def mi_metodo(self, parametro1, parametro2, parametro_n):
    cuerpo del método
```

Ejemplo:

```
class Persona:
      def __init__(self, n, e, c):
    self.nombre = n
    self.edad = e
    self.ciudad = c

  def cumplir_anos(self):
    self.edad += 1
```

```
  def get_edad(self):
    return self.edad

  def set_edad(self, x):
    self.edad = x

  def puede_votar(self):
    if self.edad >= 18:
      return True
    else:
      return False

  def cambiar_ciudad(self, nueva_ciudad):
    self.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {self.ciudad}.
Podrá votar??? {self.puede_votar()}")
################################################################################
##################
p = Persona("Raul Diaz", 17, "Manizales")
p.mostrar_info()
p.cumplir_anos()
p.cambiar_ciudad("Cali")
p.mostrar_info()
```

Métodos de Clase

De la misma manera que existen los atributos de clase, también existen los métodos de clase. Un método de clase es un método que tiene acceso a la clase (atributos de clase u otros métodos de clase).

Para declarar un método como método de clase debemos poner el decorador **@classmethod** antes de la declaración del método de clase.

Estos métodos en lugar de recibir como primer parámetro self, reciben como primer parámetro **cls** que representa a la clase y a través de este parámetro puede acceder a los miembros de la clase.

```
class Persona:
      ciudad = "Manizales"  #atributo de clase

  def __init__(self, nombre, edad):
```

```
    self.nombre = nombre
    self.edad = edad

  def cumplir_anos(self):
    self.edad += 1


  @classmethod    #este es un método de clase
  def cambiar_ciudad(cls, nueva_ciudad):
    cls.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {Persona.ciudad}
")
################################################################################

p1 = Persona("Hugo", 35)
p2 = Persona("Paco", 20)
p3 = Persona("Luis", 29)

p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()


p1.cambiar_ciudad("Cali")  #la invocación de un método de clase se realiza a
través de la clase

p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()

Persona.ciudad = "Bogotá"
p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()
```

Métodos Estáticos

Los métodos estáticos no acceden ni modifican el estado de los objetos (self) ni de las clases (cls). Son métodos que cumplen una función particular y que quedan asociados a una determinada clase.

Para declarar un método como método estático debemos poner el decorador **@staticmethod** antes de la declaración del método estático.

Estos métodos no reciben ni el parámetro self ni el parámetro cls ya que como lo dijimos antes no conocen ni modifican el estado de la clase ni de sus objetos.

```
import random
class Persona:
  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  @staticmethod    #este es un método estático
  def mensaje_hoy():
    mensajes =["La vida es bella", "Mejor tarde que nunca", "Dios te
bendiga", "Vive para servir"]
    print (random.choice(mensajes))

