MÓDULO 1. PROGRAMACIÓN ORIENTADA A OBJETOS II

Loreto Pelegrín Castillo

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL
Región de Murcia
Sef Servicio Regional DE EMPLEO y FORMACIÓN
AVANZA

Índice:

1

Herencia

1. Herencia

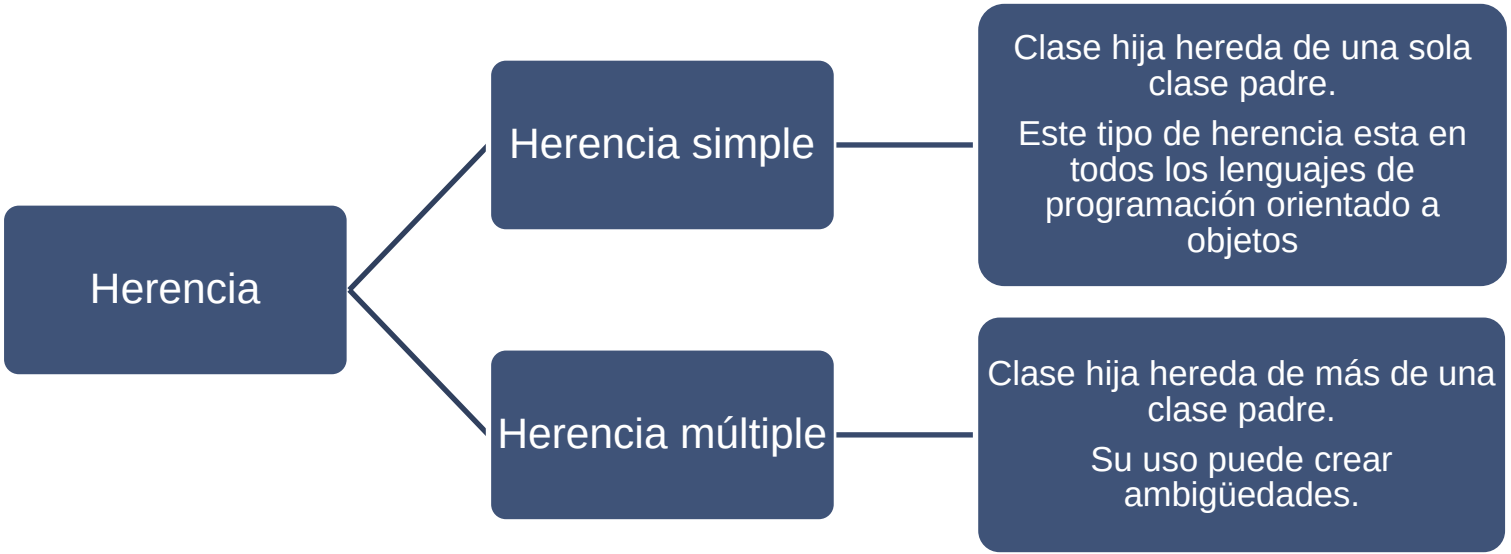
- La **herencia** es un fenómeno que ocurre cuando , desde una clase, a la que llamaremos **clase padre** o superclase, se comparten atributos y métodos a otra subclase, a la que llamaremos **clase hija**.
- Esta clase tendrá implementadas directamente todas las características de la clase padre, por lo que no tendremos que escribirlas de nuevo.
- Además, en nuestra clase hija podremos definir nuevos atributos y métodos.

1. Herencia

- La **herencia** es un concepto que facilita enormemente la reutilización de código, puesto que podemos heredar todas las veces que sea necesario.

- Esto permite que una clase de nueva creación pueda heredar de una clase padre, que a su vez a heredado de otra clase. Se produce así lo que se conoce como **herencia multinivel**.

1. Herencia


Herencia
Herencia simple
Clase hija hereda de una sola clase padre.
Este tipo de herencia esta en todos los lenguajes de programación orientado a objetos
Herencia múltiple
Clase hija hereda de más de una clase padre.
Su uso puede crear ambigüedades.

1. Herencia

- En Python se pueden usar ambos tipos de herencia.
- Se utiliza la palabra clave **pass** para definir métodos abstractos (sin implementación) que serán implementados en las subclases.

