OOP - Python

Jorge Mario Londoño Peláez & Varias AI

February 3, 2025

1 Repaso Conceptos OOP - Versión Python

1.1 Qué es la programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que utiliza ”objetos” para diseñar aplicaciones. 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.

1.2 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.

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

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

Listing 1: Definición de una clase y creación de objetos

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.

Componentes estáticos de una clase

Además de los atributos y métodos de instancia, las clases en Python pueden tener atributos y métodos estáticos. Los atributos estáticos son compartidos por todas las instancias de la clase, y los métodos estáticos no tienen acceso al objeto de instancia (`self`). Se definen usando el decorador `@staticmethod`.

```
class Circle:
    pi = 3.14159  # Variable estatica

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

    @classmethod
    def get_pi(cls): # Metodo estatico
        return cls.pi

    def area(self):
        return Circle.pi * self.radius * self.radius

print(Circle.get_pi()) # Output: 3.14159
c = Circle(5)
print(c.area()) # Output: 78.53975
```

Listing 2: Ejemplo de variables y métodos estáticos

En este ejemplo, `add` es un método estático que se puede llamar directamente en la clase `MathUtils`, sin necesidad de crear una instancia de la clase.

Variables y métodos estáticos: Las variables y métodos estáticos pertenecen a la clase en sí, no a las instancias de la clase. Se acceden utilizando el nombre de la clase.

```
class Circle:
    pi = 3.14159  # Variable estatica

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

    @classmethod
    def get_pi(cls): # Metodo estatico
        return cls.pi

    def area(self):
        return Circle.pi * self.radius * self.radius

print(Circle.get_pi()) # Output: 3.14159
c = Circle(5)
print(c.area()) # Output: 78.53975
```

Listing 3: Ejemplo de variables y métodos estáticos

1.3 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 ‘__‘).

```
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
```

Listing 4: Ejemplo de encapsulamiento

1.4 Herencia

La herencia permite crear nuevas clases (subclases) basadas en clases existentes (superclases). Las subclases heredan los atributos y métodos de sus superclases, lo que promueve la reutilización de código.

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

    def speak(self):
        pass

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

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

my_dog = Dog("Buddy")
my_cat = Cat("Whiskers")

my_dog.speak() # Output: Woof!
my_cat.speak() # Output: Meow!
```

Listing 5: Ejemplo de herencia

1.5 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.

```
from abc import ABC, abstractmethod

```

```python
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

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

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

Listing 6: Ejemplo de clases abstractas

1.6 Polimorfismo

El polimorfismo permite que objetos de diferentes clases respondan al mismo método de manera diferente. Esto se logra mediante la herencia y la sobreescritura de métodos.

```python
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!
```

Listing 7: Ejemplo de polimorfismo

1.7 Excepciones

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

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero")

```

```python
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 8: Ejemplo de manejo de excepciones