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años")
#############################################################################
########################
Persona.mensaje_hoy()
p1 = Persona ("Pablo", 38)
p1.mostrar_info()
Persona.mensaje_hoy()
```

APROPIACIÓN

1. Caso No 1

- Cree una clase llamada **Estudiante** con los siguientes atributos de instancia: **nombre**, **nota1** y **nota2**.
- Defina un constructor que reciba como parámetros el nombre, la nota1 y la nota2 del estudiante.
- Defina los siguientes métodos de instancia: **obtener_nota_promedio()** este método devuelve la nota promedio del estudiante y **mostrar_informacion()** este método muestra en pantalla todos los datos del estudiante (nombre, nota1, nota2 y nota promedio).
- Pruebe el funcionamiento de la clase.

2. Caso No 2

Encapsule la clase **Estudiante** creada en el punto anterior, de tal manera que todos sus atributos de instancia queden **privados**. Agregue los métodos

necesarios dentro de la clase para que continúe su funcionamiento y adicionalmente solo acepte notas entre 0 y 5.

3. Caso No 3

A la clase **Estudiante** del punto anterior realice las adaptaciones necesarias para que al imprimir un objeto completo de esta clase se presente su nombre y nota promedio.

4. Caso No 4

Se requiere que cualquier estudiante creado pertenezca a la misma institución de tal manera que si un estudiante cambia de institución, todos deben hacerlo. Lleve además un control del número de estudiantes de la institución. Realice los ajustes a la clase **Estudiante** del punto anterior para lograr estos objetivos.

5. Caso No 5

Realice el método **ver_escala()** que al ser invocado en la clase **Estudiante** presente una tabla con la escala de calificación de todos los estudiantes de la institución. Esta escala es la siguiente:

Nota	Escala
0 a 2.9	Baja
3 a 3.9	Media
4 a 4.5	Alta
4.6 a 5	Sobresaliente

Permita que al invocar al método desde la clase **Estudiante.ver_escala()** se muestre la tabla anterior. Utilice el módulo tabulate para que la presentación sea la indicada.

RELACIONES (Fuente)

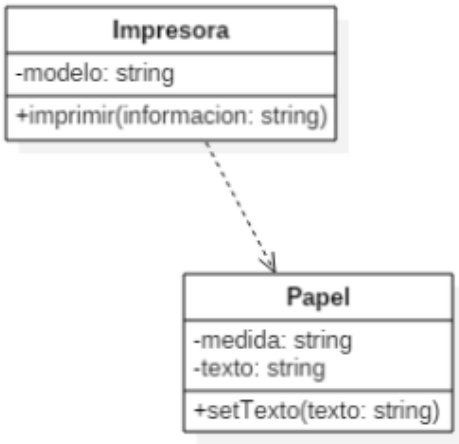
Asociación

- La asociación es una relación donde los objetos tienen su propio ciclo de vida y no hay propietario.
- La frase para comprobar una relación de este tipo es A es una parte de B.

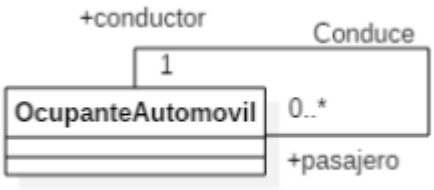
Tomemos un ejemplo de Instructor y Aprendiz.
Varios Aprendices pueden asociarse con un solo Instructor y un solo Aprendiz puede asociarse con varios Instructores. Ambos se pueden crear y eliminar de forma independiente.

Composición

Dependencia



Asociación reflexiva

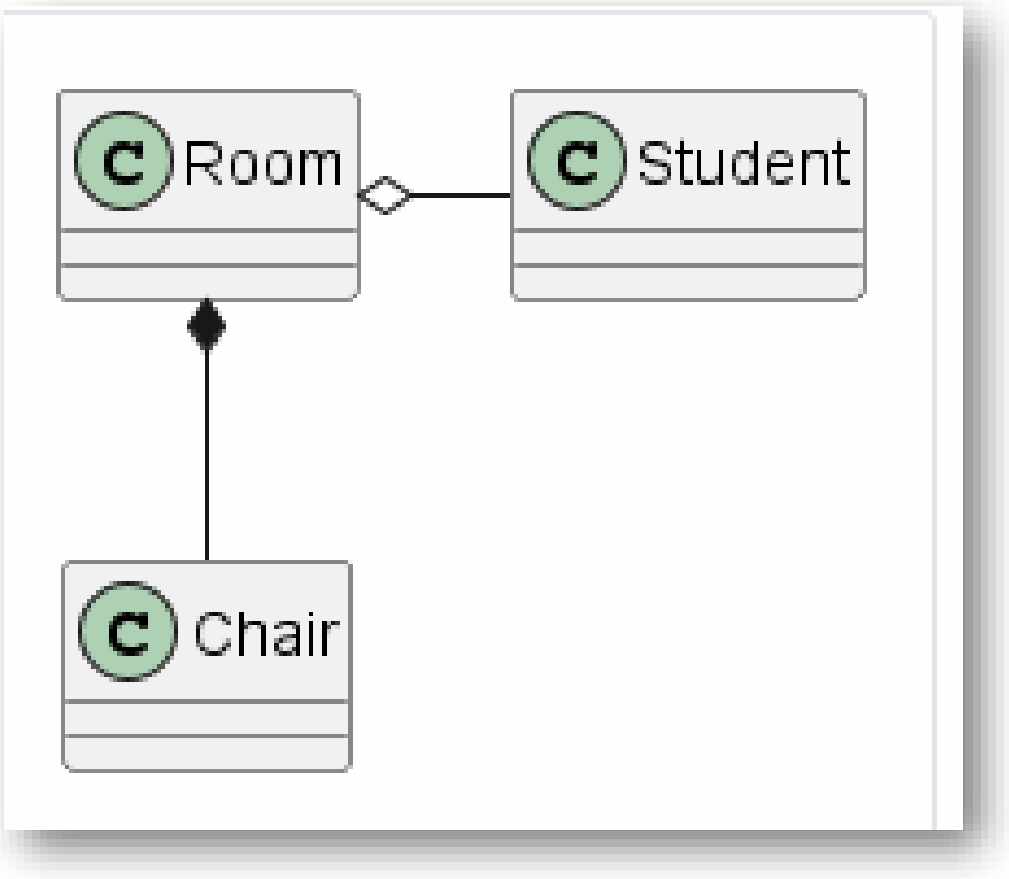


● La agregación es una forma especializada de asociación donde todos los objetos tienen su propia existencia, pero hay propiedad y los objetos secundarios no pueden pertenecer a otro objeto principal.

Un ejemplo: Área e Instructor. Un solo Instructor no puede pertenecer a múltiples Áreas, pero si eliminamos el Área, el objeto Instructor no se destruirá. Podemos pensarlo como una relación "tiene una".

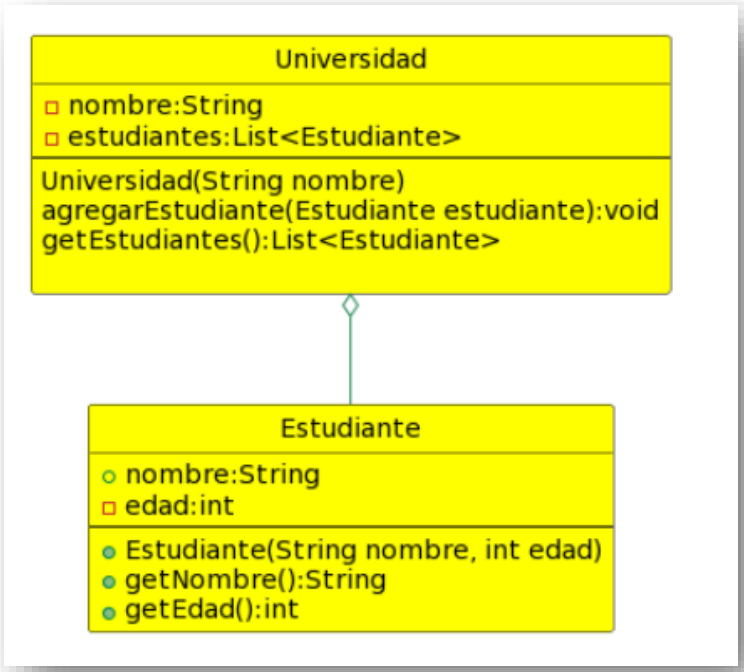
● La composición es una forma especializada de Agregación donde el objeto secundario no tiene su ciclo de vida propio y si se elimina el objeto principal, también se eliminarán todos los objetos secundarios.

La relación entre Preguntas y Opciones. Las preguntas individuales pueden tener múltiples opciones y la opción no puede pertenecer a otra pregunta.



AGREGACIÓN

La agregación: es un tipo de relación entre clases en la programación orientada a objetos, donde una clase contiene a otra, pero ambas pueden existir de manera independiente.



Supongamos que queremos modelar una relación entre una clase Universidad y una clase Estudiante. La Universidad tiene una lista de estudiantes, pero los estudiantes pueden existir sin estar asociados a una universidad específica.

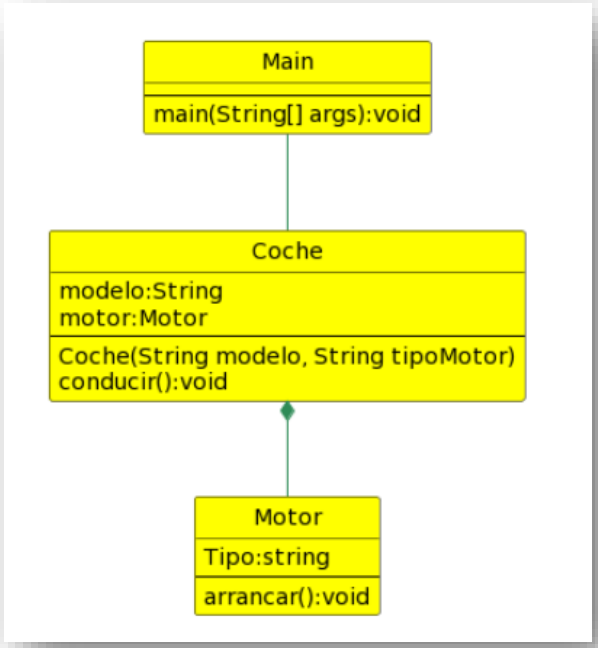
```
class Estudiante:
  def __init__(self,nom,edad):
    self.__nombre=nom
    self.__edad=edad

  def getNombre(self):
        return self.__nombre
  def getEdad(self):
        return self.__edad

class Universidad:

  def __init__(self,nombre):
    self.__nombre=nombre
    self.__estudiantes=[]

  def agregarEstudiante(self,estudiante:Estudiante):
    self.__estudiantes.append(estudiante)
```

