Programación Orientada a Objetos (P.O.O.) y Estructurada

Atributos o Propiedades ⇒ que son datos:

Es un paradigma de programación que organiza el software entorno a **objetos**, los cuales combinan **datos (atributos)** y **comportamientos (métodos)** en una misma entidad. Estos objetos se basan en **clases**, que funcionan, como **moldes o plantillas** que definen sus características y acciones.

Diferencia entre Programación Estructurada y P.O.O.

Características	**Programación Estructurada**	**Programación Orientada a Objetos**
Paradigmas	**Se basa en** procedimiento y funciones **que operan sobre** datos**.**	**Se basa en** objetos **que combinan** datos y comportamientos**.**
Organización del Código	**El programa se divide en** módulos o funciones**.**	**El programa se divide en** clases y objetos**.**
Enfoque	**Orientado ha** procesos**: las** secuencias de instrucciones **es lo principal.**	**Orientado ha** identidades**: los** objetos **del mundo real son los principales.**
Datos	**Los datos son** globales o compartidos **y pueden ser** modificados por cualquier función**.**	**Los datos (**atributos**) están** encapsulados **dentro de los objetos y solo son** accesibles mediante sus métodos**.**
Reutilización	**Baja: requiere** repetir el código o copiar funciones**.**	**Alta: gracias a la** herencia **y el** polimorfismo**.**
Mantenimiento	**Difícil por que un cambio en los datos puede afectar muchas funciones.**	**Más sencillo, porque los cambios se localizan en** clases u objetos específicos**.**

Resumen:

La **programación estructurada** organiza el software en funciones y pasos secuenciales siendo más adecuada para programas pequeños y sencillos. La **P.O.O.**, organiza el

software en **clases y objetos**, lo cual facilita la **modularidad, reutilización y escalabilidad**, siendo la más usada en sistemas modernos.

Ventajas de la P.O.O.

1. **Modularidad:** El código se organiza en **clases y objetos**, lo que facilita la comprensión y mantenimiento.
2. **Reutilización del código:** Gracias a la **herencia** y al **polimorfismo**, se pueden crear nuevas clases a partir de otras sin repetir código.
3. **Encapsulamiento:** Protege los datos internos de los objetos, permitiendo acceso solo a través de **métodos definidos**.
4. **Abstracción:** Permite representar entidades del mundo real en **modelos computacionales**, simplificando la complejidad.
5. **Escalabilidad:** Es adecuada para el desarrollo de sistemas grandes y complejos, porque facilita dividir el trabajo en módulos independientes.
6. **Mantenimiento más sencillo:** Los cambios se realizan en **clases específicas** sin afectar todo el sistema.
7. **Flexibilidad y extensibilidad:** Permite **adaptar y ampliar sistemas** de manera más rápida y ordenada.
8. **Productividad:** Facilita el **trabajo en equipo** y a que varios programadores pueden desarrollar distintas clases u objetos de forma más rápida y ordenada.

Aplicaciones de la P.O.O.

La P.O.O. se aplica en una amplia variedad de áreas de desarrollo de software, ya que facilita la construcción de sistemas **modulares, reutilizables y escalables**. Entre sus principales aplicaciones destacan:

1. **Desarrollo de aplicaciones de escritorio:** Programas como editores de texto, hojas de cálculo o software de gestión utilizan programación orientada a objetos para organizar **ventanas, menú, botones y acciones** como objetos.
2. **Aplicaciones web y móviles:** Muchos **frameworks** aplican principios de P.O.O. para manejar usuarios, productos, pedidos, mensajes, etc.
3. **Sistemas de bases de datos:** La P.O.O. se usa en el diseño de base de datos orientados a objetos (B.D.O.O.) o en el **Relacional Mapping**, que permiten mapear tablas a objetos de manera directa.
4. **Videojuegos:** Cada **personaje, enemigo, escenario o arma** puede representarse como un objeto con atributos (vida, posición, color) y métodos (moverse, atacar, desaparecer).
5. **Inteligencia artificial y simulación:** Los **agentes inteligentes, redes neuronales o simulaciones físicas** se modelan como objetos que interactúan entre sí.
6. **Sistemas distribuidos y en la nube:** Los **servicios** se modelan como objetos que se comunican en diferentes nodos, manteniendo **modularidad y escalabilidad**.
7. **Interfaces gráficas de usuario: Librerías** como Tkinter (Python), Java FX/Java y otros organizan cada elemento gráfico (**botones, cuadros de texto, menús**) como objetos reutilizables.

En conclusión: P.O.O. se aplica en cualquier área donde se requiera **organizar y modelar entidades complejas del mundo real**, siendo el paradigma más utilizado en el desarrollo de software moderno.

Clases y Objetos

Clases

En P.O.O. una **clase es una plantilla o molde** que permiten crear objetos.

- Define **atributos** (propiedades o características).
- Define **métodos** (acciones o comportamientos).

Objetos

Los **objetos** son **instancias de una clase**, es decir, **representaciones concretas** a partir de esa plantilla.

