Object-oriented programming

¿Qué es la POO - Programación Orientada a Objetos?

La **Programación Orientada a Objetos (POO)** es un paradigma de programación que se centra en el uso de "objetos" como unidades fundamentales de desarrollo de software. Estos objetos son entidades que encapsulan tanto datos, denominados **atributos**, como funciones o procedimientos, llamados **métodos**, que permiten manipular esos datos.

Conceptos Clave de la POO

1. **Clases**: Una clase es una plantilla o un plano a partir del cual se crean objetos. Define la estructura (atributos) y el comportamiento (métodos) que tendrán los objetos instanciados a partir de ella. Por ejemplo, una clase `Coche` podría tener atributos como `color`, `marca` y `modelo`, y métodos como `acelerar()` y `frenar()`.

2. **Objetos**: Un objeto es una instancia de una clase. Si consideramos la clase `Coche`, un objeto específico podría ser un `Coche` rojo de marca Toyota. Los objetos tienen sus propios valores para los atributos definidos en la clase.

3. **Encapsulamiento**: Este principio permite que los datos de un objeto estén protegidos y solo sean accesibles a través de sus métodos. Esto ayuda a mantener la integridad de los datos y a ocultar la complejidad del sistema.

4. **Herencia**: La herencia es un mecanismo que permite crear nuevas clases basadas en clases existentes. Esto facilita la reutilización de código, ya que la nueva clase (subclase) hereda los atributos y métodos de la clase original (superclase) y puede agregar o modificar comportamientos según sea necesario.

5. **Polimorfismo**: Este principio permite que diferentes clases sean tratadas como instancias de una misma clase a través de una interfaz común. Esto es útil para crear sistemas flexibles y extensibles, donde los métodos pueden operar en objetos de diferentes clases.

Ventajas de la POO

- **Modularidad**: La POO promueve la creación de módulos independientes que pueden ser desarrollados, probados y mantenidos de forma aislada.
- **Reutilización de Código**: Gracias a la herencia y la creación de clases base, los desarrolladores pueden reutilizar código existente, lo que reduce el tiempo de desarrollo y mejora la consistencia.
- **Mantenibilidad**: El encapsulamiento y la estructura organizada de la POO facilitan el mantenimiento y la actualización del código, ya que los cambios en una clase pueden realizarse sin afectar a otras partes del sistema.

Impacto en la Ingeniería de Software

La POO ha transformado la forma en que se desarrolla software, proporcionando una base sólida para diversas metodologías de desarrollo, como el desarrollo ágil y el enfoque basado en componentes. Además, muchos lenguajes de programación modernos, como Java, C++, Python y C#, están diseñados para soportar y aprovechar los principios de la programación orientada a objetos.

En resumen, la Programación Orientada a Objetos es un enfoque poderoso que permite a los desarrolladores crear software más organizado, reutilizable y mantenible. Al centrarse en la creación y manipulación de objetos, la POO no solo mejora la eficiencia del desarrollo, sino que también fomenta mejores prácticas en la ingeniería de software.

Principios Fundamentales de la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) se basa en cuatro principios fundamentales que ayudan a estructurar y organizar el código de una manera más intuitiva y modular:

1. **Abstracción de Datos**: La capacidad humana de abstraer información concreta en conceptos más amplios es esencial para la programación. Por ejemplo, al pensar en un "coche", imaginamos un vehículo con características específicas como cuatro ruedas, un volante y un motor. En POO, el concepto abstracto se refiere a una **clase**, mientras que las realizaciones concretas de esa clase son los **objetos** o **instancias**. Así, tu coche específico es un objeto, una instancia de la clase más general "Coche".

2. **Encapsulamiento**: Cada objeto en POO contiene sus propios datos (atributos) y puede ser manipulado a través de acciones específicas (métodos). Por ejemplo, un objeto `Coche` podría tener atributos como `presión_de_neumáticos`, `color` y

`nivel_de_combustible`. Estos atributos están encapsulados dentro del objeto, lo que significa que son accesibles y modificables solo a través de métodos específicos de ese objeto. El encapsulamiento asegura que los datos estén contenidos y reduce la necesidad de listas o variables separadas para almacenar información relacionada.

3. **Herencia**: La herencia permite que una clase se especialice o derive de otra. La clase especializada hereda propiedades y comportamientos de una clase más general, conocida como la **superclase**. Por ejemplo, las clases `Coche` y `Camión` pueden heredar de la superclase `Vehículo`. Este enfoque promueve la reutilización de código al permitir que las subclases hereden atributos y métodos comunes de la clase superior. Python soporta "herencia múltiple", lo que significa que una clase puede heredar de múltiples clases padres simultáneamente.

4. **Polimorfismo**: El polimorfismo permite que objetos de diferentes clases sean tratados como objetos de una superclase común. Esto se logra definiendo métodos en la superclase y permitiendo que las subclases sobrescriban esos métodos con sus propias implementaciones. Por ejemplo, tanto las clases `Cuadrado` como `Círculo` pueden ser subclases de `Forma`, y cada una implementa un método común como `dibujar()`. Cuando llamas a este método en un objeto, se ejecuta la implementación específica de la subclase. Esta flexibilidad permite que las funciones acepten objetos de la superclase como parámetros, promoviendo la reutilización y adaptabilidad del código.

Estos cuatro principios forman la base de la POO, permitiendo un desarrollo de código más organizado, modular y flexible.

Clases e Instancias en Python

En Python, una **clase** sirve como un plano o plantilla para crear objetos, y cada objeto creado a partir de una clase se llama una **instancia**. Una clase define tanto la estructura (atributos) como el comportamiento (métodos) de los objetos que pertenecen a ella.

- **Definición de Clase**: Aquí hay un ejemplo básico de cómo definir una clase en Python:

```python
class Car:
    def __init__(self, color):
        self.color = color
        self.position = 0  # Inicializa la posición como un atributo

    def move(self):
        self.position += 1  # Incrementa el atributo de posición en 1
        return self.position  # Retorna la posición actualizada
```

En este ejemplo:

- Definimos una clase llamada `Car`, que sirve como el plano para crear objetos coche (instancias).

- El método `__init__` es un método especial llamado constructor. Inicializa los atributos del objeto cuando se crea. En este caso, establecemos el atributo `color` del coche y la `posición` a 0.
- El método `move` es una función simple definida dentro de la clase. Simula el movimiento del coche y devuelve una nueva posición.

Creando Instancias, una vez que hemos definido la clase, podemos crear instancias de esa clase, cada una representando un coche único:

```
bmw = Car("red")
```

After creating an instance, we can access its attributes and methods using dot notation:

```
position = bmw.move()  # Llama al método move()
car_color = bmw.color  # Accede al atributo color
```

En esta línea de código, creamos una instancia de la clase `Car` llamada `bmw` y especificamos su color como "rojo".

En este ejemplo, llamamos al método `move()` para simular el movimiento del coche y almacenar su posición. También accedemos al atributo de color para recuperar el `color` del coche.

Las clases y las instancias proporcionan una manera poderosa de modelar objetos del mundo real y sus comportamientos en Python, promoviendo la reutilización de código y la organización.

Resumen

1. **Programación Imperativa**:
   - La **programación imperativa** es un paradigma que se centra en describir cómo opera un programa paso a paso.
   - Hace énfasis en cambiar el estado del programa a través de una serie de declaraciones o comandos.
   - Las características clave incluyen:
     - **Variables**: Contenedores que almacenan datos y cuyo valor puede cambiar a lo largo de la ejecución del programa.
     - **Bucles**: Estructuras que permiten la repetición de un bloque de código mientras se cumpla una condición, facilitando tareas repetitivas.
     - **Condicionales**: Estructuras que permiten la ejecución de diferentes bloques de código en función de si se cumplen ciertas condiciones.
     - **Cambios de estado explícitos**: Modificaciones directas de las variables y el estado del programa a lo largo de su ejecución.
   - Los programas escritos en este paradigma son a menudo fáciles de entender secuencialmente, pero pueden volverse complejos con el aumento del tamaño y la complejidad. Esto puede llevar a un código difícil de mantener y entender, especialmente en sistemas grandes donde se requiere un seguimiento minucioso del estado de las variables.
2. **Programación Funcional**:

- La **programación funcional** es un paradigma que trata el cálculo como la evaluación de funciones matemáticas y evita cambiar el estado y los datos mutables.
- Enfatiza la inmutabilidad, funciones puras y expresiones declarativas.
- Las características clave incluyen:
  - **Funciones de orden superior**: Funciones que pueden recibir otras funciones como argumentos o devolverlas como resultados, permitiendo una gran flexibilidad en el diseño del programa.
  - **Funciones de primera clase**: Las funciones son tratadas como ciudadanos de primera clase, lo que significa que pueden ser asignadas a variables, pasadas como argumentos y retornadas desde otras funciones.
  - **Ausencia de efectos secundarios**: Las funciones no modifican el estado externo ni interactúan con datos mutables, lo que facilita la prueba y la razonabilidad de los programas.
- Los programas funcionales son a menudo concisos, mantenibles y más fáciles de razonar, dado que se centran en el "qué" hacer en lugar del "cómo" hacerlo, lo que reduce la complejidad en la gestión del estado.