```
  def getEstudiantes(self):
    for x in self.__estudiantes:
          print(f"Nombre: {x.getNombre()},Edad: {x.getEdad()}")


estudiante1=Estudiante("Juan",20)
estudiante2=Estudiante("Maria",22)
universidad=Universidad("Universidad xyz")
universidad.agregarEstudiante(estudiante1)
universidad.agregarEstudiante(estudiante2)

universidad.getEstudiantes()
```

En este ejemplo, la clase Universidad contiene una lista de estudiantes (estudiantes). La relación entre Universidad y Estudiante es de agregación, ya que los estudiantes pueden existir de forma independiente y no se destruyen si la universidad deja de existir. La universidad simplemente mantiene una lista de estudiantes asociados.

COMPOSICIÓN

La composición en programación orientada a objetos se refiere a la relación entre dos clases donde una clase contiene una instancia de otra clase.



Supongamos que queremos modelar una relación entre una clase Motor y una clase Coche. En este caso, un coche tiene un motor, y utilizaremos la composición para representar esta relación.

```
// Definición de la clase Motor
class Motor:
  def __init_(self,tipo):
    self.tipo=tipo
  def arrancar(self):
    print("Motor arrancado")

class Coche:
  def __init__(self,modelo,tipoMotor):
    self.__modelo=modelo
    self.__tipoMotor=tipoMotor
  def getModelo(self):
        return self.__modelo
  def conducir(self):
```

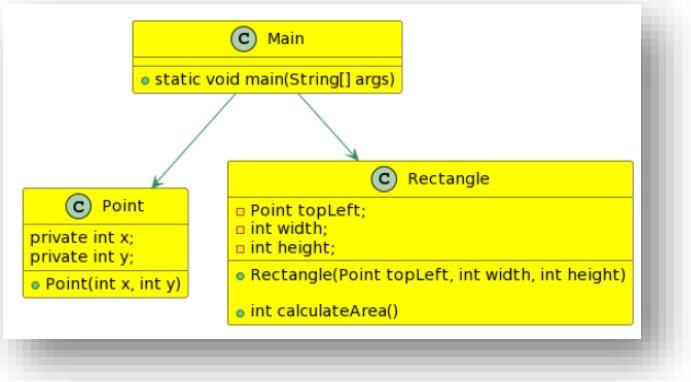
```
    print("Conduciendo el coche modelo: "+self.getModelo())

miCoche=Coche("Sedan","Gasolina")
miCoche.conducir()
```

En este ejemplo, la clase Coche tiene una instancia de la clase Motor. Al crear un objeto Coche, también se crea un objeto Motor dentro de él. La composición permite que el coche utilice las funcionalidades del motor, como el método arrancar().

DEPENDENCIA

En Python, la dependencia de clases se refiere a la relación en la cual una clase utiliza o depende de otra clase para llevar a cabo ciertas operaciones o funcionalidades. Estas dependencias se manifiestan a través de la utilización de instancias de otras clases, llamadas a métodos de otras clases o referencias a constantes y variables de otras clases.

Cuando una clase A depende de otra clase B, significa que la clase A utiliza o requiere la funcionalidad proporcionada por la clase B para realizar alguna tarea específica. La dependencia entre clases es un concepto fundamental en la programación orientada a objetos y es esencial para construir aplicaciones modularizadas y mantenibles.

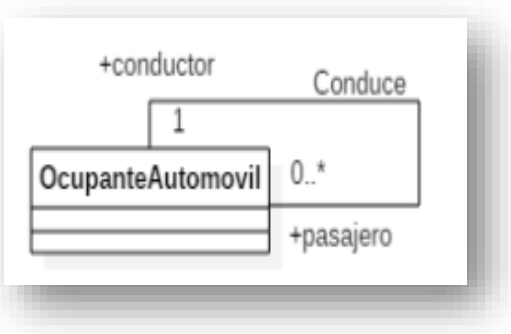



```
// Clase Rectangle que depende de la clase Point
class Point:
      def __init__(self,x,y):
            self.__x=x
            self.__y=y
```

```
class Rectangle:
  def __init__(self,topLeft:Point,width,height):
        self.__topLeft=topLeft
        self.__width=width
        self.__height=height
  def calculateArea(self):
        return self.__width*self.__height
point=Point(10,20)
rectangle =  Rectangle(point, 30, 40)
area = rectangle.calculateArea()
print("Area del rectángulo: " , area)
```

En este ejemplo, la clase Rectangle depende de la clase Point para representar las coordenadas del vértice superior izquierdo del rectángulo. La relación de dependencia se establece al crear una instancia de Point y pasarlo como argumento al constructor de Rectangle. La clase Rectangle utiliza la funcionalidad proporcionada por la clase Point para realizar sus operaciones.

ASOCIACIÓN

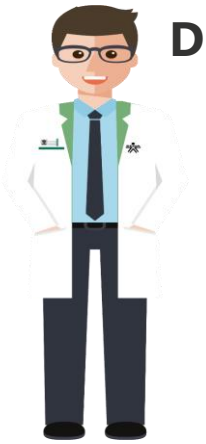


La asociación reflexiva en Python se refiere a una relación entre una clase y sí misma. En otras palabras, una instancia de la clase puede estar relacionada con otra instancia de la misma clase. Este tipo de relación se utiliza cuando necesitas modelar una relación entre objetos del mismo tipo. Puedes implementar asociaciones reflexivas a través de atributos de instancia que son instancias de la misma clase o a través de métodos que devuelven instancias de la misma clase.

```
class Persona:
      mejorAmigo=None
      def __init__(self,nombre,edad) -> None:
            self.__nombre=nombre
            self.__edad=edad
      def getNombre(self):
        return self.__nombre
```