En otras palabras:

- La clase es el **molde** o plantillas
- El objeto es el **producto** creado a partir de ese molde

EJEMPLOS:

Código 1:

```
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def mostrar_info(self):
        print(f"Coche: {self.marca} {self.modelo} {self.color}")

    def arrancar(self):
        print(f"El {self.marca} {self.modelo} ha arrancado")

marca = input("ingrese la marca del coche")
modelo = input("ingrese el modelo del coche")
color = input(" ingrese el color del coche")

mi_coche = Coche(marca, modelo, color)

mi_coche.mostrar_info()
mi_coche.arrancar()
```

código 2:

```
class rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    # calcular area
    def calcular_area(self):
        return (self.base * self.altura)

    # calcular perimetro
    def calcular_perimetro(self):
        return (self.base * 2 + self.altura * 2)

base = float(input("ingrese la base del rectangulo"))
altura = float(input("ingrese la altura del triangulo"))

# crear objeto
rec = rectangulo(base, altura)

print("el area del rectangulo es ", rec.calcular_area())
print("el perimetro es ", rec.calcular_perimetro())
```

código 3:

```
class Pared:
    def __init__(self, largo_ladrillo, alto_ladrillo, junta, area_pared):
        self.largo_ladrillo = largo_ladrillo
        self.alto_ladrillo = alto_ladrillo
        self.junta = junta
        self.area_pared = area_pared
```

```
    def ladrillos_por_metro(self):
        largo_total = self.largo_ladrillo + self.junta
        alto_total = self.alto_ladrillo + self.junta
        cantidad = 1 / (largo_total * alto_total)
        return cantidad

    def ladrillos_totales(self):
        # largo (Esta línea parece ser un residuo o incompleta en el código original)
        # alto (Esta línea parece ser un residuo o incompleta en el código original)
        return self.ladrillos_por_metro() * self.area_pared

largo = float(input("Ingresa el largo del ladrillo (en metros): "))
alto = float(input("Ingresa el alto del ladrillo (en metros): "))
junta = float(input("Ingresa el grosor de la junta (en metros): "))
area = float(input("Ingresa el área total de la pared (en m²): "))

pared = Pared(largo, alto, junta, area)

print("\n--- RESULTADOS ---")
print(f"Ladrillos por metro cuadrado: {pared.ladrillos_por_metro():.2f}")
print(f"Total de ladrillos necesarios: {pared.ladrillos_totales():.0f}")
```

Estructuras Repetitivas (Python)

En Python, las **estructuras repetitivas** (también llamadas **bucles o loops**) son **instrucciones** que permiten **ejecutar un bloque de código varias veces** dependiendo de una **condición** o una **secuencia de elementos**. Los tipos de estructura repetitivo en Python son:

1. **While (mientras):** Ejecuta un bloque de código **mientras una condición sea verdadera**.
    - *Ejemplo:*

      Ejemplo: contador = 0

```
while contador<=5:
    Print(contador)
    contador += 1
```

2. Bucle `for` (para)

El bucle `for` **recorre una secuencia** (como una lista, tupla, cadena o rango de números) **y ejecuta un bloque por cada elemento**.

- *Ejemplo:*

```
for i in range(1, 6):
    print(i)
```

CODIGO 1:

```
class Factorial:
    def __init__(self, numero):
        # atributo
        self.numero = numero
        self.resultado = 1

    def calcular(self):
        if self.numero < 0:
            print("Operación inválida")
            return None
        for i in range(1, self.numero + 1):
            self.resultado *= i
        return self.resultado

# función principal
def main():
```

```
    mi_factorial = Factorial(5)
    resultado = mi_factorial.calcular()

    # mostrar el resultado
    if resultado is not None:
        print(f"El factorial de {mi_factorial.numero} es {resultado}")

if __name__ == "__main__":
    main()
```

CODIGO 2:

```
class Fibonacci:
    def __init__(self, cantidad):
        self.cantidad = cantidad
        self.serie = []  # Corregido: Inicialización de lista vacía para la serie

    def generar_serie(self):
        a, b = 0, 1
        for _ in range(self.cantidad):
            self.serie.append(a)
            a, b = b, a + b
        return self.serie

def main():
    cantidad = int(input("introduzca la cantidad de la serie: "))
    mi_fibonacci = Fibonacci(cantidad)
```

```
    resultado = mi_fibonacci.generar_serie()

    print(resultado)


if __name__ == "__main__":

    main()
```

Encapsulamiento

Definición de Encapsulamiento

En P.O.O. el elemento **encapsulamiento** es un tema en **prot** que consiste **ocultar los detalles internos** del funcionamiento de un objeto y **restringir el acceso directo** a algunos de sus **componente**, exponiendo solo lo necesario **a través de interfaces públicas (métodos)**.

Definición: encapsulamiento es el **mecanismo** que permite **agrupar los datos (atributos) y los métodos** que operan sobre esos datos en una sola unidad (**clase**), y **controlar el acceso** a ellos para proteger el estado interno del objeto.