3. **Programación Orientada a Objetos (POO)**:
   - La **Programación Orientada a Objetos (POO)** es un paradigma basado en el concepto de "objetos", que combinan datos (atributos) y comportamiento (métodos) en una sola unidad.
   - Enfatiza la modelación de entidades del mundo real y sus interacciones en el código.
   - Las características clave incluyen:
     - **Clases**: Plantillas que definen la estructura y comportamiento de los objetos. Las clases permiten crear objetos que comparten atributos y métodos.
     - **Objetos**: Instancias concretas de clases que contienen datos específicos y pueden interactuar con otros objetos.
     - **Herencia**: Permite a una clase heredar atributos y métodos de otra, promoviendo la reutilización de código y la creación de jerarquías de clases.
     - **Encapsulamiento**: Aísla el estado de un objeto y solo permite modificarlo a través de métodos, lo que protege la integridad de los datos.
     - **Polimorfismo**: Permite que diferentes objetos sean tratados como instancias de una misma clase a través de una interfaz común, lo que mejora la flexibilidad y la reutilización del código.
   - La POO promueve la modularidad, reutilización y abstracción, facilitando el mantenimiento y la escalabilidad del software, especialmente en sistemas grandes y complejos.

Imperativa vs. POO

1. **Enfoque en el Estado**:
   - La **programación imperativa** se centra en el control del flujo del programa mediante instrucciones secuenciales y cambios de estado explícitos. Los desarrolladores deben gestionar cómo se transforma el estado a lo largo de la

ejecución. En contraste, la **Programación Orientada a Objetos (POO)** encapsula el estado y el comportamiento dentro de objetos, permitiendo que las interacciones se realicen a través de métodos en lugar de manipular el estado directamente.

2. **Estructura del Código**:
   - En la programación imperativa, el código se organiza en procedimientos o funciones que operan sobre datos globales. Esto puede llevar a un código menos estructurado y más difícil de seguir a medida que aumenta su tamaño. Por otro lado, la POO utiliza clases y objetos para estructurar el código, lo que fomenta una organización más clara y modular, facilitando la reutilización y el mantenimiento.
3. **Abstracción**:
   - La programación imperativa proporciona abstracción a través de funciones, permitiendo agrupar lógica, pero puede resultar en un código menos intuitivo a medida que se complican los estados. La POO, en cambio, permite una abstracción más natural al modelar conceptos del mundo real con clases y objetos, mejorando la legibilidad y la alineación del código con el dominio del problema.
4. **Manejo de Datos**:
   - En la programación imperativa, los datos son a menudo manipulados directamente a través de variables globales y locales. Esto puede provocar efectos secundarios inesperados y hacer que el código sea más difícil de depurar. La POO, sin embargo, encapsula los datos dentro de objetos y proporciona métodos para interactuar con ellos, reduciendo la posibilidad de efectos secundarios y mejorando la claridad del flujo de datos.
5. **Reutilización de Código**:
   - La reutilización en la programación imperativa a menudo se logra mediante funciones que pueden ser llamadas en diferentes partes del código. Sin embargo, esto puede resultar en duplicación de lógica si no se gestiona adecuadamente. En la POO, la herencia y la composición permiten una reutilización de código más robusta, ya que las clases pueden heredar comportamientos y atributos de otras clases, reduciendo la duplicación y promoviendo la modularidad.
6. **Complejidad del Proyecto**:
   - La programación imperativa puede volverse compleja a medida que el proyecto crece, especialmente cuando se requiere un seguimiento del estado a través de múltiples funciones. Esto puede resultar en un código difícil de entender y mantener. En contraste, la POO organiza el código en objetos, facilitando la comprensión y el manejo de la complejidad a través de la encapsulación y la separación de responsabilidades.
7. **Manejo de Concurrencia**:
   - La programación imperativa a menudo utiliza técnicas como el control de acceso y la sincronización manual para manejar la concurrencia, lo que puede ser complicado y propenso a errores. La POO, al proporcionar objetos que pueden interactuar de manera independiente, puede facilitar el manejo de la concurrencia mediante la creación de objetos que gestionan su propio estado, promoviendo un diseño más robusto y menos propenso a errores.

En resumen, la programación imperativa y la POO son paradigmas distintos con diferentes enfoques y beneficios. La programación imperativa se centra en instrucciones secuenciales y cambios de estado, lo que puede complicar la gestión de la complejidad en proyectos grandes. En contraste, la POO promueve la encapsulación, la reutilización y una organización más intuitiva del código, lo que la hace más adecuada para modelar sistemas complejos y en constante evolución. La elección entre ambos paradigmas dependerá del dominio del problema, la naturaleza del proyecto y las preferencias del equipo de desarrollo.

Funcional vs. POO

1. **Gestión del Estado**:
   - La programación funcional típicamente evita el estado mutable y fomenta la inmutabilidad, lo que facilita razonar sobre el código. Esto significa que, una vez que se crea un valor, no puede ser cambiado, lo que ayuda a prevenir errores y facilita la depuración. En contraste, la POO a menudo implica cambiar los estados de los objetos, lo que puede complicar la comprensión del flujo del programa, especialmente en aplicaciones grandes donde múltiples objetos pueden interactuar entre sí.
2. **Estructuras de Datos**:
   - La programación funcional se apoya en estructuras de datos inmutables y funciones incorporadas para manipularlas. Esto permite operaciones más seguras y predecibles. Por otro lado, la POO define clases personalizadas que encapsulan tanto los datos como el comportamiento, lo que proporciona un enfoque más orientado a la modelación de entidades del mundo real.
3. **Abstracción**:
   - La POO se enfoca en modelar objetos del mundo real a través de clases, promoviendo una abstracción natural que permite reflejar la lógica del dominio en el código. En cambio, la programación funcional abstrae a través de funciones y transformaciones, lo que permite expresar operaciones de manera declarativa, centrándose en el qué en lugar del cómo.
4. **Herencia**:
   - La POO utiliza la herencia para definir relaciones jerárquicas entre clases, permitiendo la reutilización del código y la creación de especializaciones. En contraste, la programación funcional se basa en la composición de funciones y funciones de orden superior para combinar comportamientos, lo que fomenta la modularidad y la flexibilidad sin la necesidad de una jerarquía rígida.
5. **Efectos Secundarios**:
   - La programación funcional desalienta los efectos secundarios, promoviendo el uso de funciones puras que no alteran el estado externo ni dependen de él. Esto resulta en un código más predecible y fácil de probar. En cambio, la POO puede implicar efectos secundarios a través de llamadas a métodos y cambios de estado, lo que puede dificultar la prueba y la comprensión del código.
6. **Modularidad**:
   - Ambos paradigmas admiten la modularidad, pero la logran de maneras diferentes. La POO utiliza clases y objetos para encapsular comportamiento y datos, facilitando la organización del código en unidades manejables. La programación funcional, por otro lado, se basa en la composición de funciones,

donde pequeñas funciones puras se combinan para crear soluciones más complejas.

7. **Complejidad**:
   - La programación funcional a menudo resulta en código más conciso y menos complejo debido a la inmutabilidad y la pureza de las funciones, lo que reduce la cantidad de estado que el desarrollador debe gestionar. El código de la POO, sin embargo, puede volverse complejo con jerarquías de herencia profundas y múltiples interacciones entre objetos, lo que puede complicar el mantenimiento y la comprensión.