```
      def establecerMejorAmigo(self,amigo:'Persona'):
            self.mejorAmigo=amigo
      def obtenerMejorAmigo(self) -> 'Persona':
            return self.mejorAmigo
persona1=Persona("Alice",25)
persona2=Persona("Bob",30)
persona1.establecerMejorAmigo(persona2);
amigoDePersona1 = persona1.obtenerMejorAmigo()
print((persona1.getNombre() + " es el mejor amigo es " +
amigoDePersona1.getNombre()))
```

En este ejemplo, la clase Persona tiene un atributo llamado mejorAmigo que es de tipo Persona, lo que indica que una persona puede tener otra persona como su mejor amigo. Los métodos establecerMejorAmigo y obtenerMejorAmigo se utilizan para establecer y obtener la relación de mejor amigo entre instancias de la clase Persona. La asociación reflexiva se crea cuando se establece que una instancia de Persona puede tener otra instancia de Persona como su mejor amigo.



Diferencia entre Agregación y Composición

- Las relaciones en una composición son requeridas, en la agregación son opcionales.
- En la composición una clase partícula no puede ser compartida por otras clases compuestas, en la agregación esto es posible.
- La relación de vida de la clase partícula y la clase contenedora, es muy fuerte, de hecho es la relación más fuerte; tanto así que si un objeto de la clase contenedora es destruido la clase partícula también lo será. Esto en la agregación no ocurre.

Visibilidad

La visibilidad de una propiedad, un método o una constante se puede definir anteponiendo a su declaración una de las palabras reservadas *public*, *protected* o *private*. A los miembros de clase declarados como 'public' se puede acceder desde donde sea; a los miembros declarados como 'protected', solo desde la misma clase, mediante clases heredadas o desde la clase padre. A los miembros declarados como 'private' únicamente se puede acceder desde la clase que los definió.

Visibilidad de propiedades

Las propiedades de clases deben ser definidas como 'public', 'private' o 'protected'. Si se declaran usando *var*, serán definidas como 'public'.

HERENCIA DE OBJETOS

Esto es útil para la definición y abstracción de la funcionalidad y permite la implementación de funcionalidad adicional en objetos similares sin la necesidad de Re implementar toda la funcionalidad compartida.

En el contexto de programación en Python, la herencia es un concepto fundamental que permite la creación de nuevas clases basadas en clases ya existentes. La herencia es un mecanismo que permite a una clase heredar propiedades y comportamientos de otra clase, lo que facilita la reutilización del código y la creación de una jerarquía de clases.

Una clase que hereda de otra se llama subclase o clase hija, y la clase de la cual se hereda se llama superclase o clase padre.

```
// Superclase o clase padre
class Animal:
      def comer(self):
```

```
    print("El animal está comiendo")
class Perro(Animal):
  def ladrar(self):
    print("El perro está ladrando")
miPerro =  Perro()
miPerro.comer()
miPerro.ladrar()
```

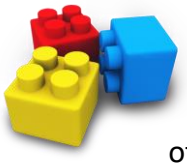
```
El animal está comiendo
El perro está ladrando
```

En este ejemplo, la clase **`Perro`** hereda de la clase **`Animal`**. Esto significa que la clase **`Perro`** tiene automáticamente el método **`comer`** de la clase **`Animal`**, además de tener su propio método **`ladrar`**. La herencia permite organizar el código de una manera que refleje la relación "es un/a" entre las clases, lo que facilita la comprensión y el mantenimiento del código.

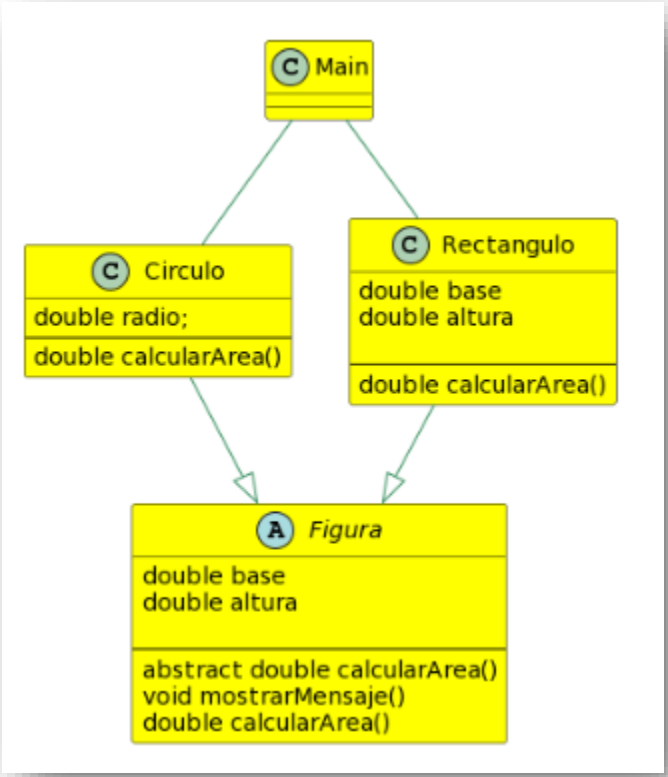
DESDE EL INTERIOR DE LA DEFINICIÓN DE LA CLASE

Las dos palabras claves especiales *self* y *cls* son utilizadas para acceder a propiedades y métodos desde el interior de la definición de la clase.

CLASES ABSTRACTAS



Las clases abstractas son aquellas que por sí mismas no se pueden identificar con algo 'concreto' (no existen como tal en el mundo real), pero sí poseen determinadas características que son comunes en otras clases que pueden ser creadas a partir de ellas. .



En Python, una clase abstracta es una clase que no se puede instanciar por sí misma y que sirve como base para otras clases. Las clases abstractas en Python suelen usarse para definir interfaces o comportamientos comunes que las clases derivadas deben implementar.

Python proporciona la biblioteca abc (Abstract Base Classes) que permite crear clases abstractas. Para crear una clase abstracta en Python, debes seguir los siguientes pasos:

Importar el módulo abc: El módulo abc contiene las herramientas necesarias para crear clases abstractas.

Hacer que la clase herede de ABC: La clase abstracta debe heredar de la clase ABC del módulo abc.

Definir métodos abstractos con el decorador @abstractmethod: Los métodos que deben ser implementados por las subclases se marcan con el decorador @abstractmethod del módulo abc.

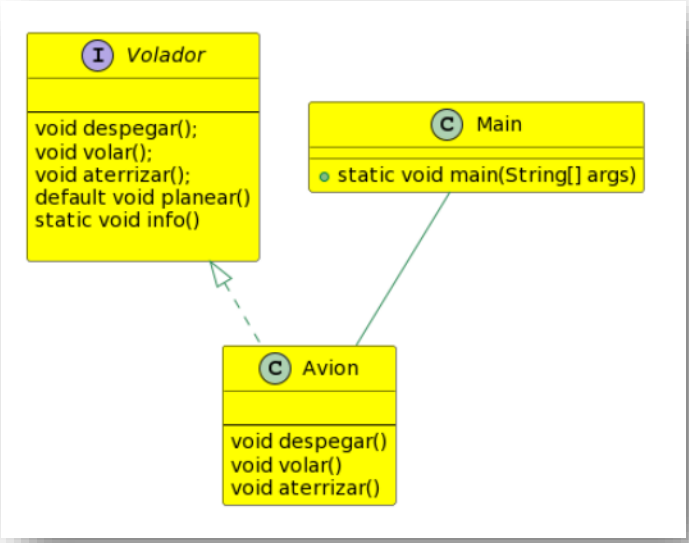
A continuación, un ejemplo de cómo crear una clase abstracta en Python: Una clase abstracta es una clase que no se puede instanciar directamente, es decir, no se pueden crear objetos de ella. La finalidad principal de una clase abstracta es proporcionar una estructura común para clases que comparten ciertas características, pero que pueden tener comportamientos específicos diferentes.

Un ejemplo de cómo crear una clase abstracta en Python:

```python
from abc import ABC, abstractmethod

class Vehiculo(ABC):
    def __init__(self, color):
        self.color = color

    @abstractmethod
    def acelerar(self):
        pass

    @abstractmethod
    def frenar(self):
        pass

class Carro(Vehiculo):
    def acelerar(self):
        print("El carro está acelerando")

    def frenar(self):
        print("El carro está frenando")

# No puedes instanciar Vehiculo directamente porque es abstracto.
# vehiculo = Vehiculo("Rojo")  # Esto generará un error.

