1. OBJETOS

En la programación orientada a objetos (POO), un objeto es una entidad que combina datos (llamados atributos) y comportamiento (llamado métodos) relacionados en una sola unidad. Los objetos son instancias de una clase, que es una plantilla o molde que define la estructura y el comportamiento de los objetos de ese tipo en particular.

Las ventajas de la programación orientada a objetos son las siguientes:

- Reutilización de código: La POO fomenta la reutilización de código a través de la creación de clases y la creación de múltiples objetos a partir de esas clases. Esto permite aprovechar y mantener el código existente en lugar de tener que volver a escribirlo desde cero.
- Modularidad: La POO facilita la creación de programas modulares. Los objetos encapsulan datos y comportamiento relacionados, lo que permite dividir un programa en componentes más pequeños y manejables. Cada objeto puede ser desarrollado y probado de manera independiente, lo que mejora la legibilidad y mantenibilidad del código.
- Abstracción: La POO permite la creación de abstracciones que representan conceptos del mundo real. Puedes modelar objetos del mundo real como objetos en tu programa, lo que facilita el diseño y la comprensión del software. Por ejemplo, puedes crear una clase "Coche" que tenga atributos como "color" y "velocidad" y métodos como "acelerar" y "frenar", lo que refleja la realidad de un coche.
- Encapsulación: La encapsulación es un principio importante en la POO. Permite ocultar los detalles internos de un objeto y exponer solo una interfaz bien definida para interactuar con él. Esto proporciona una mayor seguridad y evita que se acceda o se modifique directamente los datos internos del objeto.
- Herencia: La herencia es un concepto clave en la POO que permite crear nuevas clases basadas en clases existentes. Una clase derivada (o subclase) hereda los atributos y métodos de la clase base (o superclase). Esto permite extender y especializar el comportamiento de una clase sin tener que volver a escribir código, lo que facilita la organización jerárquica y la reutilización.

En contraste, la programación estructurada se basa en el concepto de procedimientos o funciones, donde se divide el programa en funciones independientes que manipulan datos estructurados. La programación estructurada se enfoca en la lógica y los algoritmos, mientras que la POO se enfoca en el modelado de entidades y su interacción.

2. TYPE

Si recordamos de las unidades anteriores, Python nos proveía de la la función type() que se puede utilizar para verificar los tipos de entidades, como números, cadenas de texto, listas, diccionarios, entre otros.

```
# Verificar el tipo de un número
numero = 10
print(type(numero))  # Imprimirá "<class 'int'>"

# Verificar el tipo de una cadena de texto
texto = "Hola, mundo"
print(type(texto))  # Imprimirá "<class 'str'>"

# Verificar el tipo de una lista
lista = [1, 2, 3, 4, 5]
print(type(lista))  # Imprimirá "<class 'list'>"

# Verificar el tipo de un diccionario
diccionario = {"clave": "valor"}
print(type(diccionario))  # Imprimirá "<class 'dict'>"
```

¡Cómo se puede ver, type nos está devolviendo como información class, lo que indica que la entidad que estamos consultando es un objeto!

3. OBJETOS EN PYTHON

En Python, un objeto es una entidad que combina datos y funciones relacionadas en una sola unidad. Los objetos son instancias de una clase, que es una plantilla que define la estructura y el comportamiento de los objetos de ese tipo en particular.

En este ejemplo, se define una clase llamada Persona. La clase tiene un constructor __init__() que se ejecuta cuando se crea una nueva instancia de la clase. El constructor toma dos parámetros: nombre y edad. Estos

parámetros se asignan a los atributos nombre y edad del objeto utilizando la sintaxis self.nombre y self.edad.

La clase Persona también tiene un método llamado saludar(), que imprime un mensaje de saludo con el nombre y la edad de la persona.

Luego, se crea una instancia de la clase Persona llamada persona1, pasando los valores "Juan" y 30 como argumentos al constructor. Para acceder a los atributos del objeto persona1, se utiliza la sintaxis objeto.atributo (por ejemplo, persona1.nombre).