En resumen, la programación funcional y la POO son paradigmas distintos con diferentes filosofías y compensaciones. La programación funcional prioriza la inmutabilidad, funciones puras y expresiones declarativas, facilitando la razonabilidad y la concisión en el código. Por otro lado, la POO se enfoca en modelar entidades del mundo real usando clases y objetos, promoviendo la encapsulación y la reutilización. La elección entre ambos paradigmas depende del dominio del problema, los objetivos de diseño y las preferencias del equipo de desarrollo en un proyecto de software.

Un Ejemplo Específico de una Clase en Python

En la programación orientada a objetos (POO), una **clase** es como un plano o plantilla para crear objetos. Una **instancia** es un objeto específico creado a partir de una clase. Para entender mejor las instancias, tomemos el ejemplo de la clase incorporada `str`.

La Clase `str`:

- En Python, `str` es una clase incorporada que representa cadenas (secuencias de caracteres).
- La clase `str` define varios métodos y atributos que pueden usarse con objetos de cadena.

```
# Ejemplo 1: Creando Instancias
name = "Alfons"  # Creando una instancia de la clase str
text = "sdsfdsxg"  # Otra instancia de la clase str
```

Instancias de `str`:

- Cuando creas una cadena, como `name = "Alfons"` o `text = "sdsfdsxg"`, estás creando instancias de la clase `str`.
- Cada instancia de `str` (por ejemplo, "Alfons" o "sdsfdsxg") es un objeto específico con sus propios datos (los caracteres en la cadena) y comportamiento (métodos para la manipulación de cadenas).

Explorando Métodos y Atributos:

- Para explorar los métodos y atributos de una clase (incluidas las clases incorporadas como `str`), puedes usar la función `dir()`.
- Puedes llamar a `dir()` en un objeto o en la clase misma para ver qué métodos y atributos están disponibles.

```python
# Ejemplo 2: Explorando Métodos y Atributos
# Usando la función dir() para ver los métodos y atributos disponibles
print("Métodos y atributos de la clase str:")
print(dir(str))
```

```
Methods and attributes of the str class:
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix',
'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Métodos Dunder (Mágicos)

- En Python, los métodos y atributos con nombres encerrados en dobles guiones bajos, como `.__str__()` o `.__len__()`, se conocen como **métodos dunder** o **métodos mágicos**. Esta nomenclatura proviene de "double underscore" (doble guión bajo) y se utilizan para proporcionar funcionalidades especiales a los objetos.
- Los métodos dunder están destinados al uso interno de Python y se invocan automáticamente en ciertas situaciones, lo que permite una interacción más fluida con las construcciones del lenguaje. Por ejemplo, cuando usas la función `str()` para convertir un objeto a cadena, Python internamente llama al método dunder `.__str__()` si está definido para ese objeto.

Principales Métodos Dunder

Aquí hay una lista de algunos de los métodos dunder más comunes y su propósito:

- **`.__init__(self, ...)`**:
  - Se invoca automáticamente al crear una nueva instancia de una clase. Se utiliza para inicializar los atributos del objeto.
  - **Ejemplo**: ```python class Persona: def **init**(self, nombre, edad): self.nombre = nombre self.edad = edad

    p = Persona("Alice", 30) print(p.nombre) # Salida: Alice ```
- **`.__str__(self)`**:

- Se llama cuando usas la función `str()` o `print()` en un objeto. Debe devolver una cadena que represente al objeto de manera comprensible.
- **Ejemplo**: ```python class Persona: def **init**(self, nombre, edad): self.nombre = nombre self.edad = edad

```
  def __str__(self):
      return f"{self.nombre}, {self.edad} años"
```

p = Persona("Alice", 30) print(p) # Salida: Alice, 30 años ```

- **`.__len__(self)`**:
  - Se llama cuando se usa la función `len()` en una instancia, permitiendo definir cómo se mide la longitud del objeto.
  - **Ejemplo**: ```python class Grupo: def **init**(self): self.miembros = []

```
  def agregar(self, miembro):
      self.miembros.append(miembro)

  def __len__(self):
      return len(self.miembros)
```

g = Grupo() g.agregar("Alice") g.agregar("Bob") print(len(g)) # Salida: 2 ```

- **`.__getitem__(self, key)`**:
  - Permite acceder a los elementos de un objeto utilizando la notación de corchetes, como en `objeto[key]`.
  - **Ejemplo**: ```python class Coleccion: def **init**(self): self.elementos = ["a", "b", "c"]

```
  def __getitem__(self, index):
      return self.elementos[index]
```

c = Coleccion() print(c[1]) # Salida: b ```

- **`.__iter__(self)`**:
  - Permite que un objeto se convierta en un iterable, facilitando su uso en bucles `for` y otras construcciones que requieren iteración.
  - **Ejemplo**: ```python class Contador: def **init**(self, limite): self.limite = limite

```
  def __iter__(self):
      self.contador = 0
      return self

  def __next__(self):
```

```
        if self.contador < self.limite:
            self.contador += 1
            return self.contador
        raise StopIteration
```

for numero in Contador(5): print(numero) # Salida: 1, 2, 3, 4, 5 ```

- **`.__add__(self, other)`**:
  - Se llama cuando se usa el operador `+` para sumar dos objetos. Permite definir cómo se combinan los objetos de una clase específica.
  - **Ejemplo**: ```python class Numero: def **init**(self, valor): self.valor = valor

```
    def __add__(self, otro):
        return Numero(self.valor + otro.valor)
```

n1 = Numero(5) n2 = Numero(10) n3 = n1 + n2 print(n3.valor) # Salida: 15 ```

Ventajas de Usar Métodos Dunder

- **Personalización del Comportamiento**: Los métodos dunder permiten a los desarrolladores personalizar cómo sus objetos interactúan con funciones y operadores integrados, mejorando la usabilidad y la intuitividad.
- **Integración con Python**: Facilitan que las instancias de clases se comporten de manera coherente con las expectativas de los usuarios de Python, lo que puede hacer que las clases sean más fáciles de usar y entender.
- **Mejor Organización del Código**: Proporcionan un mecanismo para encapsular comportamientos específicos dentro de la clase, haciendo que el código sea más modular y mantenible.

Explorar Métodos Dunder

En general, las instancias son objetos específicos creados a partir de clases, y puedes explorar sus métodos y atributos disponibles usando `dir()`, que devuelve una lista de nombres de atributos y métodos que un objeto tiene. Esto te permite ver fácilmente qué métodos dunder y otros métodos están disponibles para una instancia específica.

- **Ejemplo de Uso de `dir()`**: `python p = Persona("Alice", 30) print(dir(p)) # Muestra todos los métodos y atributos de la instancia p`

Resumen

En resumen, los métodos dunder son métodos especiales en Python, reconocidos por sus dobles guiones bajos, y te permiten personalizar el comportamiento de tus clases en varias situaciones. Su correcta implementación puede hacer que tus objetos sean más intuitivos y fáciles de usar, alineándose con las expectativas del comportamiento de Python. Esto no solo

mejora la experiencia del usuario final, sino que también puede ayudar a reducir errores y aumentar la claridad del código.

```python
# Ejemplo 3: Métodos Dunder
# Definiendo una clase personalizada con métodos dunder
class CustomString:
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f"CustomString({self.value})"

    def __len__(self):
        return len(self.value)

# Creando instancias de la clase personalizada
custom_text = CustomString("Hello, World!")

# Usando la función str() para convertir un objeto a una cadena
# Internamente llama al método dunder __str__() si está definido
print(str(custom_text))  # Salida: "CustomString(Hello, World!)"

