Programación Orientada a Objetos II y Herencias

Atributos y Métodos en Clases

En la Programación Orientada a Objetos (POO), los atributos y métodos son componentes esenciales de una clase. Estos elementos permiten definir las propiedades y comportamientos de los objetos creados a partir de la clase.

Atributos: Los atributos son las variables que representan las propiedades o características de una clase. Existen dos tipos principales de atributos:

Atributos de Instancia: Son específicos para cada instancia (u objeto) de una clase. Estos atributos se definen dentro del método constructor `__init__` de la clase y se asignan a la instancia utilizando la palabra clave `self`.

Por ejemplo:

```
class Perro:
    def __init__(self, nombre, raza):
       self.nombre = nombre
       self.raza = raza
```

En este caso, nombre y raza son atributos de instancia que pertenecen a cada objeto Perro.

Atributos de Clase: Son compartidos por todas las instancias de una clase. Estos atributos se definen directamente en la clase y no dentro del método constructor `__init__`.
Por ejemplo:

```
class Perro:
    especie = "Canis lupus familiaris"
    def __init__(self, nombre, raza):
        self.nombre = nombre
        self.raza = raza
```

Aquí, especie es un atributo de clase compartido por todos los objetos Perro.

Métodos: Los métodos son funciones definidas dentro de una clase que describen los comportamientos que los objetos de esa clase pueden realizar. Al igual que con los atributos, existen métodos especiales y comunes:

Métodos de instancia: Operan sobre instancias específicas de una clase (un objeto). Reciben el primer parámetro `self`, que hace referencia a la instancia que llama al método.
Por ejemplo:

```
class Perro:
   def __init__(self, nombre, raza):
      self.nombre = nombre
      self.raza = raza
   def ladrar(self):
      print(f"{self.nombre} está ladrando.")
```

En este ejemplo, ladrar es un método de instancia que hace que el perro ladre.

Métodos de clase: Utilizan el decorador `@classmethod` y reciben un parámetro `cls`, que hace referencia a la clase en sí misma, no a una instancia particular. Son utilizados para operar sobre los atributos de clase.

Por ejemplo:

```
class Perro:
   especie = "Canis lupus familiaris"
   def __init__(self, nombre, raza):
      self.nombre = nombre
      self.raza = raza
   @classmethod
   def obtener_especie(cls):
   return cls.especie
```

Aquí, `obtener_especie` es un método de clase que devuelve el atributo de clase especie.

Métodos estáticos: Utilizan el decorador `@staticmethod` y no reciben ni `self` ni `cls`. Son métodos que no dependen de la instancia ni de la clase y generalmente se utilizan para realizar funciones auxiliares.

Por ejemplo:

```
class Perro:
   def __init__(self, nombre, raza):
      self.nombre = nombre
      self.raza = raza
@staticmethod
   def es_perro_valido(nombre):
   return len(nombre) > 0
```

En este caso, `es_perro_valido` es un método estático que verifica si el nombre de un perro es válido.

Ejemplo Completo:

```
class Perro:
    especie = "Canis lupus familiaris"

    def __init__(self, nombre, raza):
        self.nombre = nombre
        self.raza = raza

    def ladrar(self):
        print(f"{self.nombre} está ladrando.")

    @classmethod
    def obtener_especie(cls):
        return cls.especie

    @staticmethod
    def es_perro_valido(nombre):
        return len(nombre) > 0

# Crear una instancia de Perro
mi_perro = Perro("Firulais", "Labrador")
mi_perro.ladrar()  # Firulais está ladrando.
print(Perro.obtener_especie())  # Canis lupus familiaris
print(Perro.es_perro_valido("Firulais"))  # True
```

Los atributos y métodos permiten a las clases en Python ser flexibles y potentes, facilitando la creación de objetos complejos y bien organizados.

Encapsulamiento en POO

El encapsulamiento es una técnica de la Programación Orientada a Objetos (POO) que consiste en ocultar los detalles internos de una clase, de manera que los datos y métodos internos no sean accesibles directamente desde el exterior de la clase. En su lugar, se proporcionan métodos públicos para interactuar con los datos, garantizando un acceso controlado y seguro.

