Workshop de Programación – Pilares de la Programación Orientada a Objetos (POO)

1. Pregunta teórica

¿Cuáles son los 4 pilares de la programación orientada a objetos y qué aporta cada uno?

1. **Abstracción**
   Permite centrarse en los aspectos esenciales del objeto, ignorando los detalles innecesarios. Ayuda a manejar la complejidad del sistema.

2. **Encapsulamiento**
   Oculta el estado interno de un objeto y expone solo lo necesario a través de métodos públicos. Protege los datos y mejora la seguridad y mantenibilidad del código.

3. **Herencia**
   Permite crear nuevas clases basadas en clases existentes, reutilizando código y facilitando la extensión de funcionalidades.

4. **Polimorfismo**
   Permite que diferentes clases respondan al mismo mensaje (método) de diferentes formas. Favorece la flexibilidad del código.

Mini autoevaluación

1. ¿Cuál es la diferencia entre encapsulamiento y abstracción?

- *Encapsulamiento* oculta los detalles internos del objeto (por ejemplo, atributos privados).

- *Abstracción* muestra solo lo relevante, ocultando la complejidad del sistema.

2. ¿Qué pilar permite a las subclases sobrescribir métodos?

- *Herencia*, en combinación con *polimorfismo*, permite que las subclases redefinan métodos de la clase padre.

Código con errores para corregir

Código original:

```
class Dog:
   def __init__(self, name):
      name = name

   def speak(self):
      return "woof"

dog = Dog("Bobby")
print(dog.name)
```

Corrección:

```
class Dog:
   def __init__(self, name):
      self.name = name  # Se usa self para asignar al atributo del objeto

   def speak(self):
      return "woof"

dog = Dog("Bobby")
print(dog.name)  # Imprime: Bobby
```

Ejercicio

Jerarquía simple de vehículos:

```
class Vehiculo:
   def __init__(self, marca):
      self.marca = marca
```

```
    def info(self):
        return f"Vehículo de la marca {self.marca}"

class Coche(Vehiculo):
    def info(self):
        return f"Coche de la marca {self.marca}"

class Camion(Vehiculo):
    def info(self):
        return f"Camión de la marca {self.marca}"

def mostrar_info(vehiculo):
    print(vehiculo.info())

# Ejemplo de uso
mostrar_info(Coche("Toyota"))
mostrar_info(Camion("Volvo"))
```

5) Reflexión individual:
Siento que entiendo bien la herencia porque me ayuda a reducir código repetido. Me cuesta más el polimorfismo porque a veces no sé bien cómo aprovecharlo.

6) Desafío opcional – Encapsulamiento con getters y setters

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

    def get_nombre(self):
        return self.__nombre

    def set_nombre(self, nombre):
        self.__nombre = nombre

    def get_edad(self):
```

```
        return self.__edad

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

persona = Persona("Ana", 30)
print(persona.get_nombre())  # Ana
persona.set_nombre("Juan")
print(persona.get_nombre())  # Juan
```

Workshop de Programación – Decoradores en Python

1. Pregunta teórica

¿Qué es un decorador en Python y para qué se utiliza comúnmente?

Un **decorador** es una función que recibe otra función como argumento y retorna una nueva función que extiende o modifica el comportamiento de la original, sin cambiar su estructura interna. Se usan para reutilizar lógica de manera elegante y legible, por ejemplo: autorización, logging, control de acceso, validaciones, etc.

Mini autoevaluación

1. ¿Cómo se aplica a una función?
Mediante la sintaxis `@decorador` justo arriba de la definición de una función.

2. ¿Qué función interna suele tener un decorador?
Un decorador suele tener una función interna llamada `wrapper`, que encapsula la lógica adicional y llama a la función original.

Código con errores para corregir

Código original:

```
def decorator(func):
    print("Decorating...")
    return func

@decorator
def greet():
    print("Hi!")

greet()
```

Problema: El decorador imprime al definirse, no al ejecutarse, y no tiene wrapper.
Corrección:

```
def decorator(func):
    def wrapper():
        print("Decorating...")
        return func()
    return wrapper

@decorator
def greet():
    print("Hi!")

greet()
# Salida esperada:
# Decorating...
# Hi!
```

Ejercicio práctico

Decorador @authorize con control de acceso:

```
def authorize(func):
    def wrapper(user):
        if getattr(user, 'is_admin', False):
            return func(user)
        else:
            print("Acceso denegado")
    return wrapper

# Clase de ejemplo para probar
class User:
    def __init__(self, nombre, is_admin):
        self.nombre = nombre
        self.is_admin = is_admin

@authorize
def ver_datos(user):
    print(f"Bienvenido, {user.nombre}. Accediste a los datos.")

# Prueba
admin = User("Ana", True)
invitado = User("Luis", False)

ver_datos(admin)     # Bienvenido, Ana. Accediste a los datos.
ver_datos(invitado)  # Acceso denegado
```

Reflexión individual

¿Qué otras funciones de Python conocés que usan decoradores? ¿Te gustaría usarlos en tus propios proyectos?

Conozco decoradores como `@staticmethod` y `@property` porque los vi en ejemplos básicos. Me gustaría aprender a usarlos para hacer mi código más organizado y evitar repetir instrucciones, por ejemplo para verificar permisos o mostrar mensajes antes de ejecutar funciones.