# Usando la función len() para obtener la longitud de un objeto
# Internamente llama al método dunder __len__() si está definido
print(len(custom_text))  # Salida: 13
```

```
CustomString(Hello, World!)
13
```

Definición de una clase

En Python, podemos definir nuestras propias clases para modelar entidades del mundo real o conceptos abstractos. Una clase es como un plano que define los atributos (datos) y métodos (funciones) que tendrán los objetos de esa clase. Una vez que una clase está definida, podemos crear instancias (objetos) de esa clase, cada una con su propio conjunto de atributos y la capacidad de realizar acciones definidas por los métodos de la clase.

Estructura de una Clase

Para crear una clase, usamos la palabra clave `class` y definimos los atributos y métodos dentro del bloque de la clase. A continuación se describen los componentes principales:

- **Atributos**: Son variables que almacenan datos específicos para cada instancia de la clase. Estos pueden ser de varios tipos, como cadenas de texto, números, listas, diccionarios, etc. Generalmente se definen dentro del método especial `.__init__()`, que se ejecuta automáticamente cuando se crea una nueva instancia. Los atributos permiten que cada objeto mantenga su propio estado.
- **Métodos**: Son funciones definidas dentro de la clase que permiten a las instancias realizar acciones u operaciones. Los métodos pueden acceder y modificar los

atributos de la instancia y pueden aceptar parámetros para realizar cálculos o acciones adicionales. Los métodos proporcionan comportamiento a los objetos y permiten la interacción con los datos almacenados.

Creación de Instancias

Cada vez que creas un objeto a partir de una clase, estás creando una **instancia** de esa clase. Cada instancia tiene su propia copia de los atributos definidos en la clase. Esto significa que puedes crear múltiples instancias de la misma clase, y cada una de ellas puede tener valores diferentes para sus atributos. La creación de instancias permite que los objetos compartan la misma estructura de clase, mientras que sus datos pueden variar.

Herencia de Clases

Python también admite la **herencia**, que permite que una clase derive de otra. Esto significa que una clase hija puede heredar atributos y métodos de una clase padre. La herencia promueve la reutilización del código y la creación de jerarquías de clases. Al heredar de una clase base, la subclase puede extender o modificar el comportamiento de la clase padre, lo que facilita la creación de clases especializadas a partir de clases más generales.

En resumen, las clases en Python son una herramienta poderosa para la creación de objetos y la estructuración del código. Definir una clase implica establecer atributos y métodos que modelan la funcionalidad y las características de las entidades que deseas representar. La posibilidad de crear instancias permite que múltiples objetos compartan la misma estructura pero tengan datos únicos, mientras que la herencia proporciona un mecanismo para construir sobre clases existentes y extender su funcionalidad. Esta flexibilidad es fundamental para la programación orientada a objetos en Python.

Vamos a explorar la definición e instanciación de clases con un ejemplo que involucra una clase `Teacher` para representar a los educadores en un sistema de gestión escolar.

```
# Definición de una clase - Teacher
class Teacher:
    # Método constructor (__init__) para inicializar atributos
    def __init__(self, name, subject):
        self.name = name
        self.subject = subject
        self.courses_taught = []

    # Método para agregar un curso impartido por el profesor
    def add_course(self, course_name):
        self.courses_taught.append(course_name)

    # Método para mostrar la información del profesor
    def display_info(self):
        print(f"Nombre del Profesor: {self.name}")
        print(f"Asignatura que Imparte: {self.subject}")
        print(f"Cursos Impartidos: {', '.join(self.courses_taught)}")
```

```python
# Creando instancias de la clase Teacher
teacher1 = Teacher("Alice", "Matemáticas")
teacher2 = Teacher("Bob", "Ciencias")

# Agregando cursos impartidos por los profesores
teacher1.add_course("Álgebra")
teacher2.add_course("Biología")
teacher2.add_course("Química")

# Mostrando la información del profesor
teacher1.display_info()
teacher2.display_info()
```

```
Teacher Name: Alice
Teaching Subject: Math
Courses Taught: Algebra
Teacher Name: Bob
Teaching Subject: Science
Courses Taught: Biology, Chemistry
```

Self

En las clases de Python, la palabra clave `self` juega un papel crucial. Se refiere a la instancia de la clase en sí y se utiliza para acceder a sus atributos y métodos. Aunque puede parecer un requisito adicional en comparación con otros lenguajes de programación, entender y usar `self` correctamente es fundamental para escribir código de Python orientado a objetos eficaz y mantenible.

Importancia de `self`

1. **Referencia a la Instancia**: `self` permite que los métodos de la clase hagan referencia a los atributos y métodos de la instancia específica. Esto es esencial para diferenciar entre atributos de instancia y variables locales que pueden tener el mismo nombre. Sin `self`, el método no sabría a qué atributo o método específico se está refiriendo, lo que podría causar confusión y errores.

2. **Encapsulación**: Al usar `self`, se fomenta la encapsulación, un principio clave de la programación orientada a objetos. Esto asegura que los atributos de una instancia estén protegidos y solo sean accesibles a través de métodos de esa instancia, lo que ayuda a mantener la integridad del estado del objeto.

3. **Consistencia**: El uso de `self` establece una convención clara y consistente en el código. Facilita la lectura y comprensión del código al permitir a otros desarrolladores identificar rápidamente que se está trabajando con atributos y métodos de una instancia, en lugar de variables locales o globales.

4. **Compatibilidad con Herencia**: En contextos de herencia, `self` asegura que las instancias de las clases hijas tengan acceso a los métodos y atributos de la clase padre. Esto es especialmente útil cuando se sobrescriben métodos, ya que `self` se refiere a la instancia más específica, permitiendo el comportamiento deseado.

Convenciones de Nomenclatura

Para mantener la consistencia y adherirse a las convenciones de nomenclatura de Python, es recomendable seguir recursos como la guía de estilo Python PEP 8, aquí. Esta guía proporciona pautas para nombrar variables, clases, funciones y más, lo que ayuda a mejorar la legibilidad del código y hace que sea más accesible para otros desarrolladores de Python. Por ejemplo, se recomienda usar nombres descriptivos para atributos y métodos que reflejen su propósito, lo que complementa el uso de `self`.

En resumen, `self` es un componente esencial de la programación orientada a objetos en Python. Su uso permite a los desarrolladores manejar la complejidad de las clases y sus instancias de manera efectiva. Comprender cómo y por qué utilizar `self` es crucial para escribir código claro y mantenible que aproveche al máximo las características de la programación orientada a objetos en Python.

Vamos a sumergirnos en el papel de `self` y cómo se utiliza dentro de las clases de Python.

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

    def bark(self):
        return f"{self.name} is barking!"

# Creando instancias de la clase Dog
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Max", "Labrador")

# Accediendo a las variables de instancia usando self
print(dog1.name)  # Salida: Buddy
print(dog2.breed)  # Salida: Labrador

Buddy
Labrador

# Llamando a un método usando self
print(dog1.bark())  # Salida: ¡Buddy está ladrando!
print(dog2.bark())  # Salida: ¡Max está ladrando!

Buddy is barking!
Max is barking!
```

En este ejemplo:

- Definimos una clase `Perro` con un método `__init__` que toma dos parámetros, `nombre` y `raza`, e inicializa las variables de instancia `self.nombre` y `self.raza`.
- El método `ladrar` usa `self` para acceder al atributo `nombre` de la instancia para generar un mensaje de ladrido.

- Creamos dos instancias de la clase `Perro`, `perro1` y `perro2`, y las inicializamos con diferentes nombres y razas.
- Accedemos a las variables de instancia (`nombre` y `raza`) y llamamos al método `ladrar` en cada instancia usando `self`.

Atributos Predeterminados

En las clases de Python, puedes inicializar atributos con valores predeterminados. Esto significa que si no se proporciona un valor específico al crear instancias, los atributos tendrán estos valores predeterminados. Esta característica te permite establecer valores iniciales que son sensatos y útiles, facilitando así la creación de objetos sin necesidad de especificar cada atributo en cada instancia.

Ventajas de los Atributos Predeterminados

1. **Facilidad de Uso**: Al definir atributos con valores predeterminados, los desarrolladores pueden crear instancias de clases de manera más sencilla y rápida. Esto es especialmente útil cuando la mayoría de las instancias comparten características comunes, permitiendo evitar la repetición de datos.
2. **Flexibilidad**: Aunque los atributos tienen valores predeterminados, los usuarios pueden sobrescribir estos valores al crear una nueva instancia. Esto proporciona flexibilidad, ya que permite personalizar la instancia según sea necesario, sin perder la funcionalidad de los valores predeterminados.
3. **Legibilidad**: Los atributos predeterminados pueden hacer que el código sea más legible, ya que clarifican las intenciones del desarrollador al establecer ciertas configuraciones iniciales. Esto ayuda a otros desarrolladores a entender rápidamente las expectativas y comportamientos de la clase.
4. **Seguridad**: Al proporcionar valores predeterminados, se reduce la probabilidad de que los atributos queden sin inicializar. Esto minimiza el riesgo de errores en tiempo de ejecución debido a referencias a variables no definidas.

