

OOP - Python

Jorge Mario Londoño Peláez & Varias AI

February 5, 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.

```
1 class Dog:
2     def __init__(self, name, breed):
3         self.__name = name
4         self.__breed = breed
5
6     def bark(self):
7         print("Woof!")
8
9 # Crear objetos (instancias) de la clase Dog
10 my_dog = Dog("Buddy", "Golden Retriever")
11 your_dog = Dog("Lucy", "Poodle")
12
13 print(my_dog.name)    # Output: Buddy
14 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.

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 '__').

```
1 class BankAccount:
2     def __init__(self, balance):
3         self._balance = balance # Atributo "protegido"
4
5     def deposit(self, amount):
6         self._balance += amount
7
8     def withdraw(self, amount):
9         if amount <= self._balance:
10             self._balance -= amount
11         else:
12             print("Insufficient funds")
13
14     def get_balance(self):
15         return self._balance
16
17 account = BankAccount(1000)
18 account.deposit(500)
19 print(account.get_balance()) # Output: 1500
```

Listing 2: Ejemplo de encapsulamiento

Componentes estáticos de una clase / variables y métodos de 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 de clase no tienen acceso directo al objeto de instancia (aunque pueden acceder a la clase misma a través de `cls`). Los métodos de clase se definen usando el decorador `@classmethod`.

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.

```
1 class Circle:
2     pi = 3.14159 # Variable estatica
3
4     def __init__(self, radius):
5         self.radius = radius
6
7     @classmethod
8     def get_pi(cls): # Metodo estatico
9         return cls.pi
10
11     def area(self):
12         return Circle.pi * self.radius * self.radius
13
14 print(Circle.get_pi()) # Output: 3.14159
15 c = Circle(5)
16 print(c.area()) # Output: 78.53975
```

Listing 3: 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 de clase: Las variables y métodos de clase pertenecen a la clase en sí, no a las instancias de la clase. Se acceden utilizando el nombre de la clase o a través del parámetro `cls` en un método de clase.

```
1 class Circle:
2     pi = 3.14159 # Variable estatica
3
4     def __init__(self, radius):
5         self.radius = radius
6
7     @classmethod
8     def get_pi(cls): # Metodo de clase
9         return cls.pi
10
11     def area(self):
12         return cls.pi * self.radius * self.radius
13
14 print(Circle.get_pi()) # Output: 3.14159
15 c = Circle(5)
16 print(c.area()) # Output: 78.53975
```

Listing 4: Ejemplo de variables y métodos estáticos

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.

```
1 class Animal:
2     def __init__(self, name):
3         self.name = name
4
5     def speak(self):
6         pass
7
8 class Dog(Animal):
9     def speak(self):
10         print("Woof!")
11
12 class Cat(Animal):
13     def speak(self):
14         print("Meow!")
15
16 my_dog = Dog("Buddy")
17 my_cat = Cat("Whiskers")
18
19 my_dog.speak() # Output: Woof!
20 my_cat.speak() # Output: Meow!
```

Listing 5: Ejemplo de herencia

1.4.1 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 class Animal:
2     def __init__(self, name):
3         self.name = name
4
5     def speak(self):
6         print("Generic animal sound")
7
8 class Dog(Animal):
9     def __init__(self, name, breed):
10         super().__init__(name) \# Llama al constructor de la clase Animal
11         self.breed = breed
12
13     def speak(self):
14         super().speak() \# Llama al metodo speak de la clase Animal
15         print("Woof!")
16
17 my_dog = Dog("Buddy", "Golden Retriever")
18 my_dog.speak()
19 # Output: Generic animal sound
20 #         Woof!
```

Listing 6: Ejemplo de uso de `super()`

En este ejemplo, `super().__init__(name)` llama al constructor de la clase `Animal` para inicializar el atributo `name`. `super().speak()` llama al método `speak` de la clase `Animal` antes de imprimir "Woof!".

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.

```
1 from abc import ABC, abstractmethod
2
3 class Shape(ABC):
4     @abstractmethod
5     def area(self):
6         pass
7
8 class Circle(Shape):
9     def __init__(self, radius):
10         self.radius = radius
11
12     def area(self):
13         return 3.14159 * self.radius * self.radius
14
15 # shape = Shape() # Error: No se puede instanciar una clase abstracta
16 c = Circle(5)
17 print(c.area()) # Output: 78.53975
```

Listing 7: 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 sobrescritura de métodos.

```
1 class Animal:
2     def speak(self):
3         pass
4
5 class Dog(Animal):
6     def speak(self):
7         print("Woof!")
8
9 class Cat(Animal):
10    def speak(self):
11        print("Meow!")
12
13 def animal_sound(animal):
14     animal.speak()
15
16 my_dog = Dog()
17 my_cat = Cat()
18
19 animal_sound(my_dog) # Output: Woof!
20 animal_sound(my_cat) # Output: Meow!
```

Listing 8: 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’.

```
1 try:
2     result = 10 / 0
3 except ZeroDivisionError:
4     print("Error: Division by zero")
5
6 try:
7     age = int(input("Enter your age: "))
8     if age < 0:
9         raise ValueError("Age cannot be negative")
10 except ValueError as e:
11     print(f"Error: {e}")
```

Listing 9: Ejemplo de manejo de excepciones

2 Referencias

Real Python: [Object-Oriented Programming \(OOP\) in Python](#)

Pickl.ai: [Beginner's Guide to OOPS Concepts in Python](#)

ScholarHat: [Oops Concepts in Python With Examples](#)