# Puedes instanciar Carro, que es una subclase concreta de Vehiculo.
carro = Carro("Azul")
carro.acelerar()  # Salida: "El carro está acelerando"
carro.frenar()    # Salida: "El carro está frenando"
```

En este ejemplo, Vehiculo es una clase abstracta con dos métodos abstractos: acelerar y frenar. La clase Carro es una subclase concreta de Vehiculo que implementa estos métodos abstractos. Si intentaras crear una instancia de Vehiculo directamente, recibirías un error, porque Vehiculo es abstracta. En cambio, puedes crear una instancia de Carro y llamar a los métodos acelerar y frenar que están implementados en Carro.

INTERFACES



En Python, una interfaz puede ser implementada de forma similar a una clase abstracta. De hecho, en Python no existe una distinción explícita entre una clase abstracta y una interfaz como en otros lenguajes de programación. Ambas se definen de la misma manera utilizando clases abstractas con métodos abstractos.

A continuación, se muestra cómo definir una interfaz en Python utilizando la biblioteca abc (Abstract Base Classes) y las clases abstractas:

Importar el módulo abc: Importa el módulo abc que contiene las herramientas necesarias para crear clases abstractas.

Definir una clase abstracta: Crea una clase que hereda de ABC y contiene métodos abstractos que las subclases deben implementar.

Usar el decorador @abstractmethod: Utiliza el decorador @abstractmethod para marcar los métodos que las subclases deben implementar.

A continuación, un ejemplo de cómo definir una interfaz en Python:

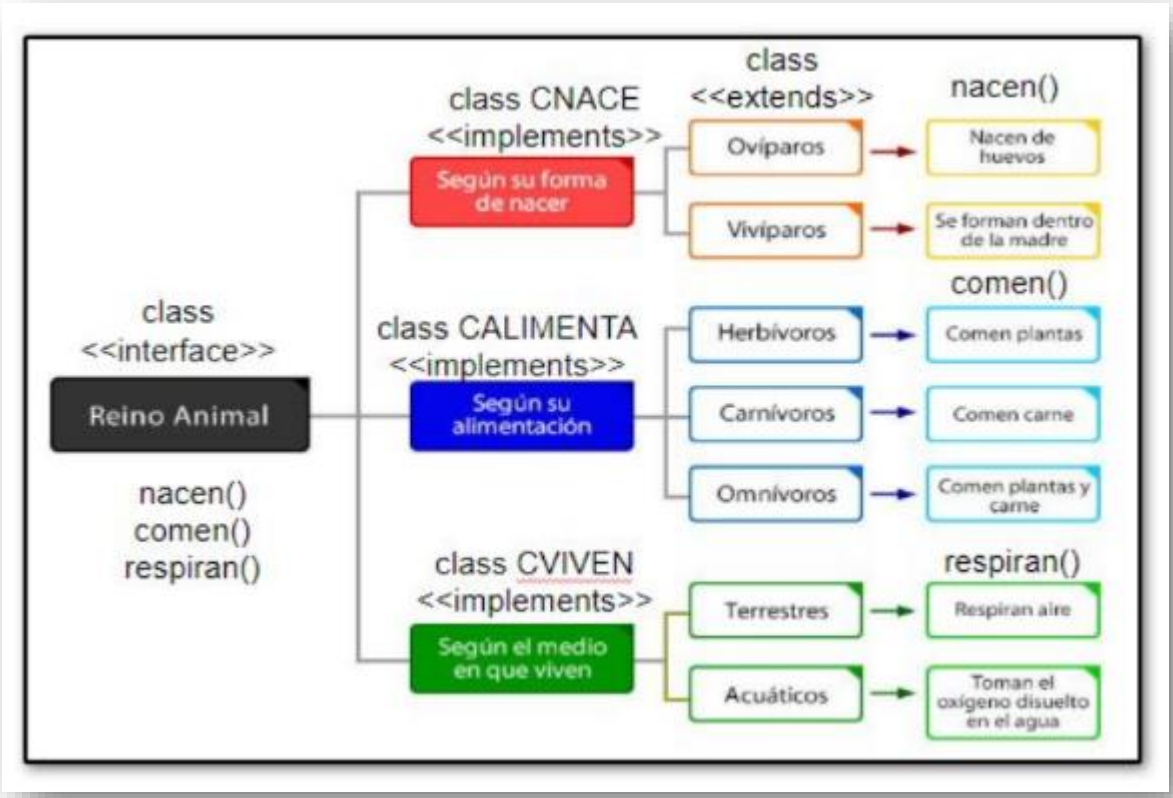
```
// Definición de la interfaz
from abc import ABC, abstractmethod

# Definimos una interfaz (clase abstracta) llamada 'Instrumento'
class Instrumento(ABC):

    # Método abstracto que las subclases deben implementar
    @abstractmethod
```

```
    def tocar(self):
        pass

# Subclase 'Guitarra' que implementa la interfaz 'Instrumento'
class Guitarra(Instrumento):

    # Implementa el método 'tocar'
    def tocar(self):
        print("Tocando la guitarra")

# Subclase 'Piano' que implementa la interfaz 'Instrumento'
class Piano(Instrumento):

    # Implementa el método 'tocar'
    def tocar(self):
        print("Tocando el piano")

# Crear instancias de las subclases
guitarra = Guitarra()
piano = Piano()

# Llamar al método 'tocar' en las instancias de las subclases
guitarra.tocar()  # Salida: "Tocando la guitarra"
piano.tocar()     # Salida: "Tocando el piano"
```

En este ejemplo, la clase Instrumento actúa como una interfaz con un método abstracto tocar. Las subclases Guitarra y Piano implementan el método tocar según la interfaz definida en Instrumento. Puedes crear instancias de Guitarra y Piano y llamar a sus métodos tocar, pero no puedes crear una instancia de Instrumento directamente, porque es una clase abstracta.

En Python, las interfaces son principalmente una forma de definir contratos o comportamientos que las clases concretas deben seguir.

PRACTICA DIRIGIDA POR EL INSTRUCTOR