En resumen, los atributos predeterminados en Python son una herramienta valiosa para mejorar la usabilidad, flexibilidad y claridad del código en la programación orientada a objetos. Al establecer valores predeterminados sensatos, se facilita la creación de instancias y se garantiza que los atributos tengan configuraciones iniciales apropiadas, lo que resulta en un código más robusto y fácil de mantener.

Ilustremos esto con un ejemplo:

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

    def bark(self):
        return f"{self.name} says Woof!"
```

```python
# Creando instancias con y sin valores de atributo
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog()  # Usando valores de atributo predeterminados

print(dog1.bark())  # Salida: "¡Buddy dice Woof!"
print(dog2.bark())  # Salida: "¡Desconocido dice Woof!"
```

```
Buddy says Woof!
Unknown says Woof!
```

En la clase `Dog`, proporcionamos valores predeterminados para `name` y `breed` en el método `__init__`. Al crear instancias como `dog2`, si no se especifican valores, se utilizan los valores predeterminados. Esta flexibilidad facilita el manejo de varios escenarios al crear objetos.

Métodos de Instancia

En Python, los **métodos de instancia** son funciones asociadas con una clase que pueden ser llamados en instancias (objetos) creados a partir de esa clase. Estos métodos permiten acceder y manipular los atributos de una instancia, convirtiéndolos en una herramienta poderosa para realizar acciones relacionadas con los datos de la clase.

Funcionamiento de los Métodos de Instancia

1. **Definición**: Los métodos de instancia se definen dentro de una clase y suelen incluir `self` como el primer parámetro. Este parámetro representa la instancia específica de la clase y permite que el método acceda a los atributos y otros métodos de esa instancia.

2. **Acceso a Atributos**: Al invocar un método de instancia, se puede acceder a los atributos de la instancia usando `self`. Esto permite modificar los atributos, realizar cálculos basados en sus valores o ejecutar cualquier lógica relacionada con la instancia.

3. **Llamadas a Métodos**: Los métodos de instancia pueden ser llamados utilizando la notación de punto sobre la instancia. Por ejemplo, si `mi_instancia` es un objeto de la clase que tiene un método llamado `mi_metodo`, se puede invocar de la siguiente manera: `mi_instancia.mi_metodo()`.

Importancia de los Métodos de Instancia

1. **Encapsulación de Comportamiento**: Los métodos de instancia permiten agrupar comportamientos relacionados dentro de la clase, promoviendo una estructura más organizada y modular. Esto facilita el mantenimiento del código, ya que los cambios en la lógica de un método no afectan a otras partes del código.

2. **Interacción con el Estado del Objeto**: Al manipular atributos de instancia, los métodos de instancia permiten que los objetos actúen sobre su propio estado. Esto es esencial en la programación orientada a objetos, donde cada objeto puede tener un comportamiento único basado en sus datos.

3. **Reutilización de Código**: Al definir funciones comunes como métodos de instancia, se reduce la duplicación de código. En lugar de repetir la misma lógica en múltiples lugares, se puede llamar al método desde diferentes partes del código, promoviendo así la reutilización.

4. **Polimorfismo**: Los métodos de instancia también permiten el uso de polimorfismo, donde diferentes clases pueden implementar métodos con el mismo nombre, pero con comportamientos diferentes. Esto es útil en estructuras de datos complejas y sistemas donde la flexibilidad y la extensibilidad son necesarias.

En resumen, los métodos de instancia son una parte fundamental de la programación orientada a objetos en Python. Proporcionan una manera de interactuar con los datos de una instancia, encapsulando comportamientos y permitiendo una estructura más organizada y mantenible. Su uso eficiente no solo mejora la legibilidad del código, sino que también facilita la reutilización y la implementación de prácticas de diseño de software más robustas.

Exploraremos el concepto de métodos de instancia con un ejemplo que involucra una clase `Staff`, que representa a los miembros del personal en una empresa ficticia. Definiremos atributos como `name`, `fname`, `email` y `time_off`, y crearemos métodos como `greetings` y `takes_days_off` para demostrar cómo funcionan los métodos de instancia dentro de una clase.

```
class Staff():

    # 1. Atributos
    def __init__(self, name, fname):
        self.name = name
        self.fname = fname

        self.email = (self.name + "." + self.fname +
"@ironhack.com").lower()

        self.time_off = 30

    # 2. Métodos

    def greetings(self):
        return f"¡Hola!!!!!!"

    def takes_days_off(self, n):
        self.time_off -= n

# Vamos a crear el primer empleado
Employee_1 = Staff("Alfons", "Marques")

# Vamos a acceder al atributo time_off
Employee_1.time_off
```

```
30
```

```
# Tomé un día de descanso
Employee_1.takes_days_off(1)

# ¿Cuántos tengo ahora?
Employee_1.time_off
```

```
29
```

```
# Vamos a hacer algo
Employee_1.greetings()
```

```
'Hello!!!!!!'
```

Los métodos de instancia son funciones dentro de las clases que pueden ahorrarnos mucho tiempo.

⚠ Especificar Variables al Instanciar, Ignorando los Valores Predeterminados

En Python, cuando instanciamos (creamos) objetos a partir de una clase, a menudo proporcionamos valores para los atributos para personalizar cada instancia. Sin embargo, puede haber casos en los que queramos utilizar los valores predeterminados definidos en el método `__init__` de la clase para ciertos atributos.

Comprendiendo la Instanciación

1. **Valores Predeterminados**: Cuando definimos una clase, podemos establecer valores predeterminados para los atributos en el método `__init__`. Esto significa que, si no se proporciona un valor al crear una nueva instancia, se utilizarán esos valores predeterminados. Esto es útil para simplificar la creación de objetos que generalmente comparten configuraciones similares.

2. **Sobrescribir Valores Predeterminados**: Al instanciar un objeto, se pueden proporcionar valores para uno o más atributos. Esto permite personalizar el comportamiento del objeto. Por ejemplo, se puede especificar un valor para un atributo específico y dejar que otros tomen sus valores predeterminados. Esta flexibilidad permite adaptar la instancia a necesidades específicas sin necesidad de definir cada atributo.

Es importante notar que al instanciar un objeto, puedes optar por no proporcionar un valor para un atributo en particular. En ese caso, se utilizará el valor predeterminado establecido en el método `__init__`. Esto resulta útil cuando solo necesitas personalizar algunos atributos, manteniendo otros con valores por defecto.

Ventajas de Esta Flexibilidad

1. **Simplicidad**: Permite crear instancias de manera más sencilla, ya que no es necesario proporcionar valores para todos los atributos cada vez que se crea un objeto.

2. **Consistencia**: Al utilizar valores predeterminados, se asegura que todos los objetos de una clase tengan un estado inicial válido, lo que puede evitar errores y problemas de lógica en el programa.

3. **Personalización Controlada**: Proporciona un balance entre personalización y simplicidad, permitiendo que los desarrolladores ajusten solo aquellos aspectos que realmente necesitan cambiar, mientras que otros atributos mantienen su configuración predeterminada.

En resumen, los métodos de instancia son esenciales para la programación orientada a objetos en Python, ofreciendo funcionalidad y ahorrando tiempo en el desarrollo. La capacidad de especificar valores al instanciar objetos, mientras se ignoran algunos atributos y se utilizan los valores predeterminados, brinda flexibilidad y conveniencia. Esta característica mejora la eficiencia en la creación de instancias y garantiza que los objetos mantengan un estado inicial coherente y predecible.

En el ejemplo proporcionado, tenemos una clase `Staff` que representa a los miembros del personal de una empresa. Esta clase tiene atributos como `name`, `fname`, `email` y `time_off`. También hemos definido dos métodos de instancia: `greetings` para un saludo amistoso y `takes_days_off` para gestionar solicitudes de días libres.

Presta atención a cómo podemos especificar variables concretas al instanciar objetos `Staff` mientras ignoramos los valores predeterminados para algunos atributos, como el número de días libres. Esta flexibilidad nos permite personalizar objetos según sea necesario, mientras confiamos en los valores predeterminados cuando es apropiado.

