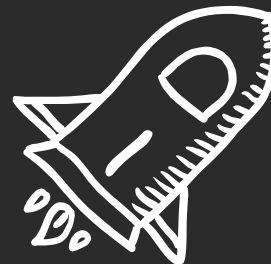




Semana 5 ¡Bienvenidos!



Temas de hoy

1 Conceptos básicos de
POO: clases, objetos,
instancia.

2 Atributos y métodos.

3 Principios SOLID.

1

Conceptos básicos de POO

¿Te imaginás poder crear tus propios objetos como si fueras un director de cine, dándoles roles y responsabilidades? Bueno, de eso se trata la POO: de organizar y estructurar nuestro código para que sea más fácil de entender, mantener y reutilizar. Nos vamos a meter de lleno en conceptos clave como clases, objetos, métodos y atributos.

¿Vamos? 💪🚀

POO: Concepto

Paradigma

En programación, un **paradigma** es un enfoque o estilo de programación que proporciona un marco para pensar y resolver problemas. Define la estructura y el estilo del código, así como la manera en que los programadores deben pensar acerca de las tareas de programación y cómo las soluciones deben ser formuladas.

La **programación orientada a objetos** (POO) es un paradigma de programación que organiza el software en torno a “**objetos**”, que son “**instancias**” de “**clases**”. Este enfoque, refleja una manera de pensar y estructurar el código, basado en conceptos del mundo real y permite crear sistemas modulares, extensibles y mantenibles.

Conceptos Fundamentales de la POO

- - Clases y Objetos:

- **Clase:** Es un **plano**, **plantilla**, o **molde** que define las propiedades (*atributos*) y comportamientos (*métodos*) que los objetos creados a partir de la *clase* tendrán. Por ejemplo, una *clase* **Coche** podría tener atributos como *marca*, *modelo*, y *color*, y métodos como *acelerar()* y *frenar()*.
- **Objeto:** Es una **instancia** de una *clase*. Cada *objeto* tiene su propio estado, definido por los valores de sus atributos. Por ejemplo, un *objeto* de la *clase* **Coche** podría representar un coche específico con *marca* "Toyota", *modelo* "Corolla", y *color* "rojo".

POO : Ventajas

Características de las listas

Ventajas de la POO

Modularidad: Facilita la división del software en componentes pequeños y manejables.

Reutilización del Código: Las clases y objetos pueden reutilizarse en diferentes partes del programa o en otros proyectos.

Mantenimiento y Extensibilidad: Los sistemas orientados a objetos son más fáciles de mantener y extender gracias a su estructura clara y bien definida.

Correspondencia con el Mundo Real: La POO permite modelar problemas del mundo real de manera intuitiva, utilizando objetos que representan entidades del mundo real.

```
1 class Coche:  
2     pass
```

Para la definición de la clase utilizamos la palabra reservada “class” seguida del nombre de la clase en construcción.

El nombre debe ser descriptivo de la clase a crear y debe seguir la convención PascalCase (o UpperCamelCase) que determina que las primeras letras de cada una de las palabras estarán en mayúsculas, incluyendo la primera.

```
4 mi_coche = Coche()  
5 otro_coche = Coche()
```

Aunque la clase Coche aún no cumple ninguna funcionalidad y solo tiene una sentencia pass en su definición, podemos crear instancias(objetos) de clase con solo asignarlas a una(en realidad, podemos crear tantos objetos como necesitemos) variable con su nombre, un paréntesis de apertura y uno de cierre.

2

Atributos y métodos

Imaginate que estás diseñando un coche de juguete.

Los atributos son las características que definen el coche: el color, la marca, y el número de ruedas.

Los métodos son las acciones que el coche puede realizar: acelerar, frenar y tocar la bocina.

Vamos a aprender un poquito más sobre esto...

POO: Atributos

Ahora bien, cómo haríamos para darle características y funcionalidades a mi coche?

Para eso, vamos a ver en profundidad estos conceptos...