1.1. Herencia Simple

```python
"""Creamos la clase padre Libro"""

class Libro:

    def __init__(self,nombre,autor,editorial):
        self.nombre = nombre
        self.autor = autor
        self.editorial = editorial

    def mostrar_datos(self):
        return f"El nombre del libro es {self.nombre}, esta escrito por {self.autor} y la editorial es {self.editorial}."

"""Creamos la clase LibroDigital, que va a heredar las caracteristicas de Libro
Para ello, despues de crear la clase LibroDigital hay que poner entre parentesis de la clase que hereda
en este caso (Libro)"""
class LibroDigital(Libro):
    #Ponemos pass ya que esta clase no va a hacer nada
    pass

primer = LibroDigital("El tunel","Ernesto Sabato","DeBolsillo")
#Usamos el metodo que hemos heredado de la clase Libro
print(primer.mostrar_datos())
#El nombre del libro es El tunel, esta escrito por Ernesto Sabato y la editorial es DeBolsillo.
```

1.2. Método super()

- Es un método especial que permite acceder directamente a los métodos y atributos de la clase padre.

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

class Mujer(Persona):
    #Constructor de la clase Mujer
    def __init__(self, nombre, edad, altura, sexo):
        #LLamada al contructor de la clase padre.
        super().__init__(nombre,edad,altura)
        #Parametro agredado a nuestra clase heredada
        self.sexo = sexo

    def presentacion(self):
        print(f"Hola, me llamo {self.nombre}, tengo {self.edad} años y soy {self.sexo}")
        #Hola, me llamo Loreto, tengo 32 años y soy Mujer

persona1 = Mujer("Loreto",32,1.63,"Mujer")
persona1.presentacion()
```

1.3. Herencia múltiple

- Para realizar herencia múltiple pasaremos como parámetros las clases de las que vamos a heredar. El **orden** en el que pasemos las clases es **importante**, puesto que, si tenemos un método con el mismo nombre en ambas clases heredadas, se usará el método de la que pongamos en primer lugar.
- Ocurrirá lo mismo si heredamos métodos con el mismo nombre mediante **super().**
- Consejo: Para disminuir en número de posibles errores de código, se debería de evitar en lo posible la herencia múltiple.

1.3. Herencia múltiple

```
"""Crearemos una clase Acciones y mantenemos las clases anteriores de Mujer y Persona."""
class Acciones(Persona):
    def presentacion(self):
        print(f"Hola, me llamo {self.nombre}, tengo {self.edad} años y mido {self.altura}")

class Descendiente(Acciones,Mujer):
    """Se llamará al metodo de presentacion de la clase Acciones por estar colocado en primer lugar"""
    def miPresentacion(self):
        super().presentacion()

adolescente = Descendiente("Clara",17,1.70,"mujer")
adolescente.presentacion()
#Hola, me llamo Clara, tengo 17 años y mido 1.7
```

Ejercicios de clase.

- Ejercicio 1:
  - Crear una clase llamada **Figura,** con los atributos de nombre y color.
  - Definir un método describir. En el cual se describa la figura.
  - Definir un método para calcular el área. (Sin implementar)
  - Crear clases hijas:
    - **Cuadrado**: Agregar un atributo lado.
    - **Circulo**: Agregar un atributo radio.
    - Para ambas clases hijas, definir un método describir. (utilizar super)

Ejercicios de clase.

Ejercicio 2:

- Crear una clase llamada **Dispositivos,** con los atributos de marca y modelo.
- Definir un método apagar y otro encender.
- Crear clases hijas:
  - **Ordenador**: Agregar el atributo procesador, y crear un método de jugar al ordenador.
  - **Móvil**: Agregar el atributo compañía de teléfono y crear un método para cambiar de compañía de teléfono.
  - **Televisor**: Agregar el atributo pulgadas y un crear un método para cambiar de canal.

Ejercicios de clase.

- Ejercicio 3:
  - Crear una clase llamada **Vehículo,** con los atributos de marca, modelo y año. Añadir en el constructor otro atributo, velocidad e inicializarlo a 0.
  - Definir los métodos de arrancar, acelerar y frenar.
  - Crear clase hija:
    - **Coche**: Agregar un atributo número de puertas.
    - **Moto**: Agregar un atributo radio cilindrada.
    - **Camión:** Agregar un atributo booleano refrigerado.

2

Encapsulamiento

2. Encapsulamiento

- El **encapsulamiento** es un concepto importante en POO. Consiste en prohibir el acceso directo a los atributos y a los métodos internos de las clases desde el exterior de estas para así poder protegerlos y que no se puedan modificar directamente.
- En Python no existe un encapsulamiento como tal, pero se puede simular, precediendo a los atributos y métodos de las clases con **2 barras bajas**.
- Cuando el intérprete de Python encuentra las 2 barras bajas, internamente lo que hace es cambiarle el nombre al atributo o al método para no poder acceder a él directamente.

2. Encapsulamiento

- Para acceder a estos datos encapsulados debemos crear métodos que funcionen como intermediarios.
- Por convención estos métodos suelen empezar por get o set según queramos coger información o modificarla.
- Los atributos encapsulados se suelen llamar **privados**.

2. Encapsulamiento