```
class Staff():

    # 1. Atributos
    def __init__(self, name, fname):
        self.name = name
        self.fname = fname

        self.email = (self.name + "." + self.fname +
"@ironhack.com").lower()

        self.time_off = 30

    # 2. Métodos

    def greetings(self):
        return f"¡Hola!!!!!!"

    def takes_days_off(self, days=1):
        """Por defecto: toma un día a menos que se pase otro valor"""
        if days <= self.time_off:
            self.time_off -= days
            return f"Tienes estos días restantes: {self.time_off}"
```

```
        else:
            return f"Solo tienes {self.time_off} días disponibles"
```

Cuidado

Al modificar una clase añadiendo nuevos atributos o métodos, es importante ser consciente de que los objetos existentes creados a partir de esa clase no tendrán automáticamente los nuevos atributos o métodos. Necesitarás recrear esos objetos para asegurar que se inicialicen con las actualizaciones.

Consideraciones al Modificar Clases

1. **Objetos Existentes**: Cuando defines una clase y creas instancias de esa clase, esos objetos se configuran de acuerdo con la definición de la clase en el momento de su creación. Si decides añadir nuevos atributos o métodos a la clase más tarde, las instancias ya creadas no se verán afectadas por esos cambios. Esto significa que no podrán acceder a los nuevos atributos o métodos a menos que se vuelvan a crear.

2. **Recreación de Objetos**: Para que los objetos reflejen las modificaciones realizadas en la clase, tendrás que instanciar de nuevo esos objetos. Esto implica llamar nuevamente al constructor de la clase, lo que creará una nueva instancia con la definición actualizada, incluidos los nuevos atributos y métodos.

3. **Impacto en el Estado**: Al recrear objetos, es importante considerar el estado de esos objetos. Si los objetos existentes tienen datos que deben preservarse, necesitarás implementar una lógica para transferir esos datos a las nuevas instancias. Esto puede ser complicado y debe ser manejado cuidadosamente para evitar la pérdida de información.

Mejores Prácticas

- **Planificación**: Antes de realizar cambios significativos en una clase, planifica cómo esos cambios afectarán a las instancias existentes. Esto te ayudará a anticipar posibles problemas y a decidir si es necesario recrear objetos.

- **Documentación**: Asegúrate de documentar cualquier cambio realizado en la clase, especialmente si estos cambios impactan en la manera en que se crean o manejan las instancias. La documentación clara puede ayudar a otros desarrolladores a comprender los cambios y a tomar decisiones informadas al utilizar la clase.

- **Pruebas**: Después de modificar una clase, realiza pruebas exhaustivas para asegurarte de que las nuevas funcionalidades se comportan como se espera y que no hay efectos secundarios no deseados en el comportamiento de los objetos existentes.

En resumen, al modificar clases en Python, es crucial ser consciente de cómo estos cambios afectan a las instancias existentes. Los nuevos atributos o métodos no se aplicarán automáticamente a los objetos ya creados, lo que requiere que se recréen para reflejar las actualizaciones. La planificación cuidadosa, la documentación y las pruebas son esenciales para

manejar estos cambios de manera efectiva y asegurar la integridad de los datos y la funcionalidad del código.

En el ejemplo proporcionado, tenemos una clase `Staff` que representa a los miembros del personal de una empresa. Inicialmente, la clase tiene atributos como `name`, `fname`, `email`, `time_off` y `hobbies`. También hemos definido métodos como `greetings`, `takes_days_off` y `hobbies_function`.

Toma nota de este punto importante, especialmente cuando hagas cambios en la estructura de una clase. Si añades nuevos atributos o métodos, cualquier objeto existente de esa clase no ganará automáticamente acceso a estos cambios. Recrear los objetos es necesario para incorporar las actualizaciones de manera efectiva.

```
class Staff():

    # 1. Atributos
    def __init__(self, name, fname):
        self.name = name
        self.fname = fname

        self.email = (self.name + "." + self.fname +
"@ironhack.com").lower()

        self.time_off = 30

        self.hobbies = []  # Inicializa la lista de hobbies

    # 2. Métodos

    def greetings(self):
        return f"¡Hola!!!!!!"

    def takes_days_off(self, days=1):
        """Por defecto: toma un día a menos que se pase otro valor"""
        if days <= self.time_off:
            self.time_off -= days
            return f"Tienes estos días restantes: {self.time_off}"
        else:
            return f"Solo tienes {self.time_off} días disponibles"

    def hobbies_function(self, hobbie):
        """Añade elementos a la lista de hobbies"""
        self.hobbies.append(hobbie)  # Agrega el hobbie a la lista
        return f"Mis hobbies son: {self.hobbies}"  # Devuelve la lista
de hobbies
```

Variables de Clase

En la programación orientada a objetos, las **variables de clase** son atributos compartidos por todas las instancias (objetos) de una clase. A diferencia de las **variables de instancia**, que son únicas para cada objeto, las variables de clase mantienen el mismo valor en todas las instancias de la clase.

Definición y Alcance

Las variables de clase se definen típicamente dentro de la clase, pero fuera de cualquier método de instancia. Esto significa que están asociadas con la clase en sí, no con instancias específicas de la clase. Como resultado, cualquier cambio realizado en una variable de clase afecta a todas las instancias creadas a partir de esa clase.

Esto contrasta con las variables de instancia, que son definidas dentro del método `__init__` y se inicializan con valores específicos para cada objeto. La modificación de una variable de instancia solo afectará al objeto en cuestión y no a los demás.

Propósitos y Usos

Las variables de clase son útiles para almacenar datos o propiedades que son comunes a todas las instancias de una clase, como:

- **Configuraciones**: Para almacenar parámetros de configuración que deben ser iguales para todas las instancias.
- **Valores Predeterminados**: Para definir valores que pueden ser utilizados como predeterminados por las instancias.
- **Constantes**: Para almacenar valores que no deberían cambiar durante la ejecución del programa.

Algunos ejemplos de uso de variables de clase incluyen:

- Contar el número total de instancias creadas de una clase.
- Almacenar configuraciones globales que se aplican a todos los objetos.

Acceso y Modificación

Las variables de clase se pueden acceder y modificar a través de la clase misma, así como a través de instancias. Sin embargo, es importante tener en cuenta que si una variable de clase es modificada a través de una instancia, se creará una variable de instancia con el mismo nombre, y esto no afectará la variable de clase original. Este comportamiento se debe a que Python asocia el nombre de la variable al objeto más cercano (la instancia en este caso) en lugar de la clase.

Convención de Nomenclatura

En Python, las variables de clase a menudo se denotan utilizando una convención de nomenclatura con letras mayúsculas para distinguirlas de las variables de instancia, que generalmente tienen nombres en minúsculas. Esto mejora la legibilidad del código y permite a los desarrolladores identificar fácilmente las variables de clase.

Es importante resaltar que al trabajar con variables de clase, se debe tener cuidado de no sobrescribir accidentalmente las variables de clase al definir variables de instancia con el mismo nombre.

Comprender el uso de **variables de clase** es esencial al diseñar y trabajar con clases en la programación orientada a objetos. Estas variables permiten compartir datos entre todas las instancias de una clase, lo que facilita la gestión de información común y contribuye a un diseño de software más eficiente y organizado. Con su correcta implementación, las variables de clase pueden mejorar la cohesión y la claridad de tu código, ayudando a mantener una estructura lógica en tus aplicaciones.

```
class Employee:
    # Variable de clase
    COMPANY = "XYZ Corporation"

    def __init__(self, name, emp_id):
        # Variables de instancia
        self.name = name
        self.emp_id = emp_id

    def get_employee_info(self):
        # Método para obtener información del empleado
        return f"Name: {self.name}, Employee ID: {self.emp_id},
Company: {Employee.COMPANY}"

# Crear instancias de la clase Employee
employee1 = Employee("John Doe", 1001)
employee2 = Employee("Alice Smith", 1002)

# Acceder y mostrar la variable de clase
print("Company:", Employee.COMPANY)
```

```
Company: XYZ Corporation
```

```
# Acceder y mostrar información específica de la instancia
print(employee1.get_employee_info())
print(employee2.get_employee_info())
```

```
Name: John Doe, Employee ID: 1001, Company: XYZ Corporation
Name: Alice Smith, Employee ID: 1002, Company: XYZ Corporation
```

En general, los atributos de clase no deben usarse, excepto para almacenar valores constantes.

Ejercicio: Creando una Clase para Registros de Estudiantes

En este ejercicio, exploraremos cómo usar clases en un contexto de análisis de datos. Imagina que trabajas en el departamento de análisis de datos de una universidad, y te han encargado gestionar y analizar los registros de los estudiantes. Para agilizar tu trabajo y mejorar la organización de los datos, decides crear una clase de Python llamada `Student` para representar registros individuales de estudiantes.

La clase `Student` encapsulará atributos de estudiantes como nombre, ID, cursos y calificaciones. Además, proporcionará métodos para calcular el GPA del estudiante, agregar cursos y mostrar la información del estudiante.

Al crear esta clase, demostrarás cómo las clases pueden ayudarte a organizar y analizar los datos de los estudiantes de manera eficiente. Este ejercicio es amigable para principiantes y tiene como objetivo ilustrar el uso práctico de las clases en un contexto de análisis de datos.

Ahora, sumergámonos en el código de Python para crear la clase `Student` y realizar varias operaciones en los registros de los estudiantes.