Finalmente, se llama al método saludar() del objeto persona1, utilizando la sintaxis objeto.metodo() (por ejemplo, persona1.saludar()). Esto imprimirá el mensaje "Hola, mi nombre es Juan y tengo 30 años."

```
# Definir una clase llamada Persona
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

# Crear una instancia de la clase Persona
persona1 = Persona("Juan", 30)

# Acceder a los atributos del objeto persona1
print(persona1.nombre)  # Imprimirá "Juan"
print(persona1.edad)  # Imprimirá 30

# Llamar al método saludar() del objeto persona1
```

```
persona1.saludar()  # Imprimirá "Hola, mi nombre es Juan y tengo 30 años."
```

3.1. CONSTRUCTOR

Un constructor en programación orientada a objetos es un método especial que se utiliza para inicializar un objeto cuando se crea una instancia de una clase. El constructor se llama automáticamente al crear un nuevo objeto y se encarga de establecer los valores iniciales de los atributos del objeto.

En Python, el constructor se define mediante un método llamado __init__(). El nombre __init__ es reservado y no debe ser cambiado. Este método puede tener parámetros para recibir valores que se utilizarán para inicializar los atributos del objeto.

Ejemplo 1: Constructor sin parámetros:

```
class Persona:
    def __init__(self):
        self.nombre = "Juan"
        self.edad = 30

# Crear una instancia de la clase Persona
persona1 = Persona()

# Acceder a los atributos del objeto persona1
print(persona1.nombre)  # Imprimirá "Juan"
print(persona1.edad)  # Imprimirá 30
```

En este ejemplo, se define una clase Persona con un constructor __init__() que no recibe ningún parámetro. Dentro del constructor, se inicializan los atributos nombre y edad con valores predeterminados. Al crear una instancia de la clase Persona utilizando persona1 = Persona(), se llama automáticamente al constructor y se crean los atributos nombre y edad en el objeto persona1.

Ejemplo 2: Constructor con parámetros:

```
class Persona:
    def __init__(self, nombre, edad):
```

```
        self.nombre = nombre
        self.edad = edad

# Crear una instancia de la clase Persona con valores personalizados
persona2 = Persona("María", 25)

# Acceder a los atributos del objeto persona2
print(persona2.nombre)  # Imprimirá "María"
print(persona2.edad)  # Imprimirá 25
```

En este ejemplo, se define una clase Persona con un constructor __init__() que recibe dos parámetros: nombre y edad. Los parámetros se utilizan para inicializar los atributos nombre y edad del objeto. Al crear una instancia de la clase Persona utilizando persona2 = Persona("María", 25), se pasa los valores "María" y 25 al constructor, que se utilizan para inicializar los atributos correspondientes en el objeto persona2.

El constructor es opcional en una clase. Si no se define un constructor, Python proporcionará uno por defecto que no realiza ninguna acción específica.

3.2. ATRIBUTOS

En Python, un atributo de clase es una variable que pertenece a la clase en sí, en lugar de pertenecer a una instancia particular de la clase. Esto significa que todos los objetos creados a partir de la clase comparten el mismo valor para el atributo de clase.

Para definir un atributo de clase en Python, se coloca la variable dentro de la definición de la clase, pero fuera de cualquier método. Se accede a los atributos de clase utilizando el nombre de la clase o cualquier objeto creado a partir de esa clase.

Ejemplo 1: Atributo de clase básico:

```
class Circulo:
    # Atributo de clase
    pi = 3.14159
```

```
# Acceder al atributo de clase
print(Circulo.pi)  # Imprimirá 3.14159

# Crear objetos de la clase Circulo
circulo1 = Circulo()
circulo2 = Circulo()

# Acceder al atributo de clase a través de los objetos
print(circulo1.pi)  # Imprimirá 3.14159
print(circulo2.pi)  # Imprimirá 3.14159
```

En este ejemplo, se define la clase Circulo con un atributo de clase llamado pi. El atributo pi se define fuera de cualquier método y se accede utilizando el nombre de la clase Circulo o cualquier objeto creado a partir de ella. Todos los objetos de la clase Circulo comparten el mismo valor para el atributo pi, que es 3.14159.

