OOP - Python

Jorge Mario Londoño Peláez & Varias AI

May 7, 2025

1 Repaso Conceptos OOP - Versión Python

1.1 Objetivos de Aprendizaje

Al finalizar este módulo, serás capaz de:

- Comprender los fundamentos de la Programación Orientada a Objetos
- Implementar clases y objetos en Python
- Aplicar los principios de encapsulamiento, herencia y polimorfismo
- Utilizar decoradores y propiedades para mejorar el diseño de clases
- Manejar excepciones en programas orientados a objetos
- Implementar patrones de diseño básicos

1.2 Qué es la programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que utiliza ”objetos” para diseñar aplicaciones. La POO ayuda a organizar el código de manera más modular, reutilizable y mantenible, especialmente en proyectos grandes.

Un objeto es una entidad que contiene datos (atributos) y código (métodos) que operan sobre esos datos. La POO se centra en la organización del código en torno a estos objetos, en lugar de funciones o lógica. Los principios clave de la POO son: encapsulamiento, herencia y polimorfismo.

1.3 Clases y Objetos

En Python, una clase es un plano para crear objetos. Define la estructura y el comportamiento que tendrán los objetos de esa clase. Un objeto es una instancia específica de una clase.

El siguiente es un ejemplo de creación de una clase y algunas instancias. `__init__` es el constructor de la clase, se invoca automaticamente al crear objetos y permite inicializar las variables de instancia.

Listing 1: Definición de una clase y creación de objetos

```
class Dog:
    def __init__(self, name, breed):
        self.__name = name
        self.__breed = breed

    def get_name(self):             # Metodo getter (accesor)
```

```
        return self.__name

    def bark(self):
        print("Woof!")

# Crear objetos (instancias) de la clase Dog
my_dog = Dog("Buddy", "Golden Retriever")
your_dog = Dog("Lucy", "Poodle")

print(my_dog.get_name())  # Output: Buddy
my_dog.bark()        # Output: Woof!
```

El parámetro `self`

En Python, el primer parámetro de un método de instancia es siempre `self`. Este parámetro es una referencia al objeto sobre el cual se invoca el método. Cuando se llama a un método en un objeto, Python automáticamente pasa el objeto como el primer argumento. Por convención, este parámetro se llama `self`, pero podría tener cualquier nombre.

1.4 Encapsulamiento

El encapsulamiento es el principio de ocultar los detalles internos de un objeto y exponer solo una interfaz para interactuar con él. En Python, se logra mediante el uso de atributos ”protegidos” (convención con un guión bajo ‘_‘) y ”privados” (convención con doble guión bajo ‘__‘). En realiad el prefijo ‘__‘ hace un ”name mangling” (alteración del nombre) para hacerlos más difíciles de acceder accidentalmente desde fuera de la clase, pero conociendo la versión alterada del nombre si es posible el acceso.

Listing 2: Ejemplo de encapsulamiento

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance  # Atributo "protegido"

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self._balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance()) # Output: 1500
```

Propiedades

El usar propiedades para acceder y modificar atributos ”protegidos” o ”privados” se considera una mejor práctica para lograr el encapsulamiento. Las propiedades permiten controlar el acceso a los atributos de forma más elegante y flexible.

Listing 3: Ejemplo de encapsulamiento con propiedades

```python
class BankAccount:
    def __init__(self, balance):
        self._balance = balance      # Atributo "protegido"

    @property
    def balance(self):               # Getter (propiedad)
        return self._balance

    @balance.setter
    def balance(self, value):        # Setter (propiedad)
        if value >= 0:
            self._balance = value
        else:
            print("Balance cannot be negative")

    def deposit(self, amount):
        self.balance += amount       # Usa la propiedad