Cómo Proteger los Datos Internos de una Clase:

En Python, el encapsulamiento se puede implementar mediante la convención de usar un guion bajo () o doble guion bajo (_) al inicio del nombre de un atributo o método.

Atributos Privados:

- Guión Bajo Simple (_): Indica que el atributo o método es protegido, lo que significa que no debería ser accedido directamente desde fuera de la clase, aunque es posible hacerlo.
- Doble Guión Bajo (): Indica que el atributo o método es privado, lo que oculta su nombre fuera de la clase mediante un proceso llamado name mangling, ya que lo que hace internamente python es modificar el nombre del atributo privado para que quede oculto.

```
class Perro:
  def __init__(self, nombre, raza):
      self._nombre = nombre  # Atributo protegido
      self.__raza = raza      # Atributo privado

  def obtener_nombre(self):
      return self._nombre

  def obtener_raza(self):
      return self.__raza

perro1 = Perro("Aura", "caniche")
print(perro1.__raza) # Se produce un error porque no se puede acceder al
atributo privado desde afuera de la clase

print(perro1.obtener_raza()) # Salida: "caniche"
```

Métodos Privados:
Los métodos privados también se definen usando doble guion bajo (), impidiendo su acceso directo desde fuera de la clase.

```
class Perro:
  def __init__(self, nombre, raza):
      self._nombre = nombre
      self.__raza = raza

  def __ladrar(self):
      print(f"{self._nombre} está ladrando.")

  def mostrar_ladrar(self):
      self.__ladrar()
```

Implementación del Encapsulamiento en Python:

Para implementar el encapsulamiento de manera efectiva, se utilizan métodos "getter" y "setter" que permiten acceder y modificar los atributos privados de manera controlada.

Getters:
Los getters son métodos públicos que se utilizan para obtener el valor de un atributo privado.

```
class Perro:
  def __init__(self, nombre, raza):
      self.__nombre = nombre

  def obtener_nombre(self):
      return self.__nombre
```

Setters:
Los setters son métodos públicos que se utilizan para modificar el valor de un atributo privado.

```
class Perro:
  def __init__(self, nombre, raza):
      self.__nombre = nombre

  def establecer_nombre(self, nombre):
      self.__nombre = nombre
```

Ejemplo Completo:

```python
class Perro:
    def __init__(self, nombre, raza):
        self.__nombre = nombre  # Atributo privado
        self._raza = raza       # Atributo protegido

    def obtener_nombre(self):
        return self.__nombre

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

    def __ladrar(self):  # Método privado
        print(f"{self.__nombre} está ladrando.")

    def mostrar_ladrar(self):
        self.__ladrar()

# Crear una instancia de Perro
mi_perro = Perro("Firulais", "Labrador")
print(mi_perro.obtener_nombre())  # Firulais
mi_perro.establecer_nombre("Rex")
print(mi_perro.obtener_nombre())  # Rex
mi_perro.mostrar_ladrar()         # Rex está ladrando.
```

Importancia del Encapsulamiento:

El encapsulamiento es vital para proteger la integridad de los objetos. Al ocultar los datos internos y proporcionar métodos controlados para acceder y modificar estos datos, se evita la manipulación indebida y se garantiza que el objeto mantenga un estado coherente. Esto resulta especialmente importante en proyectos grandes y complejos, donde la protección de los datos y la consistencia del comportamiento de los objetos son cruciales.

Herencia en Python

La herencia es un mecanismo de la Programación Orientada a Objetos (POO) que permite a una clase (llamada clase hija o subclase) heredar métodos y atributos de otra clase (llamada clase padre o superclase). Esto facilita la reutilización del código y permite crear jerarquías de clases de manera eficiente.

Cómo Funciona la Herencia:

Cuando una subclase hereda de una superclase, adquiere todos sus métodos y atributos, pudiendo también añadir nuevos métodos y atributos, o redefinir (sobreescribir) los existentes. Esto permite que las subclases especialicen el comportamiento de la superclase.

Ejemplo Práctico:

A continuación se muestra un ejemplo de cómo crear una clase Animal y dos subclases, Perro y Gato, que heredan de Animal.