```
from abc import ABC, abstractmethod
class ReinoAnimal(ABC):
  def comen(self,tipo):
    pass
  def nacen(self,tipo):
    pass
class CNACEN(ReinoAnimal):
  def comen(self,tipo):
    pass
  def nacen(self,tipo):
    if tipo==1:
          print("OVIPAROS Nace de huevos")
    if tipo==2:
          print("VIVIPAROS Se forman dentro de la madre")
class CALIMENTAN(ReinoAnimal):
      def nacen(self,tipo):
            pass
```

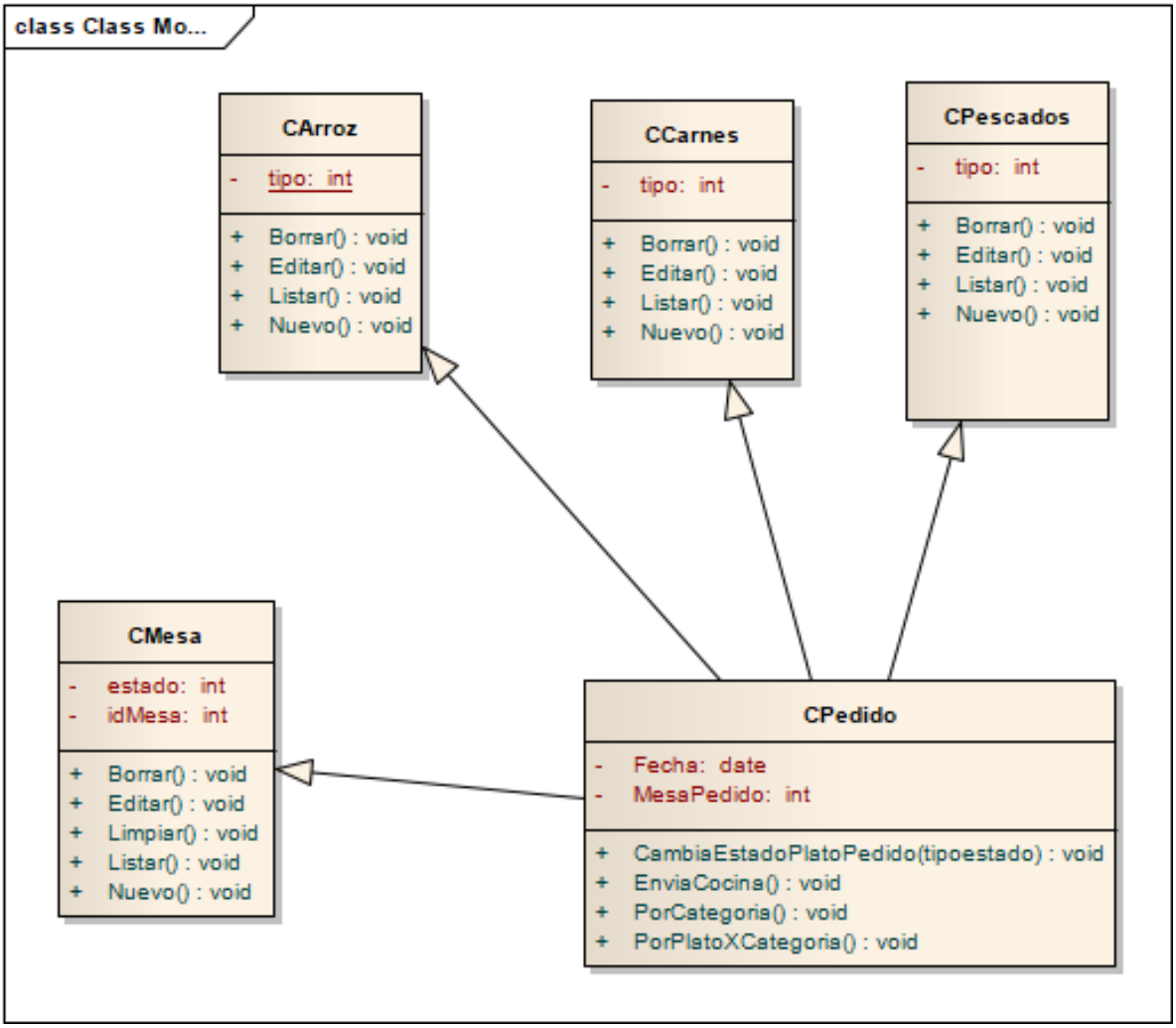
```
      def comen(self,tipo):
        if tipo==1:
              print("HERVIVOROS comen hiervas")
        if tipo==2:
              print("CARNIVOROS comen carne")
        if tipo==3:
              print("OMNIVOROS comen hierva y carne ")
class Oviparos(CNACEN):
      def __init__(self):
            self.nacen(1)
class Viviparos(CNACEN):
      def __init__(self):
            self.nacen(2)
class Hervivoros(CALIMENTAN):
      def __init__(self):
            self.comen(1)
class Carnivoros(CALIMENTAN):
      def __init__(self):
            self.comen(2)
class Ovnivoros (CALIMENTAN):
      def __init__(self):
            self.comen(3)
viviparo =  Viviparos()
oviparo =  Oviparos()
hervivoros= Hervivoros()
carnivoros= Carnivoros()
ovnivoros= Ovnivoros()
```