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount   # Usa la propiedad
        else:
            print("Insufficient funds")

account = BankAccount(1000)
account.deposit(500)
print(account.balance)  # Accede a la propiedad (getter)
account.balance = 2000  # Modifica la propiedad (setter)
account.balance = -100  # Imprime "Balance cannot be negative"
```

Variables y métodos de clase. Funciones estáticas

Además de los atributos y métodos de instancia, los cuales se acceden por medio del `self`, las clases en Python pueden tener atributos y métodos de clase. Los atributos de clase son compartidos por todas las instancias de la clase, y los métodos de clase no tienen acceso directo a la instancia. El primer argumento de los métodos de clase es la referencia a la clase y por convención se nombra `cls`. Los métodos de clase se definen usando el decorador `@classmethod`.

Las funciones estáticas son funciones que por conveniencia se incluyen dentro de la definición de la clase, pero no tienen acceso directamente ni a las variables de clase y mucho menos a las de instancia. Las funciones estáticas no tienen un primer argumento predefinido (no reciben `self`, ni `cls`).

La palabra clave `cls`: Al igual que `self`, `cls` es una convención. Representa la clase misma, no una instancia de la clase. Se utiliza para acceder y modificar atributos de clase, o para crear instancias de la clase desde dentro de un método de clase.

Listing 4: Ejemplo de variables y métodos de clase. Métodos estáticos

```python
class MyClass:
    class_variable = "Soy una variable de clase"

    def __init__(self, instance_variable):
        self.instance_variable = instance_variable

    @classmethod
```

```
    def class_method(cls, arg):
        print(f"Metodo de clase. cls: {cls}, arg: {arg}")
        print(f"Accediendo a class_variable: {cls.class_variable}")
        # cls.class_variable = "Modificada desde class_method" #Podemos modificar.
        # return cls("nueva_instancia") # Podemos crear instancia.

    @staticmethod
    def static_method(arg):
        print(f"Metodo estatico. arg: {arg}")
        # No podemos acceder a cls.class_variable o self.instance_variable
            directamente
        # print(MyClass.class_variable)  # Necesitariamos el nombre de la clase

# Uso
MyClass.class_method("Hola")
MyClass.static_method("Mundo")

instance = MyClass("Instancia")
instance.class_method("Otra cosa")  # Tambien funciona desde la instancia
instance.static_method("Otro")
```

1.5 Herencia

La herencia permite crear nuevas clases (*subclases* o *clases derivadas*) basadas en clases existentes (*superclases*, *clases base*, o *clases padre*). Las subclases heredan los atributos y métodos de sus superclases, lo que promueve la reutilización de código. En caso de ser necesario la subclase puede sobre-escribir (*override*) un método definido en la clase padre, por ejemplo el método `speak` del siguiente ejemplo.

Listing 5: Ejemplo de herencia

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclasses must implement the speak method")

    def __str__(self):
        return f"Animal: name={self.name}"


class Dog(Animal):

    def __init__(self, name, breed):
        super().__init__(name)          # Reusa el constructor de la superclase
        self.breed = breed

    def speak(self):                    # Sobre-escribe el metodo del padre
        print("Woof!")

    def __str__(self):
        return f"{super().__str__()} , breed={self.breed}" # Reusa y amplia el
            metodo del padre


class Cat(Animal):
```

```
26
27      def speak(self):              # Sobre-escribe el metodo del padre
28          print("Meow!")
29
30  my_dog = Dog("Buddy", "Golden Retriever")
31  my_cat = Cat("Whiskers")
32
33  print(my_dog)  # Output: Animal: name=Buddy, breed=Golden Retriever
34  my_dog.speak() # Output: Woof!
35  my_cat.speak() # Output: Meow!
```