Definición de la Clase Padre (Animal):

```
class Animal:
   def __init__(self, especie, edad):
      self.especie = especie
      self.edad = edad
   def describeme(self):
      print(f"Soy un {self.especie} y tengo {self.edad} años.")
```

Definición de la Subclase Perro:

```
class Perro(Animal):  # En el parámetro de la clase se coloca la superclase

    def __init__(self, especie, edad, nombre):
        super().__init__(especie, edad)  # El método super nos permite
apuntar a la superclase
        self.nombre = nombre

    def hablar(self):
        print(f"{self.nombre} dice: ¡Guau!")

    def moverse(self):
        print(f"{self.nombre} está caminando sobre cuatro patas.")
```

Definición de la Subclase Gato:

```
class Gato(Animal):

    def __init__(self, especie, edad, nombre):
        super().__init__(especie, edad)
        self.nombre = nombre
```

```
    def hablar(self):
        print(f"{self.nombre} dice: ¡Miau!")

    def moverse(self):
        print(f"{self.nombre} está caminando sobre cuatro patas.")
```

Uso de las Clases y Subclases:

```
# Crear instancias de Perro y Gato
mi_perro = Perro("Canis lupus familiaris", 5, "Firulais")
mi_gato = Gato("Felis catus", 3, "Misu")

# Usar los métodos heredados y sobrescritos
mi_perro.describeme()  # Soy un Canis lupus familiaris y tengo 5 años.
mi_perro.hablar()      # Firulais dice: ¡Guau!
mi_perro.moverse()     # Firulais está caminando sobre cuatro patas.

mi_gato.describeme()   # Soy un Felis catus y tengo 3 años.
mi_gato.hablar()       # Misu dice: ¡Miau!
mi_gato.moverse()      # Misu está caminando sobre cuatro patas.
```

Beneficios de la Herencia:

Reutilización del Código: La herencia permite reutilizar el código de la clase padre en las clases hijas, lo que reduce la duplicación y facilita el mantenimiento del código.

Especialización: Las subclases pueden añadir o modificar funcionalidades de la superclase, permitiendo especializar el comportamiento según sea necesario.

Organización: Facilita la organización del código en jerarquías lógicas, mejorando la claridad y comprensión del mismo.

Conclusión:

La herencia es un concepto poderoso en la POO que permite crear estructuras de clases más flexibles y mantenibles. A través de la herencia, se pueden construir sistemas complejos de manera más eficiente, aprovechando la reutilización del código y la especialización de las subclases.

Herencia Múltiple en Python

La herencia múltiple es una característica de la Programación Orientada a Objetos (POO) que permite a una clase heredar de más de una clase padre. Esto significa que una subclase puede incorporar atributos y métodos de múltiples superclases. Aunque es una herramienta poderosa, también puede generar complejidades adicionales, especialmente cuando las superclases tienen métodos o atributos con el mismo nombre.

Cómo Funciona la Herencia Múltiple:

En Python, la herencia múltiple se implementa simplemente listando múltiples superclases en la definición de la subclase. Sin embargo, esto introduce la necesidad de un mecanismo para resolver conflictos en la herencia, lo que se maneja mediante el Method Resolution Order (MRO).

Ejemplo Práctico:

A continuación se muestra un ejemplo de cómo crear una clase `Animal`, dos superclases `Mamifero` y `Volador`, y una subclase `Murcielago` que hereda de ambas.

1. **Definición de las Clases Padre:**

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


class Mamifero(Animal):
    def __init__(self, nombre, pelaje_color):
        super().__init__(nombre)
        self.pelaje_color = pelaje_color  # Atributo específico de Mamífero

    def caminar(self):
        print(f"{self.nombre} está caminando.")

    def hablar(self):
        print(f"{self.nombre} está haciendo un sonido de mamífero.")


class Volador(Animal):
    def __init__(self, nombre, envergadura_alas):
        super().__init__(nombre)
        self.envergadura_alas = envergadura_alas  # Atributo específico de
```

```
Volador

    def volar(self):
        print(f"{self.nombre} está volando.")