Características:

- **Ocultamiento de datos:** los atributos de una clase suelen declararse como **privados o protegidos**, lo que impide que se acceda directamente a ellos desde fuera de la clase.
- **Control mediante métodos:** Se utilizan **métodos públicos (getter y setters)** para leer o modificar los atributos, aplicando validaciones si es necesario.
- **Mayor seguridad y mantenimiento:** Al proteger los datos internos, se evita que otras partes del programa los modifique de forma incorrecta o inesperada.

Ventajas:

- Protege la **integridad de los datos**.
- Permite **cambiar la implementación interna** sin afectar el código externo.
- Favorece el **diseño modular** y la **reutilización del código**.

¿Qué es getter?

Un **getter** es un **método** que permite **acceder al valor** de un **atributo privado o protegido** de una clase.

- Su función es **obtener el valor**.
- Se une para leer un atributo **sin modificarlo directamente**.

¿Qué es setter?

Un **setter** es un **método** que permite **modificar el valor** de un **atributo privado o protegido** de una clase.

- Su función es **establecer asignar un nuevo valor**.
- Normalmente incluye **validaciones** para asegurar que el valor sea correcto.

CODIGO 1:

```
class NumeroMultiplo:
    def __init__(self, valor):
        self.valor = valor


    def mostrar_multiplo(self):
        if self.valor == 0:
            print(f"El: {self.valor} es número nulo")
        elif self.valor % 3 == 0 and self.valor % 5 == 0:
            print(f"Número: {self.valor} es múltiplo de 3 y de 5")
        elif self.valor % 3 == 0:
            print(f"Número: {self.valor} es múltiplo de 3")
        elif self.valor % 5 == 0:
            print(f"Número: {self.valor} es múltiplo de 5")
        else:
            print(f"Número: {self.valor} No es múltiplo de 3 ni de 5")
        print("-" * 60)


def main():
    i = 0
    while i <= 10:
        numero = NumeroMultiplo(i)
        numero.mostrar_multiplo()
        i += 1


if __name__ == "__main__":
```

```
    main()
```

CODIGO 2:

```
class Fibonacci:  # [cite: 2]
    def __init__(self, limite):  # [cite: 3, 4]
        self.limite = limite  # [cite: 5, 6]
        self.a = 0  # [cite: 7]
        self.b = 1  # [cite: 8]
        self.contador = 0  # [cite: 9]

    def generar(self):  # [cite: 10]
        while self.contador < self.limite:  # [cite: 11]
            print(self.a, end="-")  # [cite: 12]
            self.a, self.b = self.b, self.a + self.b  # [cite: 13]
            self.contador += 1  # [cite: 13]

n = int(input("Ingrese la cantidad de términos de Fibonacci que desea: "))  # [cite: 14]
fib = Fibonacci(n)  # [cite: 15]

print("Serie de Fibonacci:")  # [cite: 16]
fib.generar()  # [cite: 16]
```

Métodos y Sobrecargas

1. **Métodos:** Un **método** es una **función definida dentro de una clase** que realiza una **acción sobre los datos del objeto**. Los métodos representan el **comportamiento** del objeto.
    - *Ejemplo:*
2. **Parámetros en método:** Los métodos pueden recibir **parámetros** para trabajar con **datos externos**. El primer parámetro de todo método es de instancia es "**self**", que representa al objeto mismo.
    - *Ejemplo:*
3. **Retorno de métodos:** Un método puede **devolver un valor** usando la instrucción (**return**). Esto permite usar el resultado del método en otras partes del programa.
    - *Ejemplo:*
4. **Sobrecarga de métodos:** Consiste en definir **múltiples de acciones de un método** con el mismo nombre, pero **diferentes parámetros**.

Estructuras Selectivas (Python)

En el contexto de P.O.O. con Python, las **estructuras selectivas** son aquellas que permiten **tomar decisiones durante la ejecución del programa**, es decir, **ejecutar diferentes bloques de código** según se cumpla o no una condición.

Aunque las estructuras selectivas no son **exclusivas** de la **programación P.O.O.** (también se usan en programación estructurada), en un programa P.O.O. se usan **dentro de elementos o funciones** que pertenecen a clases para controlar el flujo de ejecución.

Definición: Una **estructura selectiva** en la P.O.O. en Python es una **construcción de control** que permite **ejecutar diferentes bloques de código** dentro de métodos de clases, dependiendo de si se cumple una o varias condiciones.

1. **Estructura selectiva simples:** Se ejecuta un bloque de código solo si se cumple la **condición**, en caso contrario, no hace nada.
    - *Ejemplo:*

      ```
      if (x>0)

          Print("x es positiva")
      ```

2. **Estructura selectiva doble:** Permite dos **ramas**: una si la condición es **verdadera**, otra si es **falsa**.

- *Ejemplo:*

  ```
  if (x % 2 == 0)

      Print("x es un número par")

  else:

      Print("x es un número impar")
  ```

3. **Estructura selectiva múltiple:** Gestiona **más de dos alternativas mutuamente exclusivas**. Solo se ejecuta la primera condición verdadera.

- *Ejemplo:*

  ```
  if (x >= 90)

      Print("x aprobado condistinción")

  elif (x >= 80)

      Print("x aprobado")
  ```

```
else:
    Print("x reprobado")
```