```
# Definir la clase Student
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id
        self.courses = {}

    def add_course(self, course_name, grade):
        self.courses[course_name] = grade

    def calculate_gpa(self):
        total_grade_points = sum(self.courses.values())
        num_courses = len(self.courses)
        if num_courses == 0:
            return 0.0
        else:
            return total_grade_points / num_courses

    def display_student_info(self):
        print(f"Nombre del Estudiante: {self.name}")
        print(f"ID del Estudiante: {self.student_id}")
        print("Cursos y Calificaciones:")
        for course, grade in self.courses.items():
            print(f"{course}: {grade}")
        print(f"GPA General: {self.calculate_gpa():.2f}")

# Ejemplo de código limpio:
class Student:
    def __init__(self, name: str, student_id: str):
        """
        Inicializa un objeto Student con el nombre y el ID del
estudiante.

        Args:
            name (str): El nombre del estudiante.
            student_id (str): El identificador único del estudiante.
        """
        self.name = name
        self.student_id = student_id
```

```
        self.courses = {}

    def add_course(self, course_name: str, grade: float):
        """
        Agrega un curso y su calificación al registro del estudiante.

        Args:
            course_name (str): El nombre del curso.
            grade (float): La calificación obtenida en el curso.
        """
        self.courses[course_name] = grade

    def calculate_gpa(self) -> float:
        """
        Calcula el GPA del estudiante basado en las calificaciones de
los cursos.

        Returns:
            float: El GPA calculado.
        """
        total_grade_points = sum(self.courses.values())
        num_courses = len(self.courses)
        if num_courses == 0:
            return 0.0
        else:
            return total_grade_points / num_courses

    def display_student_info(self):
        """
        Muestra información sobre el estudiante, incluyendo nombre,
ID, cursos y GPA.
        """
        print(f"Nombre del Estudiante: {self.name}")
        print(f"ID del Estudiante: {self.student_id}")
        print("Cursos y Calificaciones:")
        for course, grade in self.courses.items():
            print(f"{course}: {grade}")
        print(f"GPA General: {self.calculate_gpa():.2f}")

# Crear instancias de la clase Student
student1 = Student("Alice Smith", "S12345")
student2 = Student("Bob Johnson", "S67890")

# Agregar cursos y calificaciones para cada estudiante
student1.add_course("Math", 90)      # Agregar curso de Matemáticas
con calificación 90 para el estudiante 1
student1.add_course("History", 85)    # Agregar curso de Historia con
calificación 85 para el estudiante 1
student2.add_course("Science", 92)    # Agregar curso de Ciencias con
calificación 92 para el estudiante 2
```

```
student2.add_course("English", 88)     # Agregar curso de Inglés con calificación 88 para el estudiante 2

# Mostrar información de los estudiantes
student1.display_student_info()  # Mostrar información del estudiante 1
student2.display_student_info()  # Mostrar información del estudiante 2
```

```
Student Name: Alice Smith
Student ID: S12345
Courses and Grades:
Math: 90
History: 85
Overall GPA: 87.50
Student Name: Bob Johnson
Student ID: S67890
Courses and Grades:
Science: 92
English: 88
Overall GPA: 90.00
```

Antes de continuar, respiremos y revisemos el vocabulario

- **Clase**: Piensa en una clase como un plano o un molde para galletas. Define la estructura y el comportamiento de los objetos. Cuando creas una clase, estás especificando cómo deben verse y comportarse los objetos de esa clase.
- **Objeto** o **Instancia**: Un objeto es una realización concreta e individual de una clase, muy parecido a una galleta recién horneada de un molde. Cada objeto creado a partir de una clase comparte la misma estructura definida por la clase pero puede tener diferentes datos y atributos.
- **Atributo**: Los atributos son como los ingredientes que le dan a cada objeto sus características únicas. En Python, los atributos se definen como argumentos en la función `__init__` (el constructor) y se almacenan como datos dentro de cada objeto. Piensa en ellos como variables que contienen información específica para cada objeto.
  - **Atributo de Objeto/Instancia**: Estos atributos son específicos para cada objeto individual. Por ejemplo, en una clase `Estudiante`, el nombre del estudiante y el ID son atributos de instancia porque varían de un objeto estudiante a otro.
  - **Atributo de Clase**: Los atributos de clase se comparten entre todos los objetos creados a partir de la misma clase. Son constantes o valores que permanecen consistentes para todas las instancias. Por ejemplo, si tienes una clase `Escuela`, el nombre de la escuela puede ser un atributo de clase porque es el mismo para todos los estudiantes de esa escuela.

- **Método**: Los métodos son como las acciones o funciones que los objetos pueden realizar. Definen el comportamiento de los objetos. Los métodos son esencialmente funciones que operan sobre los datos o atributos del objeto. Por ejemplo, una clase `Estudiante` podría tener un método `calcular_gpa` que calcula el GPA del estudiante basado en sus calificaciones.

Entender estos conceptos nos ayuda a estructurar y organizar nuestro código de manera efectiva, facilitando la creación, gestión y manipulación de objetos en un paradigma de programación orientada a objetos.

Herencia

La herencia es un concepto fundamental en la programación orientada a objetos que te permite definir nuevas clases basadas en clases existentes. En esta relación, la clase existente se denomina "clase padre", "superclase" o "padre", mientras que la nueva clase se conoce como "clase hija" o "subclase".

Ventajas de la Herencia

La herencia proporciona varias ventajas importantes:

1. **Reutilización de Código**: Permite a las clases hijas utilizar y extender el código de las clases padres, evitando la necesidad de duplicar lógica común.
2. **Organización y Estructura**: Facilita la organización del código al crear una jerarquía de clases, lo que hace que el diseño sea más intuitivo.
3. **Mantenibilidad**: Cualquier cambio realizado en la clase padre se propaga a todas las clases hijas, lo que simplifica la gestión y actualización del código.
4. **Flexibilidad**: Las clases hijas pueden sobrescribir métodos para personalizar su comportamiento sin alterar la funcionalidad de la clase padre.

Cómo Funciona la Herencia

Cuando una clase hija hereda de una clase padre, hereda automáticamente todos los atributos y métodos públicos y protegidos de la clase padre. Esto significa que puede utilizar estos elementos directamente en su propia definición, sin necesidad de reescribirlos.

Tipos de Métodos en Herencia

- **Método Definido en la Clase Padre**: Cuando un método se define en la clase padre pero no en la clase hija, la clase hija hereda el método sin necesidad de modificación. Esto significa que el método en la clase hija funcionará exactamente de la misma manera que en la clase padre, y no hay requisito de sobrescribirlo.
- **Método Definido Solo en la Clase Hija**: En algunos casos, un método puede definirse exclusivamente en la clase hija y no en la clase padre. En este escenario, el

método pertenece únicamente a la clase hija, y no hay herencia en la dirección opuesta. Este enfoque permite que la clase hija tenga su comportamiento y funcionalidad únicos.

- **Método Definido en Ambas Clases, Padre e Hija**: Cuando un método se define en ambas clases, la implementación de la clase hija tiene prioridad. Esto significa que el método definido en la clase hija sobrescribirá el método de la clase padre. Sin embargo, hay una manera de utilizar el método original definido en la clase padre mientras se extiende en la clase hija.

  Para lograr esto, Python proporciona la función `super()`, que permite llamar a cualquier método de la clase padre desde la clase hija. Al invocar `super()`, puedes acceder al método de la clase padre y agregar funcionalidad o comportamiento extra en la clase hija. Es esencial asegurarse de proporcionar todos los atributos y parámetros necesarios al llamar al método de la clase padre a través de `super()`.

En resumen, la herencia es un concepto poderoso que fomenta la reutilización de código, la mantenibilidad y la flexibilidad en la programación orientada a objetos, permitiendo a los desarrolladores construir sobre clases existentes para crear nuevas con funcionalidades compartidas. A través de la herencia, se pueden modelar relaciones entre diferentes clases, promoviendo un enfoque más natural y lógico en la organización del código.

Ejemplo 1

```
# Clase padre
class Shape:
    def __init__(self, name):
        # Inicializar el atributo nombre
        self.name = name

    def get_name(self):
        # Devolver el nombre de la forma
        return f"Estamos trabajando con un {self.name}."