1.5.1 Funciones isinstance() e issubclass()

Las funciones ‘isinstance()‘ e ‘issubclass()‘ son herramientas útiles para la introspección de tipos en Python, especialmente en el contexto de la programación orientada a objetos. ‘isinstance(objeto, clase)‘ comprueba si un objeto es una instancia de una clase dada, o de una de sus subclases. Devuelve ‘True‘ si el objeto es una instancia de la clase o de una subclase, y ‘False‘ en caso contrario. Por otro lado, ‘issubclass(clase1, clase2)‘ verifica si ‘clase1‘ es una subclase de ‘clase2‘. Devuelve ‘True‘ si ‘clase1‘ hereda de ‘clase2‘ (directa o indirectamente), y ‘False‘ en caso contrario. Es importante destacar que ‘issubclass(clase, clase)‘ devuelve ‘True‘, ya que una clase se considera subclase de sí misma. Estas funciones son fundamentales para implementar comportamientos polimórficos de manera segura y para realizar comprobaciones de tipo en tiempo de ejecución, permitiendo que el código se adapte a diferentes tipos de objetos de forma flexible.

Listing 6: Ejemplos de isinstance() e issubclass()

```
1   class Animal:
2       pass
3
4   class Dog(Animal):
5       pass
6
7   class Cat(Animal):
8       pass
9
10  my_dog = Dog()
11
12  print(isinstance(my_dog, Dog))        # Output: True  (my_dog es una instancia de
        Dog)
13  print(isinstance(my_dog, Animal))     # Output: True  (my_dog es una instancia de
        Animal, a traves de Dog)
14  print(isinstance(my_dog, Cat))        # Output: False (my_dog no es una instancia
        de Cat)
15
16  print(issubclass(Dog, Animal))       # Output: True  (Dog es una subclase de Animal
        )
17  print(issubclass(Cat, Animal))       # Output: True  (Cat es una subclase de Animal
        )
18  print(issubclass(Animal, Dog))       # Output: False (Animal NO es una subclase de
        Dog)
19  print(issubclass(Dog, Dog))          # Output: True  (Dog es una subclase de si
        misma)
```

1.5.2 La función super()

La función `super()` se utiliza para acceder a métodos de la clase padre desde una clase hija. Esto es útil para extender o modificar el comportamiento de los métodos heredados. Al llamar a `super()`, se crea un objeto que representa la clase padre, lo que permite invocar sus métodos.

1.6 Clases Abstractas

Las clases abstractas son clases que no se pueden instanciar directamente. Se utilizan como plantillas para otras clases. En Python, se definen utilizando el módulo 'abc' (Abstract Base Classes). Los métodos abstractos deben ser implementados por las subclases concretas. Las clases abstractas sirven como contratos. Definen una interfaz común que todas las subclases deben implementar, garantizando que las subclases tengan ciertos métodos. Ayudan a diseñar jerarquías de clases más robustas y mantenibles.

Los métodos abstractos se decoran con `@abstractmethod`. Un método abstracto debe ser sobrescrito en las subclases concretas. Si no se sobrescribe, se producirá un error al intentar instanciar la subclase.

Una clase abstracta puede tener múltiples métodos abstractos, y también puede tener métodos concretos.

Listing 7: Ejemplo de clases abstractas

```
from abc import ABC, abstractmethod

class Shape(ABC):
    """Clase abstracta para representar figuras geometricas"""

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass


class Circle(Shape):
    """Clase concreta para representar circulos"""

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius * self.radius

    def perimeter(self):
        return 2.0 * 3.14159 * self.radius

class Rectangle(Shape):
    """Clase concreta para representar rectangulos"""

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
```

```
        return self.width * self.height

    def perimeter(self):
        return 2.0 * (self.width + self.height)


# shape = Shape() # Error: No se puede instanciar una clase abstracta
c = Circle(5)
print(c.area()) # Output: 78.53975
```

1.7 Polimorfismo

El polimorfismo (literalmente 'muchas formas') es la capacidad de objetos de diferentes clases de responder a una misma llamada a un método, cada uno a su manera. El polimorfismo se logra mediante la herencia y la sobreescritura de métodos.