Atributos

Los atributos son variables que pertenecen a una clase o a sus instancias (objetos). Los atributos pueden ser de clase o de instancia.

```

1  class Coche:
2      # Constructor
3      def __init__(self, marca, modelo, color):
4          self.marca = marca
5          self.modelo = modelo
6          self.color = color

```

De instancia:

Los atributos de instancia son específicos de cada objeto creado a partir de una clase. Se definen generalmente en el método constructor `__init__`.

```

1  class Coche:
2      ruedas = 4
3
4      # Constructor
5      def __init__(self, marca, modelo, color):
6          self.marca = marca
7          self.modelo = modelo
8          self.color = color

```

De clase:

Los atributos de clase son compartidos por todas las instancias de la clase. Se definen directamente dentro del cuerpo de la clase.

Entonces, puedes crear clases con atributos que tengan un valor que van a compartir todas las instancias y/o también atributos que tengan un valor distinto en cada instancia según lo necesites.

POO: Atributos

Ahora vamos a ver como crear objetos y luego acceder a sus atributos.

En la clase siguiente veremos una forma distinta de hacerlo, por ahora lo haremos de esta manera...

Primero vemos la definición de la clase con un atributo de clase y tres de instancia.

Luego procedemos a crear dos objetos a partir de esta misma clase, como ves dentro de los paréntesis irán ubicados los estados o valores de nuestros atributos de instancia a los que previamente en el constructor definimos como parámetros.

Posterior a esto hacemos print de cada atributo de ambos objetos para ver sus diferencias(estados de atributos de instancia) y similitudes(estado del atributo de clase).

```
1 class Coche:
2     ruedas = 4
3
4     # Constructor
5     def __init__(self, marca, modelo, color):
6         self.marca = marca
7         self.modelo = modelo
8         self.color = color
9
10    # Crear un objeto de La clase Coche
11    mi_coche = Coche("Toyota", "Corolla", "Rojo")
12    otro_coche = Coche("Ford", "Ranger", "Azúl")
13
14    # Acceder al estado de Los atributos directamente
15    print('Atributos del primer coche:')
16    print(mi_coche.marca) # Toyota
17    print(mi_coche.modelo) # Corolla
18    print(mi_coche.color) # Rojo
19    print(mi_coche.ruedas) # 4
20
21    print('Atributos del segundo coche:')
22    print(otro_coche.marca) # Ford
23    print(otro_coche.modelo) # Ranger
24    print(otro_coche.color) # Azúl
25    print(otro_coche.ruedas) # 4
```


POO: Métodos

Los métodos son funciones definidas dentro de una clase que describen los comportamientos de los objetos. Son las acciones que los objetos pueden realizar.

Así como existen atributos de clase y atributos de instancia, lo mismo sucede con los métodos e incluso hay uno un poco distinto...

Métodos de instancia: Operan sobre instancias específicas como nuestros métodos `acelerar()` y `frenar()`.

```
1 class Coche:
2     ruedas = 4
3
4     # Constructor
5     def __init__(self, marca, modelo, color):
6         self.marca = marca
7         self.modelo = modelo
8         self.color = color
9
10    #Método
11    def acelerar(self):
12        print(f"El {self.marca} {self.modelo} está acelerando.")
13
14    #Método
15    def frenar(self):
16        print(f"El {self.marca} {self.modelo} está frenando.")
17
18
19    # Crear un objeto de la clase Coche
20    mi_coche = Coche("Toyota", "Corolla", "Rojo")
21    otro_coche = Coche("Ford", "Ranger", "Azúl")
22
23    # Usar métodos del objeto
24    mi_coche.acelerar() #El Toyota Corolla está acelerando.
25    mi_coche.frenar() #El Toyota Corolla está frenando.
26
27    otro_coche.acelerar() #El Ford Ranger está acelerando.
28    otro_coche.frenar() #El Ford Ranger está frenando.
```

POO: Métodos

Métodos de clase: son métodos que afectan a la clase en su conjunto y no a instancias específicas. Se definen con el decorador `@classmethod` y reciben `cls` como primer argumento, que representa la clase.

Te imaginas para qué lo usarías ?

Ojo! Porque cuando lo uses, modificarás el valor del atributo en todos los objetos creados a partir de esa clase!

```
1  class Coche:
2      ruedas = 4
3
4      # Constructor
5      def __init__(self, marca, modelo, color):
6          self.marca = marca
7          self.modelo = modelo
8          self.color = color
9
10     @classmethod
11     def incrementar_ruedas(cls):
12         cls.ruedas += 1
13
14
15     # Crear un objeto de la clase Coche
16     mi_coche = Coche("Toyota", "Corolla", "Rojo")
17     otro_coche = Coche("Ford", "Ranger", "Azúl")
18
19     print(mi_coche.ruedas) # 4
20     # Usar métodos de clase
21     Coche.incrementar_ruedas()
22     print(mi_coche.ruedas) # 5
```

POO: Métodos

Métodos estáticos: son funciones dentro de una clase que no dependen de las instancias ni de la clase misma. Se utilizan para agrupar funciones relacionadas conceptualmente con la clase, pero que no requieren acceso a los atributos de instancia o de clase. Se definen con el decorador `@staticmethod`.