# Clase hija
class Circle(Shape):
    def __init__(self, name, radius):
        # Llamar al constructor de la clase padre
        super().__init__(name)
        # Inicializar el atributo radio
        self.radius = radius

    def area(self):
        # Calcular y devolver el área del círculo
        return 3.14 * self.radius**2

# Clase hija
class Rectangle(Shape):
    def __init__(self, name, length, width):
        # Llamar al constructor de la clase padre
```

```
        super().__init__(name)
        # Inicializar los atributos de largo y ancho
        self.length = length
        self.width = width

    def area(self):
        # Calcular y devolver el área del rectángulo
        return self.length * self.width

# Creando instancias
circle = Circle("Círculo", 5)
rectangle = Rectangle("Rectángulo", 4, 6)

# Llamando a los métodos
print(circle.area())        # Usa el método area de la clase hija (Círculo)
print(circle.get_name())    # Usa el método get_name de la clase padre (Forma)
print(rectangle.area())     # Usa el método area de la clase hija (Rectángulo)
print(rectangle.get_name()) # Usa el método get_name de la clase padre (Forma)
```

```
78.5
We are working with a Circle.
24
We are working with a Rectangle.
```

Ejemplo 2:

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

    def speak(self):
        return f"{self.name} hace un sonido"

# Clase Hija
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # tomando atributos y métodos de la clase PADRE
        self.breed = breed

    def speak(self):
        return f"{self.name} el {self.breed} ladra fuerte"

    def fetch(self):
        return f"{self.name} trae la pelota"
```

```python
# Clase Hija
class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name)  # tomando atributos y métodos de la clase PADRE
        self.color = color

    def speak(self):
        return f"{self.name} el gato {self.color} maúlla suavemente"

    def chase_mice(self):
        return f"{self.name} el gato {self.color} está persiguiendo un ratón"

# Creando instancias
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "Gray")

# Llamando a los métodos
print(dog.speak())         # Sobrescribe el método speak del padre
print(dog.fetch())         # Método adicional en la clase hija
print(cat.speak())         # Sobrescribe el método speak del padre
print(cat.chase_mice())    # Método adicional en la clase hija
```

```
Buddy the Golden Retriever barks loudly
Buddy fetches the ball
Whiskers the Gray cat meows softly
Whiskers the Gray cat is chasing a mouse
```

En este ejemplo:

- Tenemos una clase padre `Animal` con un método `__init__` para inicializar el atributo `name` y un método `speak` que devuelve un sonido genérico.
- Creamos dos clases hijas, `Dog` y `Cat`, que heredan de la clase padre `Animal`.
- Ambas clases hijas tienen sus métodos `__init__` que llaman al método `__init__` de la clase padre usando `super()`. También tienen sus métodos `speak`, que sobrescriben el método `speak` de la clase padre para proporcionar sonidos específicos de la especie. Además, cada clase hija tiene su propio método único (`fetch` para `Dog` y `chase_mice` para `Cat`).
- Creamos instancias de las clases `Dog` y `Cat` y llamamos a sus métodos. Cuando llamamos al método `speak` en cada instancia, utiliza el método sobrescrito específico de esa clase. Los métodos adicionales (`fetch` y `chase_mice`) también están disponibles para las respectivas instancias.

Clases y Métodos Estáticos en Python

Introducción a las Clases

En programación orientada a objetos (OOP), una **clase** es un modelo o plantilla que se utiliza para crear objetos (instancias). Una clase encapsula datos y comportamientos relacionados, lo que permite organizar y estructurar el código de manera más eficiente y modular. Dentro de una clase, puedes definir **atributos** (características) y **métodos** (comportamientos) que describen cómo los objetos de esa clase deben comportarse.

Las clases facilitan la reutilización de código, la organización de datos y la implementación de conceptos como la herencia y el polimorfismo, lo que mejora la mantenibilidad y escalabilidad de las aplicaciones.

Métodos Estáticos

Un **método estático** es un método que pertenece a la clase en sí, en lugar de a cualquier instancia específica de la clase. Esto significa que un método estático no tiene acceso a los atributos de instancia (`self`) ni a los atributos de clase (`cls`). Los métodos estáticos son útiles para definir funciones que realizan tareas relacionadas con la clase, pero que no dependen de los datos de instancia.

Comportamiento de Clases y Métodos Estáticos en Python

Aunque Python no tiene clases "estáticas" en el sentido estricto, ofrece funcionalidades que permiten simular este comportamiento a través de métodos estáticos y atributos de clase.

1. **Métodos Estáticos en Python**

En Python, puedes definir métodos estáticos utilizando el decorador `@staticmethod`. Estos métodos pueden ser llamados sin crear una instancia de la clase. Son ideales para utilidades que no requieren acceso a la instancia o a la clase.

```
class MathUtils:
   @staticmethod
   def add(x, y):
       return x + y

Usando el método estático
result = MathUtils.add(5, 3)  # result es 8
```

2. **Atributos de Clase**

Los atributos de clase son variables que son compartidas por todas las instancias de la clase. Puedes definir atributos de clase directamente dentro de la clase, y estos pueden ser utilizados como si fueran "variables estáticas". Al igual que los métodos estáticos, puedes acceder a estos atributos sin necesidad de crear una instancia de la clase.

```
class Counter:
   count = 0  # Atributo de clase

   def __init__(self):
       Counter.count += 1  # Incrementa el contador cada vez que se
crea una instancia

Creando instancias
c1 = Counter()
c2 = Counter()

print(Counter.count)  # Imprime 2
```

3. **Clases con Comportamiento Estático**

Puedes diseñar una clase que contenga métodos estáticos y atributos de clase para lograr un comportamiento similar al de las clases estáticas. Esto permite encapsular funciones relacionadas y datos que no requieren instancias.

```
class Config:
   version = "1.0"

   @staticmethod
   def get_version():
       return Config.version

Usando el método estático
print(Config.get_version())  # Imprime "1.0"
```

4. **Ventajas de Usar Métodos Estáticos**

- **Claridad del Código:** Los métodos estáticos pueden hacer que el código sea más legible, indicando que no dependen del estado de la instancia.
- **Organización:** Permiten organizar funciones relacionadas dentro de una clase, mejorando la estructura del código.
- **Reutilización:** Facilitan la reutilización de código sin necesidad de crear instancias de una clase.

5. **Limitaciones de los Métodos Estáticos**

- **Sin Acceso a Instancia:** No pueden acceder ni modificar atributos de instancia, lo que puede limitar su uso en algunos contextos.
- **No pueden ser Polimórficos:** A diferencia de los métodos de instancia, no se pueden sobrescribir en clases derivadas, lo que limita el polimorfismo.

Ejemplos Prácticos

Ejemplo 1: Método Estático para Validar Datos

Imagina que necesitas una función que verifique si un número es positivo. Usar un método estático es una buena práctica aquí:

```
class Validator:
   @staticmethod
   def is_positive(number):
       return number > 0

# Validación de números
print(Validator.is_positive(10))  # Imprime True
print(Validator.is_positive(-5))  # Imprime False
```

Ejemplo 2: Uso de Atributos de Clase para Contar Instancias

Podemos utilizar un atributo de clase para contar cuántas instancias de una clase se han creado:

```
class InstanceCounter:
   instance_count = 0  # Atributo de clase

   def __init__(self):
       InstanceCounter.instance_count += 1

c1 = InstanceCounter()
c2 = InstanceCounter()

print(InstanceCounter.instance_count)  # Imprime 2
```

Conclusión

En resumen, aunque Python no cuenta con clases estáticas en el sentido convencional, proporciona métodos estáticos y atributos de clase que permiten agrupar funcionalidades relacionadas con una clase sin necesidad de instanciarla. Estos conceptos son útiles para mejorar la organización y la eficiencia en el código, manteniendo al mismo tiempo la flexibilidad característica de la programación orientada a objetos.