    def hablar(self):
        print(f"{self.nombre} está haciendo un sonido de ave.")
```

2. **Definición de la Subclase Murciélago:**

```
class Murcielago(Mamifero, Volador):
    def __init__(self, nombre, pelaje_color, envergadura_alas):
        # Llamando al constructor de Mamífero y Volador
        Mamifero.__init__(self, nombre, pelaje_color)
        Volador.__init__(self, nombre, envergadura_alas)

    def hablar(self):
        print(f"{self.nombre} está haciendo un sonido de murciélago.")
```

Uso de la Clase con Herencia Múltiple:

```
# Crear una instancia de Murciélago
bat = Murcielago("Bruce")

# Usar los métodos heredados
bat.caminar()  # Bruce está caminando.
bat.volar()    # Bruce está volando.

# Resolución de métodos con el mismo nombre
bat.hablar()   # Bruce está haciendo un sonido de murciélago.
```

Method Resolution Order (MRO):

El MRO es el orden en el que Python busca métodos y atributos en una jerarquía de herencia múltiple. Para determinar el MRO, se puede consultar utilizando el método `mro()` o el atributo `__mro__`.

```
# Consultar el MRO de la clase Murciélago
print(Murcielago.mro())
# [<class '__main__.Murcielago'>, <class '__main__.Mamifero'>, <class
'__main__.Volador'>, <class 'object'>]
```

En este ejemplo, Python buscará los métodos primero en `Murcielago`, luego en `Mamifero`, seguido por `Volador`, y finalmente en `object`.

Importancia del MRO:

El MRO es crucial para evitar ambigüedades y conflictos en la herencia múltiple. Garantiza que Python siga un orden predecible al buscar métodos y atributos, permitiendo a los desarrolladores comprender y controlar cómo se resuelven los métodos.

Conclusión:

La herencia múltiple es una característica avanzada de la POO que permite una gran flexibilidad en el diseño de clases. Sin embargo, debe utilizarse con cuidado para evitar complejidades innecesarias. El MRO proporciona un mecanismo robusto para manejar los conflictos que puedan surgir en la herencia múltiple, asegurando que el comportamiento del programa sea consistente y predecible.

Polimorfismo en Python

El polimorfismo es un principio de la Programación Orientada a Objetos (POO) que permite que objetos de diferentes clases sean tratados como objetos de una clase común. Esto se logra mediante la definición de una interfaz común que es implementada por múltiples clases. El polimorfismo permite a un solo método operar de manera diferente según el tipo de objeto que lo invoque.

Aplicación del Polimorfismo:

El polimorfismo se aplica a través de la sobreescritura de métodos en las clases derivadas. Esto significa que diferentes clases pueden tener métodos con el mismo nombre, pero con implementaciones distintas. Cuando se llama a estos métodos, el comportamiento específico depende de la clase del objeto que lo llama.

Ejemplo Práctico:

A continuación se muestra un ejemplo de cómo crear una clase base `Animal` y dos subclases `Perro` y `Gato`, que implementan un método `hablar` de manera diferente.

1. **Definición de la Clase Base (Animal):**

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

```
    def hablar(self):
        raise NotImplementedError("La subclase debe implementar este
método.")
```

`raise` se usa en Python para lanzar una excepción de manera manual. Es decir, sirve para interrumpir la ejecución normal del programa cuando ocurre un error, o para forzar un error si se da cierta condición.

2. **Definición de la Subclase Perro:**

```
class Perro(Animal):
    def hablar(self):
        return f"{self.nombre} dice: ¡Guau!"
```

3. **Definición de la Subclase Gato:**

```
class Gato(Animal):
    def hablar(self):
        return f"{self.nombre} dice: ¡Miau!"
```

Uso del Polimorfismo:

```
# Crear instancias de Perro y Gato
animales = [Perro("Firulais"), Gato("Misu")]

