poo-pt-2-teoria

December 22, 2023

#

09 - PROGRAMACIÓN ORIENTADA A OBJETOS - PARTE 2

Anteriormente vimos que los objetos son una forma de agrupar datos y funciones que operan sobre esos datos. En esta parte terminaremos de ver los conceptos clave de la programación orientada a objetos: herencia, polimorfismo y encapsulamiento.

0.1 1. Herencia

La herencia es una forma de crear nuevas clases usando clases que ya han sido definidas. Las clases recién formadas se denominan clases derivadas, las clases de las que se derivan se denominan clases base. Las clases derivadas heredan los atributos y métodos de las clases base, y puede agregar más atributos o métodos propios.

Por ejemplo, vamos a crear una clase `Persona` que tenga como atributos el nombre, la edad y el lugar de residencia. Además, queremos que tenga un método que muestre por pantalla los datos de la persona. Para ello definimos el método `descripcion`:

[ ]:
```python
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
    def descripcion(self):
        print(f"Nombre: {self.nombre}, edad: {self.edad}")
```

Ahora, para aplicar la herencia, vamos a crear una clase `Alumno` que deriva de la clase `Persona`. La sintaxis para crear una clase derivada es la siguiente:

```python
class NuevaClase(ClaseBase):
    # Definición de la clase
```

[ ]:
```python
class Alumno(Persona):
    pass
```

[ ]:
```python
persona_1 = Persona("Agustina", 37)
alumno_1 = Alumno("Pablo", 13)

persona_1.descripcion()
print("---------------------------")
alumno_1.descripcion()
```

Vemos como la clase derivada tiene como clase base a la clase `Persona`. Esto significa que la clase `Alumno` hereda todos los atributos y métodos de la clase `Persona`. Además, podemos añadir nuevos atributos y métodos a la clase derivada. Por ejemplo, vamos a añadir un nuevo atributo `carrera` y un método `estudiar`:

[ ]:
```
class Alumno(Persona):
    def __init__(self, nombre, edad, curso):
        self.nombre = nombre
        self.edad = edad
        self.curso = curso
    def estudiar(self):
        print(f"{self.nombre} está estudiando")
```

[ ]:
```
alumno_2 = Alumno("Pablo", 13, "1°")
alumno_2.descripcion()
alumno_2.estudiar()
```

Podemos evitar escribir el constructor `__init__()` de la clase derivada, ya que automáticamente hereda el constructor de la clase base. Sin embargo, si queremos añadir algún atributo nuevo, o modificar el valor de algún atributo heredado, debemos escribir un nuevo constructor. Para ello tenemos dos alternativas:

Utilizar el constructor de la clase base para inicializar los atributos heredados y luego inicializar el nuevo atributo:

[ ]:
```
class Alumno(Persona):
    def __init__(self, nombre, edad, curso):
        Persona.__init__(self, nombre, edad)
        self.curso = curso

alumno_3 = Alumno("Pablo", 13, "1°")
alumno_3.descripcion()
alumno_3.curso
```

La otra alternativa es utilizar la función `super()` que nos permite invocar un método de la clase base. En este caso, podemos invocar el constructor de la clase base para inicializar los atributos heredados y luego inicializar el nuevo atributo:

[ ]:
```
class Alumno(Persona):
    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)        # self no es necesario
        self.curso = curso

alumno_4 = Alumno("Ana", 17, "5°")
alumno_4.descripcion()
alumno_4.curso
```

Gracias a la herencia podemos reutilizar el código de la clase base y además añadir nuevo código en la clase derivada. Así como reutilizamos el constructor de la clase base, también podemos reutilizar

los métodos de la clase base. Por ejemplo, vamos a modificar el método `descripcion()` de la clase `Alumno` para que muestre por pantalla los datos del alumno y el curso en el que está:

[ ]:
```python
class Alumno(Persona):
    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)
        self.curso = curso
    def descripcion(self):
        super().descripcion()
        print(f"Curso: {self.curso}")

alumno_5 = Alumno("Roberto", 15, "3°")
alumno_5.descripcion()
```

En este caso, hemos reutilizado el método `descripcion()` de la clase base `Persona` y hemos añadido el curso en el que está el alumno. Sin embargo, también podemos sobrescribir el método de la clase base, es decir, podemos redefinir el método en la clase derivada.

0.2 2. Polimorfismo

El polimorfismo es la capacidad de un objeto para tomar varias formas. En Python, esto significa que un objeto puede pasar por diferentes tipos. El polimorfismo nos permite definir métodos en la clase base que serán sobreescritos en las clases derivadas.

Por ejemplo, vamos a definir una clase `Animal` que tenga un método `hacer_sonido()`, y luego definiremos dos clases derivadas `Perro` y `Gato` que sobrescribirán el método `hacer_sonido()`:

[ ]:
```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "Woof!"

class Gato(Animal):
    def hacer_sonido(self):
        return "Meow!"

def funcion_polimorfica(animal):
    print(animal.hacer_sonido())

mi_perro = Perro("Toby")
mi_gato = Gato("Michi")

funcion_polimorfica(mi_perro)
```

```
funcion_polimorfica(mi_gato)
```

Ambas clases derivadas tienen el método `hacer_sonido()`, sin embargo, cada uno de ellos tiene un comportamiento diferente. Esto es lo que se conoce como polimorfismo, un mismo método se comporta de forma diferente en cada clase derivada.

0.3 3. Encapsulamiento

El encapsulamiento es un mecanismo que permite restringir el acceso a los métodos y atributos de la clase. Esto permite proteger los datos, para evitar que sean modificados por error. Sin embargo, aún es posible acceder a ellos mediante métodos de acceso (getters y setters). En Python no existen los modificadores de acceso como en otros lenguajes (public, private, protected), sin embargo, se puede acceder a los atributos protegidos y privados. Por convención, se utiliza un guión bajo antes del nombre del atributo para indicar que es protegido, y dos guiones bajos para indicar que es privado. Veamos un ejemplo:

[ ]:
```
class Auto:
    def __init__(self, marca, modelo, chasis):
        self.marca = marca
        self.modelo = modelo
        self.__chasis = chasis
    def __str__(self):
        return f"Marca: {self.marca}, modelo: {self.modelo}"
    def ver_chasis(self):
        return self.__chasis

auto_1 = Auto("Ford", "Fiesta", "123456")
print(auto_1)
```

En este caso, chasis es un atributo privado, y marca y modelo son atributos públicos. Podriamos acceder al chasis mediante el método `ver_chasis()` (el cual simula un getter):

[ ]:
```
print(auto_1.ver_chasis())
```

Sin embargo, si intentamos acceder al chasis directamente, nos dará un error:

[ ]:
```
print(auto_1.chasis)
print(auto_1.__chasis)
```

A pesar de esto, aún es posible acceder al atributo privado, aunque no es recomendable:

[ ]:
```
print(auto_1._Auto__chasis)
```

Por lo tanto, el encapsulamiento en Python no es tan estricto como en otros lenguajes, pero aún así nos permite proteger los datos de la clase en cierta medida.