Ejemplo 2: Atributo de clase modificado por una instancia:

```
class CuentaBancaria:
    tasa_interes = 0.05

# Acceder al atributo de clase
print(CuentaBancaria.tasa_interes)  # Imprimirá 0.05

# Modificar el atributo de clase a través de una instancia
cuenta1 = CuentaBancaria()
cuenta1.tasa_interes = 0.07
```

```
# Acceder al atributo de clase a través de la instancia y la clase
print(cuenta1.tasa_interes)  # Imprimirá 0.07
print(CuentaBancaria.tasa_interes)  # Imprimirá 0.05
```

En este ejemplo, se define la clase CuentaBancaria con un atributo de clase llamado tasa_interes. El atributo tasa_interes tiene un valor predeterminado de 0.05. Sin embargo, también se puede acceder y modificar a través de una instancia de la clase. Cuando se modifica el atributo de clase a través de la instancia (cuenta1.tasa_interes = 0.07), se crea un nuevo atributo específico para esa instancia, que tiene prioridad sobre el atributo de clase. Los demás objetos y la clase en sí mantendrán el valor original del atributo de clase.

3.3. MÉTODOS

En Python, un método es una función que está asociada a una clase y se utiliza para definir el comportamiento de los objetos de esa clase. Los métodos son similares a las funciones, pero se definen dentro de la definición de la clase y pueden acceder y manipular los atributos del objeto.

Existen también los métodos especiales, también conocidos como métodos mágicos o dunder methods (por su nombre en inglés, "double underscore"). Estos métodos tienen nombres especiales que comienzan y terminan con doble guion bajo (__). Proporcionan una forma de implementar comportamientos específicos en una clase y son llamados automáticamente en ciertas situaciones.

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

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

    def cumpleanios(self):
        self.edad += 1
```

```
        print(f"Feliz cumpleaños, ahora tengo {self.edad} años.")


# Crear una instancia de la clase Persona
persona1 = Persona("Juan", 30)


# Llamar al método saludar()
persona1.saludar()  # Imprimirá "Hola, mi nombre es Juan y tengo 30 años."


# Llamar al método cumpleanios()
persona1.cumpleanios()  # Imprimirá "Feliz cumpleaños, ahora tengo 31 años."
```

En este ejemplo, se define la clase Persona con tres métodos: __init__(), saludar(), y cumpleanios(). El método __init__() es el constructor y se llama automáticamente al crear una instancia de la clase. Los métodos saludar() y cumpleanios() son definidos por el usuario.

El método saludar() imprime un mensaje de saludo utilizando los atributos nombre y edad del objeto. El método cumpleanios() incrementa la edad en 1 y luego imprime un mensaje de felicitación.

Para llamar a un método, se utiliza la sintaxis objeto.metodo(). En el ejemplo, se crea una instancia de la clase Persona llamada persona1. Luego, se llama a los métodos saludar() y cumpleanios() utilizando persona1.saludar() y persona1.cumpleanios() respectivamente.

Los métodos especiales se utilizan para proporcionar comportamientos específicos, como la representación de un objeto como una cadena de texto, la sobrecarga de operadores, el manejo de iteraciones, entre otros. Algunos ejemplos de métodos especiales en Python son __str__(), __len__(), __add__(), __iter__(), entre otros.

Por ejemplo, aquí tienes un método especial __str__() que permite representar un objeto Persona como una cadena de texto:

```
class Persona:

    def __init__(self, nombre, edad):

        self.nombre = nombre
```

```
        self.edad = edad

    def __str__(self):
        return f"Persona(nombre={self.nombre}, edad={self.edad})"

# Crear una instancia de la clase Persona
persona1 = Persona("Juan", 30)

# Utilizar el método especial __str__()
print(persona1)  # Imprimirá "Persona(nombre=Juan, edad=30)"
```

En este caso, el método __str__() se define para devolver una representación en forma de cadena de texto del objeto Persona. Al utilizar print(persona1), se llama automáticamente al método __str__() y se imprime la representación personalizada del objeto.

3.4. COMPARACIÓN DE OBJETOS

En Python, la comparación de objetos se realiza utilizando los operadores de comparación, como el operador de igualdad (==), el operador de desigualdad (!=), los operadores de comparación mayor que (>), menor que (<), mayor o igual que (>=) y menor o igual que (<=).

Cuando se comparan objetos utilizando los operadores de comparación, se compara la identidad de los objetos por defecto. Es decir, se verifica si los objetos son el mismo objeto en memoria, no si tienen los mismos valores o atributos.

Sin embargo, es posible personalizar la comparación de objetos definiendo los métodos especiales __eq__(), __ne__(), __gt__(), __lt__(), __ge__(), y __le__(). Estos métodos permiten especificar cómo se debe realizar la comparación entre objetos de una clase.

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