```python
class Vivienda:
    baños = 2
    #Atributo encapsulado
    __habitaciones = 4

    #Metodo para obtener información
    def get_habitaciones(self):
        return self.__habitaciones

    #Metodo para modificar el atributo
    def set_habitaciones(self,num_habitaciones):
        self.__habitaciones = num_habitaciones


casa = Vivienda()
print(casa.baños) #2

#casa.__habitaciones
"""Traceback (most recent call last):
  File "c:\Users\loreto\Desktop\Academia Avanza\Python\vivienda.py", line 18, in <module>
    casa.__habitaciones
AttributeError: 'Vivienda' object has no attribute '__habitaciones'. Did you mean: 'get_habitaciones'?"""

#Obtenemos el atributo mediante el método
print(casa.get_habitaciones()) #4

#Modificamos el atributo encapsulado del objeto
casa.set_habitaciones(5)
print(casa.get_habitaciones())#5
```

2. Encapsulamiento

```python
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.__titular = titular  # Atributo privado
        self.__saldo = saldo_inicial # Atributo privado

    #Metodo para obtener la información del titular
    def get_titular(self):
        return self.__titular

    #Metodo para modificar la informacón del titular
    def set_titular(self,nuevo_titular):
        self.__titular = nuevo_titular

    #Metodos para modificar el salgo
    def depositar(self, cantidad):
        if cantidad > 0:
            self.__saldo += cantidad
            print("Depósito realizado con éxito.")
        else:
            print("Cantidad inválida para depósito.")

    def retirar(self, cantidad):
        if 0 < cantidad <= self.__saldo:
            self.__saldo -= cantidad
            print("Retiro realizado con éxito.")
        else:
            print("Fondos insuficientes o cantidad inválida.")

    def get_saldo(self):
        return self.__saldo
```

```python
cuenta = CuentaBancaria("Loreto",1500)

print(cuenta.get_titular()) #Loreto
cuenta.depositar(200) #Depósito realizado con éxito.
print(cuenta.get_saldo()) #1700
```

Ejercicios de clase.

- Modificar los 3 ejercicios de clase anteriores para que los atributos de la clase padre sean privados.

- Crear métodos **get** y **set** para acceder a esos atributos y para poder modificar su valor.

3

Poliformismo

3. Polimorfismo

- El **polimorfismo** es la característica de POO que permite a diferentes clases heredadas de una misma clase implementar un mismo método de la clase padre con diferentes funcionalidades.

- Esto quiere decir que un mismo método se puede adaptar a objetos diferentes y darle la funcionalidad que requiera cada objeto.

3. Polimorfismo

```python
class Vehiculo:
    def velocidad(self):
        pass

class Avion(Vehiculo):
    def velocidad(self):
        print("La velocidad maxima es de 1000 km/h")

class Coche(Vehiculo):
    def velocidad(self):
        print("La velocidad maxima es de 250 km/h")

class Barco(Vehiculo):
    def velocidad(self):
        print("La velocidad maxima es de 50 mph")


vehiculos = [Avion(),Coche(),Barco()]

for vehiculo in vehiculos:
    vehiculo.velocidad()

"""La velocidad maxima es de 1000 km/h
La velocidad maxima es de 250 km/h
La velocidad maxima es de 50 mph"""
```

Ejercicios de clase.

- Ejercicio 1:
  - Definir un método para calcular el área.
  - Clases hijas:
    - Sobrescribir el método de calcular área.

Ejercicios de clase.

- Ejercicio 2:
  - Definir un método apagar y otro encender.
  - Clases hijas:
    - Sobrescribir los métodos apagar y encender en cada clase hija.

Ejercicios de clase.

- Ejercicio 3:
  - Clases hijas:
    - Sobrescribir los métodos arrancar, acelerar y frenar. Utilizando la velocidad que se declaró en el constructor de Vehículo.

¡Gracias!