Listing 8: Ejemplo de polimorfismo

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("Woof!")

class Cat(Animal):
    def speak(self):
        print("Meow!")

def animal_sound(animal):
    animal.speak()

my_dog = Dog()
my_cat = Cat()

animal_sound(my_dog) # Output: Woof!
animal_sound(my_cat) # Output: Meow!
```

1.8 Excepciones

Las excepciones son errores que ocurren durante la ejecución de un programa. Python permite manejar excepciones utilizando bloques 'try/except/finally/else'. También se pueden lanzar excepciones personalizadas utilizando 'raise'.

El bloque try encierra la porción de código que puede producir una excepción. Si se produce una excepción, el bloque except se ejecuta. Se pueden tener múltiples bloques except para atrapar distintos tipos de excepciones. El bloque else es opcional y se invoca si no se produce ninguna excepción. El bloque finally es opcional y se ejecuta siempre al final del bloque try. Se utilizan para la limpieza de recursos (por ejemplo, cerrar archivos, cerrar conexiones).

Listing 9: Ejemplo básico de manejo de excepciones

```
try:
    result = 10 / 0
except ZeroDivisionError:
```

```
    print("Error: Division by zero")

try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise ValueError("Age cannot be negative")
except ValueError as e:
    print(f"Error: {e}")
```

Listing 10: Ejemplo de manejo de excepciones con multiples excepciones else finally

```
try:
    # Codigo que podria lanzar una excepcion
    f = open("myfile.txt", "r")
    result = 10 / 0
except FileNotFoundError:
    print("El archivo no existe.")
except ZeroDivisionError:
    print("Error: Division por cero.")
else:
    # Se ejecuta si NO hubo excepcion
    print("La division se realizo correctamente.")
    print(f"Resultado: {result}")
finally:
    # Se ejecuta SIEMPRE (haya o no excepcion)
    if 'f' in locals() and not f.closed: #Verifica si esta definido y abierto.
        f.close()
    print("Bloque finally ejecutado.")
```

2 Diagramas y Visualizaciones

2.1 Diagramas de Clases

Los diagramas de clases son una herramienta fundamental para visualizar la estructura de un sistema orientado a objetos. A continuación se muestran algunos ejemplos:

Listing 11: Ejemplo de diagrama de clases en formato PlantUML