# Iterar sobre la lista de animales y llamar al método hablar
for animal in animales:
    print(animal.hablar())

# Output:
# Firulais dice: ¡Guau!
# Misu dice: ¡Miau!
```

Beneficios del Polimorfismo:

Flexibilidad y Reutilización del Código:

- Permite escribir funciones y métodos que pueden operar con objetos de diferentes tipos.
- Facilita la extensión y modificación del código sin alterar su estructura básica.

Mantenimiento Simplificado:

- Al permitir que una interfaz común maneje diferentes tipos de objetos, se simplifica el mantenimiento y la comprensión del código.

Interfaz Común y Comportamiento Diferente:

El polimorfismo permite tratar diferentes objetos de la misma manera a través de una interfaz común, pero con comportamientos distintos. Esto es útil en situaciones donde se necesita aplicar el mismo conjunto de operaciones a una colección de objetos heterogéneos.

Conclusión:

El polimorfismo es una característica poderosa de la POO que permite que diferentes clases implementen métodos con el mismo nombre pero con comportamientos distintos. Este principio facilita la creación de código flexible y reutilizable, mejorando la mantenibilidad y extensibilidad de las aplicaciones.

Ejercicios para analizar:

Ejercicio 1. Gestión básica de entradas de blog

Enunciado:
Crear una clase `BlogPost` que represente una entrada de blog con los siguientes requerimientos:

1. **Atributos de instancia**:
   - `title` (título de la entrada).
   - `content` (contenido completo de la entrada).
   - `author` (autor de la entrada).

2. **Atributo de clase**:
   - `SITE_NAME`, que almacene el nombre del sitio (por ejemplo, `"MiBlogWeb"`).

3. **Métodos de instancia**:
   - `get_summary(self, length=100)`: devuelve los primeros `length` caracteres de `content` seguidos de `"..."`, o todo el contenido si es más corto.
   - `to_html(self)`: retorna un string con un pequeño bloque HTML que muestra título (`<h2>`), autor y contenido (`<p>`).

4. **Método de clase**:
    - `get_site_name(cls)`: devuelve el valor de `SITE_NAME`.

5. **Método estático**:
    - `is_valid_title(title)`: retorna `True` si `title` no está vacío y tiene menos de 100 caracteres; de lo contrario, `False`.

Escribir también un bloque de código que:

- Cree dos instancias de `BlogPost`.
- Imprima su resumen.
- Genere y muestre el HTML completo.
- Verifique la validez de un título mediante el método estático.

Resolución

Ejercicio 2. Gestión de usuarios con encapsulamiento e herencia

Enunciado:
En una aplicación web necesitamos manejar usuarios de forma segura. Implementar:

1. **Clase base `User`** con:
    - Atributo público: `username`.
    - Atributo protegido (*protected*): `_email`.
    - Atributo privado: `__password`.
    - Un constructor que reciba `username`, `email` y `password`, y los asigne correctamente (recordá usar `self.__password` para el privado).

    - Métodos públicos:
        - `get_email(self)`: devuelve el correo.
        - `set_email(self, new_email)`: actualiza `_email`.
        - `check_password(self, pwd)`: retorna `True` si `pwd` coincide con `__password`.
        - `set_password(self, new_pwd)`: asigna un nuevo valor a `__password`, solo si el nuevo cumple cierto criterio (por ejemplo, que tenga al menos 6 caracteres). Si no cumple, imprimir un mensaje de error.

2. **Clase derivada `AdminUser`** que herede de `User` y agregue:

- Atributo adicional: `role = "admin"` (definido directamente en la clase).
- Método `delete_post(self, post_id)`: simula la eliminación de una entrada de blog imprimiendo `"El post con ID {post_id} ha sido eliminado por {self.username}"`.
- Método `promote_user(self, other_user)`: recibe otro objeto `User` y le asigna un nuevo role `"editor"`, pero solo si `other_user` es instancia de `User` y su username es distinto al del administrador.

3. Un bloque de prueba que:
   - Cree un `User` y un `AdminUser`.
   - Muestre cómo funciona la encapsulación: intentar acceder a `__password` directamente (debería fallar), y luego usar `check_password`.
   - Cambiar el correo de usuario a través de `set_email`.
   - Cambiar la contraseña (una válida y otra inválida).
   - Que el `AdminUser` intente promover al `User` y luego eliminar un post.

Resolución