```
    def __eq__(self, otro):
        if isinstance(otro, Persona):
            return self.nombre == otro.nombre and self.edad
== otro.edad
        return False


# Crear instancias de la clase Persona
persona1 = Persona("Juan", 30)
persona2 = Persona("Juan", 30)
persona3 = Persona("María", 25)


# Comparar objetos utilizando el operador de igualdad (==)
print(persona1 == persona2)  # Imprimirá True
print(persona1 == persona3)  # Imprimirá False
```

En este ejemplo, se define la clase Persona con un método especial __eq__() para personalizar la comparación de igualdad entre objetos. El método __eq__() verifica si el objeto pasado como argumento (otro) es una instancia de la clase Persona y luego compara los atributos nombre y edad de los objetos para determinar si son iguales.

Al utilizar el operador de igualdad (==) para comparar persona1 y persona2, se llamará al método __eq__() definido en la clase Persona. Como los objetos tienen el mismo nombre y edad, la comparación devolverá True. Sin embargo, al comparar persona1 y persona3, que tienen diferentes nombres, la comparación devolverá False.

Es importante tener en cuenta que la implementación de la comparación de objetos depende de las necesidades específicas de tu programa y de la lógica que deseas aplicar en la comparación. Los métodos especiales __eq__(), __ne__(), __gt__(), __lt__(), __ge__(), y __le__() te permiten personalizar estas comparaciones según tus requerimientos.

4. PENSAR EN OBJETOS

Cuando se aborda un problema utilizando programación orientada a objetos, se enfatiza en identificar los objetos relevantes en el dominio del problema y cómo interactúan entre sí.

- Identificar los objetos: Analiza el problema y determina qué elementos o entidades están involucrados. Los objetos son representaciones de estas entidades en el mundo real o abstracto. Por ejemplo, si estamos resolviendo un problema de una tienda en línea, algunos objetos potenciales podrían ser "cliente", "producto", "carrito de compras", etc.
- Definir las propiedades y comportamientos de los objetos: Para cada objeto identificado, piensa en las características que lo describen (propiedades) y las acciones que puede realizar (comportamientos o métodos). Por ejemplo, un objeto "producto" podría tener propiedades como "nombre", "precio" y "cantidad en stock", y comportamientos como "calcular_descuento" o "actualizar_stock".
- Identificar las interacciones entre objetos: Determina cómo los objetos interactúan entre sí para resolver el problema. ¿Cómo se comunican los objetos y cómo se influyen mutuamente? Estas interacciones pueden ser en forma de mensajes enviados entre objetos para solicitar información o realizar acciones.
- Modelar las relaciones entre objetos: Define las relaciones entre los objetos, como la asociación, la composición o la herencia. Estas relaciones ayudan a estructurar y organizar el código de manera más coherente. Por ejemplo, en una tienda en línea, puede haber una relación de asociación entre los objetos "cliente" y "carrito de compras", ya que un cliente puede tener un carrito de compras asociado.
- Implementar las clases y objetos en el código: Con base en la estructura conceptual desarrollada, implementa las clases y crea instancias de los objetos en el código. Cada clase representa un tipo de objeto y define sus propiedades y comportamientos. Las instancias de objetos son los datos reales que se crean a partir de esas clases.
- Utilizar los objetos para resolver el problema: Utiliza los objetos creados para realizar las operaciones necesarias y resolver el problema. Llama a los métodos de los objetos, pasa mensajes entre ellos y utiliza sus propiedades para llevar a cabo las acciones requeridas.

Al pensar en la solución como objetos, se fomenta el enfoque en la estructura y el diseño del código, lo que permite una mayor modularidad, reutilización y mantenibilidad. Además, la programación orientada a objetos proporciona abstracciones más cercanas a la forma en que pensamos y nos

relacionamos con los conceptos en el mundo real, lo que puede hacer que el código sea más comprensible y escalable.

Es importante tener en cuenta que el proceso de pensar en la solución como objetos puede variar según el problema y la complejidad del mismo. La práctica y la experiencia en el desarrollo orientado a objetos ayudarán a perfeccionar esta habilidad y a tomar decisiones más efectivas en el diseño de las clases y objetos.

Ejemplo:

Problema: Crear un sistema de gestión de una biblioteca.

Pasos del proceso:

1. Identificar los objetos: En este caso, algunos objetos relevantes podrían ser "Biblioteca", "Libro" y "Usuario".

2. Definir las propiedades y comportamientos de los objetos:

El objeto "Biblioteca" podría tener propiedades como "nombre" y "ubicación", y comportamientos como "agregar_libro", "prestar_libro" y "buscar_libro".

El objeto "Libro" podría tener propiedades como "título", "autor" y "disponible", y comportamientos como "obtener_info" y "cambiar_estado_disponibilidad".

El objeto "Usuario" podría tener propiedades como "nombre", "ID" y "libros_prestados", y comportamientos como "registrar_prestamo" y "devolver_libro".

3. Identificar las interacciones entre objetos:

Un usuario puede solicitar un préstamo de un libro a la biblioteca.

La biblioteca verifica la disponibilidad del libro y registra el préstamo.

El libro actualiza su estado de disponibilidad y se agrega a la lista de libros prestados del usuario.

4. Modelar las relaciones entre objetos:

La biblioteca puede tener una asociación con los objetos "Libro" y "Usuario".

El objeto "Usuario" podría tener una composición con el objeto "Libro", ya que puede poseer varios libros.

5. Implementar las clases y objetos en el código:

```
class Biblioteca:
```

```
    def __init__(self, nombre, ubicacion):
        self.nombre = nombre
        self.ubicacion = ubicacion
        self.libros = []

    def agregar_libro(self, libro):
        self.libros.append(libro)

    def prestar_libro(self, libro, usuario):
        if libro.disponible:
            libro.cambiar_estado_disponibilidad(False)
            usuario.registrar_prestamo(libro)
            print(f"El libro '{libro.titulo}' ha sido
prestado a {usuario.nombre}.")
        else:
            print(f"El libro '{libro.titulo}' no está
disponible en este momento.")

    def buscar_libro(self, titulo):
        for libro in self.libros:
            if libro.titulo == titulo:
                return libro
        return None


class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor
        self.disponible = True
```

```
    def obtener_info(self):
        print(f"Título: {self.titulo}")
        print(f"Autor: {self.autor}")
        print(f"Disponible: {'Sí' if self.disponible else 'No'}")

    def cambiar_estado_disponibilidad(self, disponible):
        self.disponible = disponible


class Usuario:
    def __init__(self, nombre, ID):
        self.nombre = nombre
        self.ID = ID
        self.libros_prestados = []

    def registrar_prestamo(self, libro):
        self.libros_prestados.append(libro)

    def devolver_libro(self, libro):
        if libro in self.libros_prestados:
            libro.cambiar_estado_disponibilidad(True)
            self.libros_prestados.remove(libro)
            print(f"El libro '{libro.titulo}' ha sido devuelto.")
        else:
            print(f"No tienes el libro '{libro.titulo}' prestado.")
```

```
# Crear instancias de objetos
biblioteca = Biblioteca("Biblioteca Central", "Ciudad A")
libro1 = Libro("El Gran Gatsby", "F. Scott Fitzgerald")
libro2 = Libro("1984", "George Orwell")
usuario1 = Usuario("Juan", 123456)

# Agregar libros a la biblioteca
biblioteca.agregar_libro(libro1)
biblioteca.agregar_libro(libro2)

# Prestar un libro a un usuario
biblioteca.prestar_libro(libro1, usuario1)

# Mostrar información del libro
libro1.obtener_info()

# Devolver un libro
usuario1.devolver_libro(libro1)
```

En este ejemplo, se definen las clases Biblioteca, Libro y Usuario con sus respectivas propiedades y comportamientos. Luego, se crean instancias de los objetos y se utilizan los métodos para agregar libros a la biblioteca, prestar un libro a un usuario, obtener información del libro y devolver un libro prestado.

Este enfoque de programación orientada a objetos ayuda a organizar la lógica del programa de manera más modular y facilita la representación de las entidades del mundo real (en este caso, una biblioteca, libros y usuarios).