```
VIVIPAROS Se forman dentro de la madre
OVIPAROS Nace de huevos
HERVIVOROS comen hiervas
CARNIVOROS comen carne
OMNIVOROS comen hierva y carne
```

RETO: Termine la codificación de: "Según el medio en que viven"

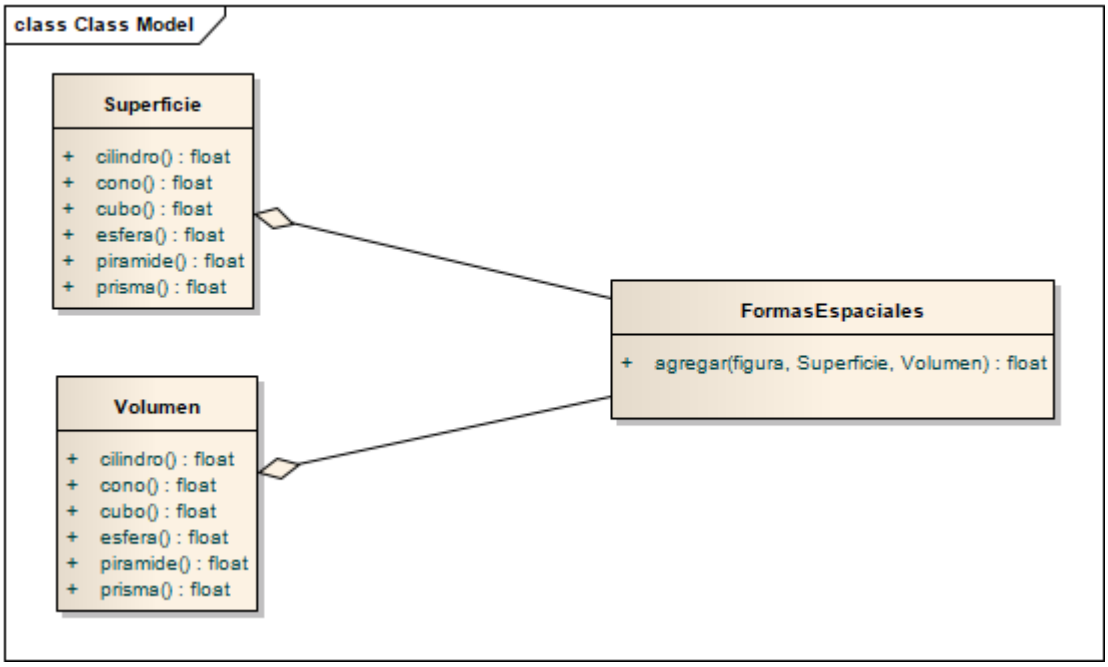
APROPIACIÓN No 1:

Realizar en Python el siguiente diagrama de clases, no olvide guardar una copia en el portafolio de evidencias.



APROPIACIÓN No 2:

Realizar en Python el siguiente diagrama de clases, no olvide guardar una copia en el portafolio de evidencias.

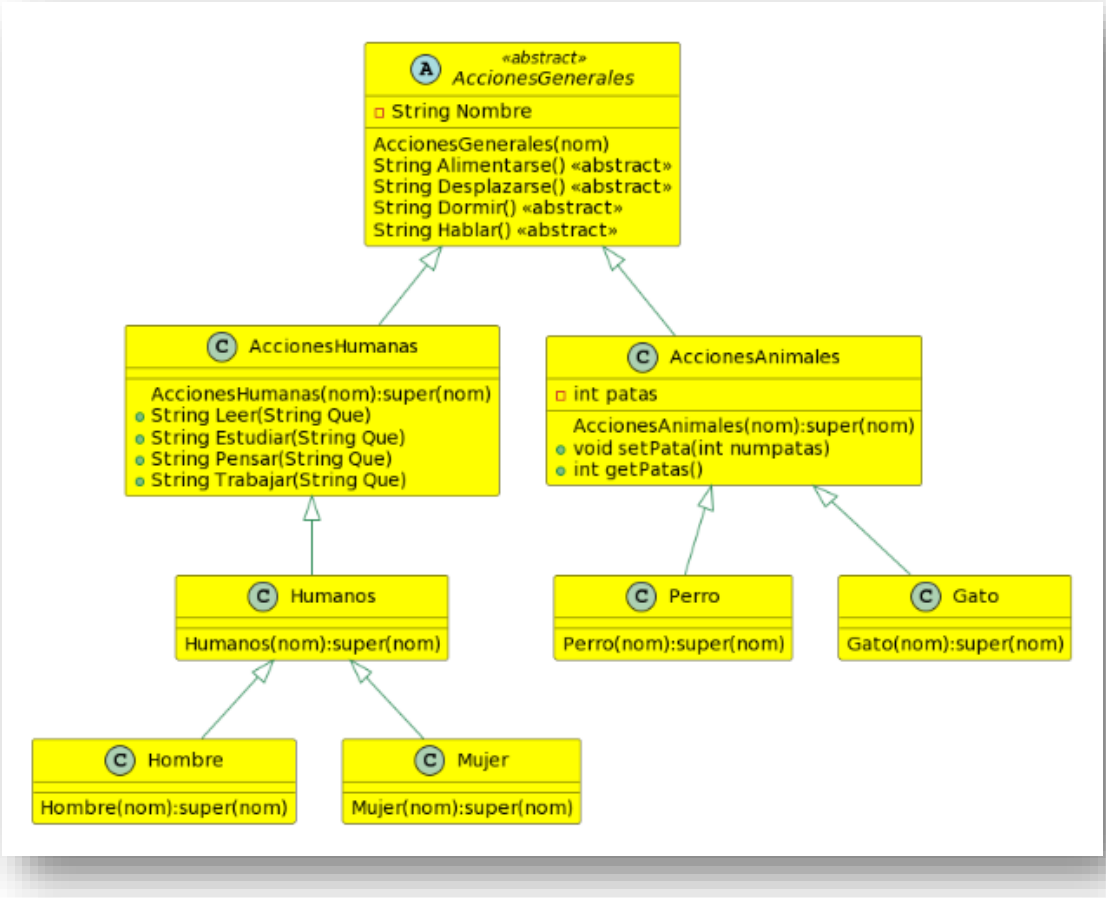


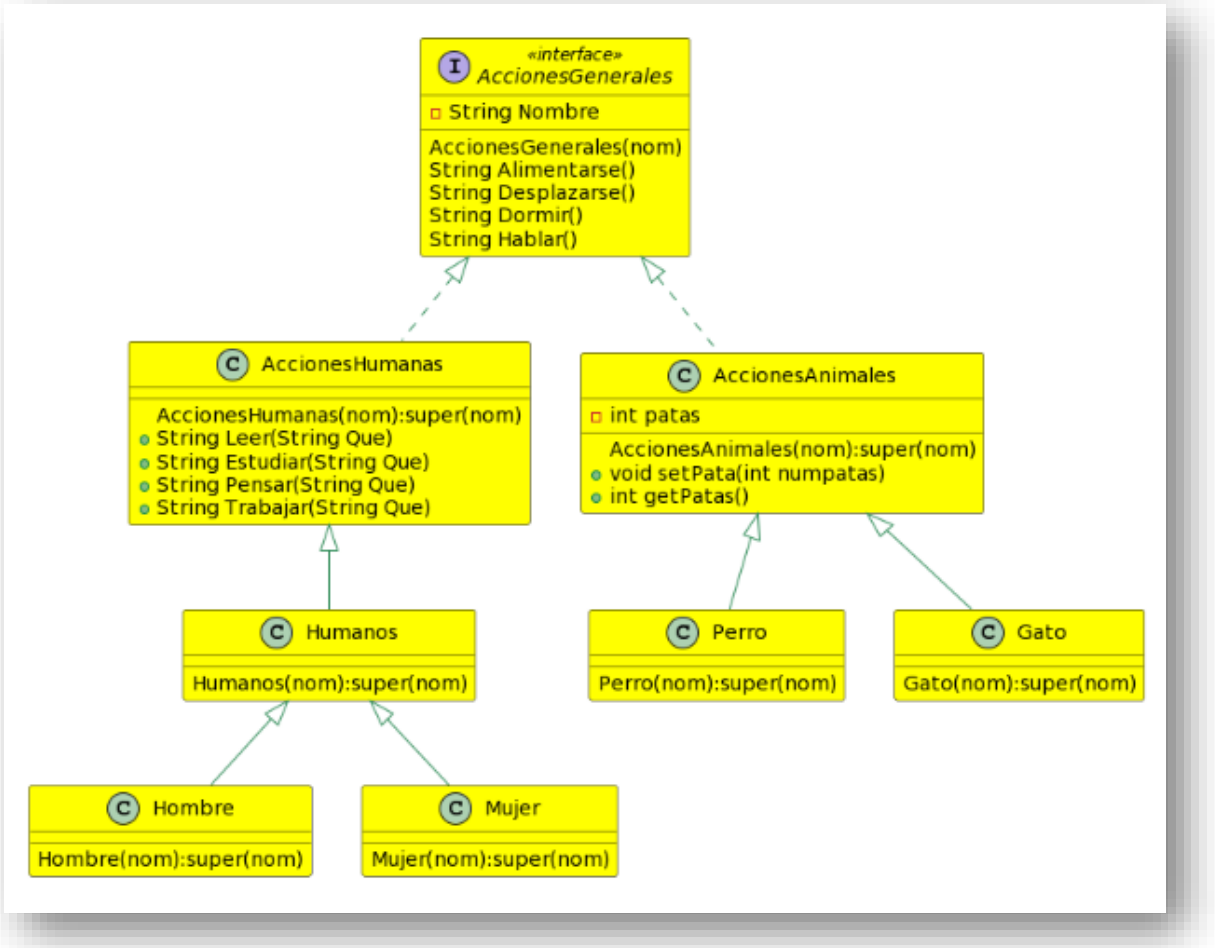
APROPIACIÓN No 3:

Realizar en Python el siguiente diagrama de clases, no olvide guardar una copia en el portafolio de evidencias.

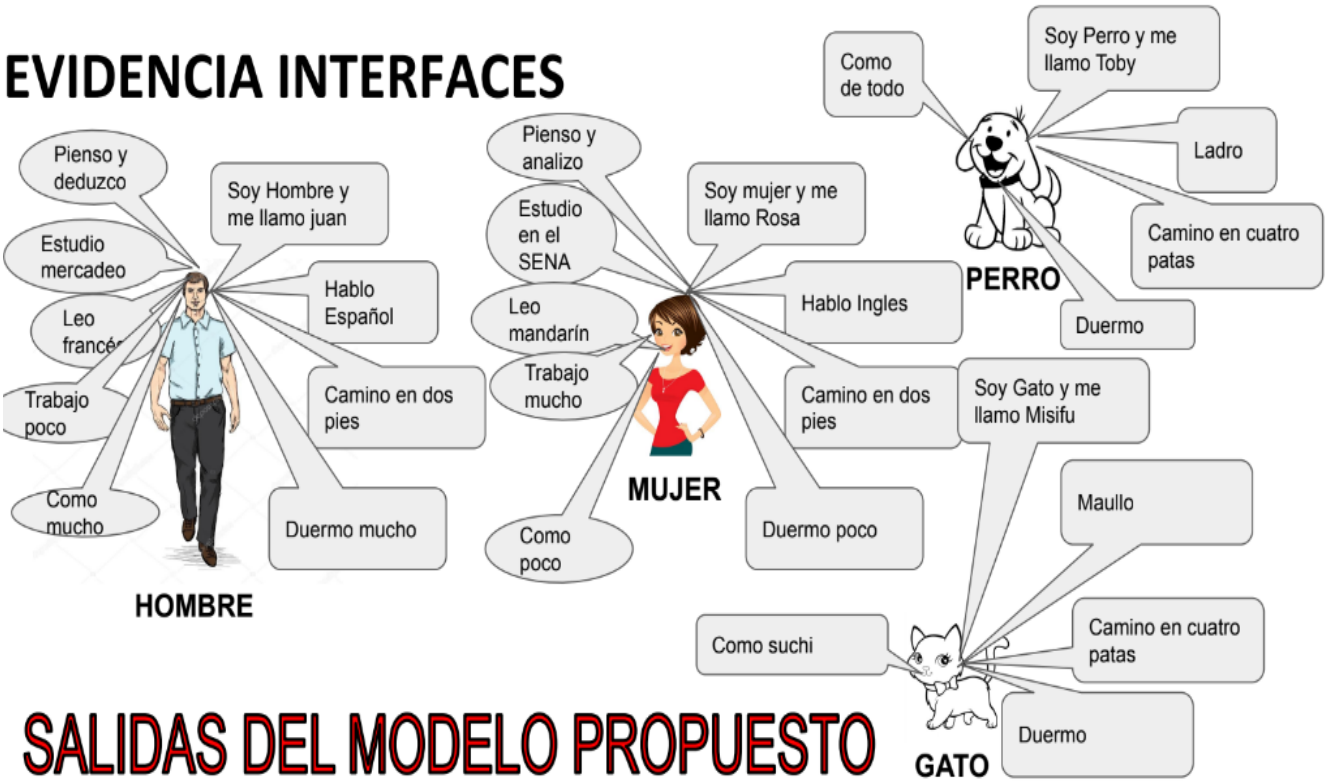
APROPIACIÓN No 4:

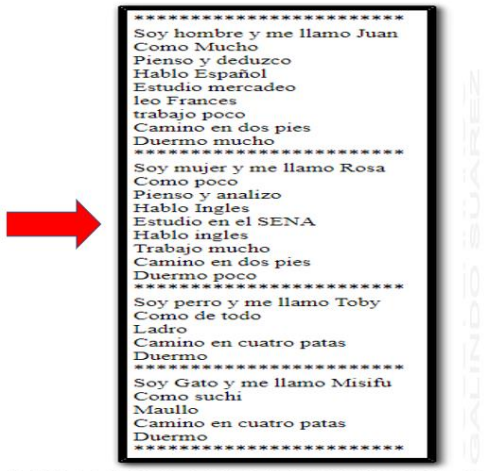
Realizar en Python el siguiente diagrama de clases, no olvide guardar una copia en el portafolio de evidencias.



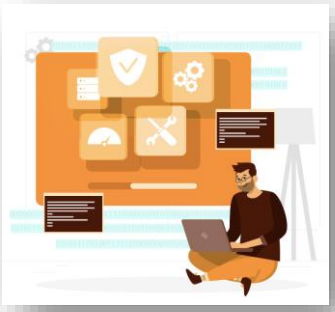


Desarrollar la evidencia de interfaces de acuerdo a la salida de la siguiente grafica:





SALIDAS DEL MODELO PROPUESTO



Desarrolle esta evidencia en archivo , de acuerdo con la gráfica anterior y teniendo en cuenta las salidas del modelo propuesto, en la herramienta de su preferencia y envíela al instructor, guarde una copia en el portafolio del aprendiz; este taller se debe realizar con los integrantes del grupo de proyecto de formación.

Recuerde enviarlo en un archivo ZIP no RAR ni otra extensión de comprimido y colocar como nombre "**T3_poo.zip**" y subirlo al LMS individualmente.