```
@startuml
class Animal {
    -name: str
    +speak(): void
}

class Dog {
    -breed: str
    +speak(): void
}

class Cat {
    -color: str
    +speak(): void
}

Animal <|-- Dog
Animal <|-- Cat
@enduml
```

2.2 Relaciones entre Clases

En OOP, las clases pueden tener diferentes tipos de relaciones:

- **Herencia (is-a)**: Una clase hereda de otra (ej: Dog es un Animal)
- **Composición (has-a)**: Una clase contiene instancias de otra (ej: Car tiene un Engine)
- **Asociación (uses-a)**: Una clase utiliza otra (ej: Student usa Book)
- **Agregación (part-of)**: Una clase es parte de otra (ej: Wheel es parte de Car)

3 Ejercicios de Repaso

3.1 Ejercicios Básicos

1. **Libro y Biblioteca**
   - Crea una clase `Libro` con atributos como `titulo`, `autor`, `isbn` y `precio`.
   - Implementa propiedades usando `@property` para acceder y modificar los atributos.
   - Crea una clase `Biblioteca` que pueda almacenar múltiples libros y realizar búsquedas.
   - Implementa métodos mágicos como `__str__` y `__repr__`.
2. **Sistema de Empleados**
   - Implementa una jerarquía de clases para diferentes tipos de empleados:
     - `Empleado` (clase base con nombre, salario base)
     - `Desarrollador` (con especialidad y nivel)
     - `Gerente` (con departamento y bonificación)
   - Implementa el cálculo de salario usando propiedades.
   - Utiliza métodos mágicos para operaciones comunes.
3. **Sistema de Vehículos**
   - Crea una clase abstracta `Vehiculo` con métodos abstractos.
   - Implementa subclases como `Coche`, `Bicicleta` y `Motocicleta`.
   - Utiliza decoradores para propiedades y métodos de clase.
   - Implementa métodos mágicos para operaciones comunes.

3.2 Ejercicios Intermedios

1. **Sistema de Reservas**
   - Crea un sistema de reservas para un hotel con:
     - `Habitacion` (número, tipo, precio)
     - `Cliente` (datos personales, historial de reservas)
     - `Reserva` (fechas, habitación, cliente)
   - Implementa manejo de excepciones personalizadas.

- Utiliza context managers para el manejo de recursos.
- Implementa validación de datos usando propiedades.

2. **Red Social Simple**

   - Implementa un sistema básico de red social con:
     - `Usuario` (perfil, amigos, publicaciones)
     - `Publicacion` (contenido, fecha, likes)
     - `Comentario` (texto, autor, fecha)
   - Utiliza colecciones para manejar las relaciones.
   - Implementa métodos mágicos para operaciones comunes.
   - Utiliza dataclasses para clases que principalmente almacenan datos.

3.3 Ejercicios Avanzados

1. **Sistema de Gestión de Proyectos**

   - Crea un sistema para gestionar proyectos con:
     - `Proyecto` (nombre, fecha inicio/fin, presupuesto)
     - `Tarea` (descripción, estado, asignado)
     - `Equipo` (miembros, roles)
   - Implementa el patrón Observer usando decoradores.
   - Utiliza el patrón Factory para crear diferentes tipos de tareas.
   - Implementa logging para seguimiento de cambios.

2. **Simulador de Banco**

   - Implementa un sistema bancario con:
     - `Cuenta` (abstracta) con subclases `CuentaCorriente` y `CuentaAhorro`
     - `Cliente` (con múltiples cuentas)
     - `Transaccion` (depósitos, retiros, transferencias)
   - Implementa manejo de excepciones personalizadas.
   - Utiliza el patrón Singleton para el registro de transacciones.
   - Implementa validación de datos usando propiedades.

3.4 Consejos para los Ejercicios

- Utiliza type hints para mejorar la legibilidad del código.
- Implementa pruebas unitarias usando `unittest` o `pytest`.
- Documenta tus clases y métodos usando docstrings.
- Utiliza dataclasses para clases que principalmente almacenan datos.
- Aprovecha las características específicas de Python como decoradores y métodos mágicos.

4 Buenas prácticas

4.1 Principios SOLID en Python

Los principios SOLID son igualmente aplicables en Python:

- **S**ingle Responsibility Principle (SRP): Una clase debe tener una única razón para cambiar.
- **O**pen/Closed Principle (OCP): Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación.
- **L**iskov Substitution Principle (LSP): Los objetos de una superclase deben poder ser reemplazados por objetos de sus subclases.
- **I**nterface Segregation Principle (ISP): Los clientes no deben depender de interfaces que no utilizan.
- **D**ependency Inversion Principle (DIP): Los módulos de alto nivel no deben depender de módulos de bajo nivel.

4.2 Patrones de Diseño en Python

- **Singleton**: Implementado usando módulos o decoradores.
- **Factory**: Utilizando funciones factory o clases abstractas.
- **Observer**: Implementado con decoradores o usando el módulo `observer`.
- **Strategy**: Utilizando funciones como objetos de primera clase.

4.3 Depuración de Código OOP en Python

- Utiliza el depurador de Python (`pdb`) o el de tu IDE.
- Implementa logging usando el módulo `logging`.
- Escribe pruebas unitarias para verificar el comportamiento.
- Utiliza type hints y herramientas de análisis estático como `mypy`.

5 Referencias

Real Python: Object-Oriented Programming (OOP) in Python
Real Python: Python's Instance, Class, and Static Methods Demystified
Pickl.ai: Beginner's Guide to OOPS Concepts in Python
ScholarHat: Oops Concepts in Python With Examples
Documentación oficial: Python documentation