A la hora de llamar a este método estaremos utilizando al nombre de la clase misma y pasando valores para hacer cálculos externos.

```
1 class Coche:
2     ruedas = 4
3
4     # Constructor
5     def __init__(self, marca, modelo, color):
6         self.marca = marca
7         self.modelo = modelo
8         self.color = color
9
10    @staticmethod
11    def calcular_distancia(velocidad, tiempo):
12        '''Calcula una distancia recorrida dadas la velocidad y el tiempo'''
13        return velocidad * tiempo
14
15    print(Coche.calcular_distancia(80,2)) # 160
```

POO: Ventajas

Modularidad

La POO fomenta la creación de programas modulares, donde cada clase representa una unidad de funcionalidad que puede ser desarrollada, probada y mantenida de manera independiente.

En una aplicación compleja, puedes tener diferentes clases para manejar usuarios, productos y pedidos. Cada clase encapsula su propio conjunto de datos y operaciones, lo que simplifica el desarrollo y la depuración.

```
1 class Usuario:
2     def __init__(self, nombre, email):
3         self.nombre = nombre
4         self.email = email
5
6     def mostrar_info(self):
7         print(f"Usuario: {self.nombre}, Email: {self.email}")
8
9 class Producto:
10     def __init__(self, nombre, precio):
11         self.nombre = nombre
12         self.precio = precio
13
14     def mostrar_info(self):
15         print(f"Producto: {self.nombre}, Precio: {self.precio}")
```

POO: Ventajas

Inmutabilidad

La inmutabilidad es una característica donde el estado de un objeto no puede cambiar después de su creación. En Python, los objetos inmutables incluyen tipos como int, float, str, y tuple.

Esto facilita el razonamiento sobre el código, especialmente en entornos concurrentes, y evita errores relacionados con cambios de estado inesperados.

Implementación: Para hacer una clase inmutable, evita definir setters para sus atributos o modifica los atributos para que sean constantes (aunque en Python esto se basa más en convenciones).

```
1 class Punto:
2     def __init__(self, x, y):
3         # Atributos "privados" para sugerir que no deben modificarse directamente
4         self._x = x
5         self._y = y
6
7     @property
8     def x(self):
9         return self._x
10
11    @property
12    def y(self):
13        return self._y
14
15    def __str__(self):
16        return f"Punto({self.x}, {self.y})"
17
18 # Crear un objeto de la clase Punto
19 p = Punto(3, 4)
20
21 # Acceder a sus atributos
22 print(p.x) # 3
23 print(p.y) # 4
24
25 # Intentar modificar los atributos lanzará un error
26 # p.x = 10 # Esto generará un error de atribución
```

POO: Ventajas

Abstracción:

La POO permite a los desarrolladores trabajar con niveles más altos de abstracción, ocultando los detalles de implementación y exponiendo solo lo necesario para el uso de una clase o un objeto.

Una clase **CuentaBancaria** puede ocultar los detalles de cómo se gestionan los depósitos y retiros internamente, proporcionando métodos públicos (la clase siguiente hablaremos sobre esto) que los usuarios pueden invocar.

```

1  class CuentaBancaria:
2      def __init__(self, saldo_inicial):
3          self._saldo = saldo_inicial
4
5      def depositar(self, cantidad):
6          self._saldo += cantidad
7
8      def retirar(self, cantidad):
9          if cantidad <= self._saldo:
10             self._saldo -= cantidad
11         else:
12             print("Fondos insuficientes")
13
14     def mostrar_saldo(self):
15         print(f"Saldo actual: {self._saldo}")
```

POO: Ventajas

Mantenibilidad:

Los programas orientados a objetos son generalmente más fáciles de mantener. Los cambios realizados en una clase no afectan a otras partes del programa, siempre que las interfaces se mantengan consistentes.

Si necesitas cambiar la forma en que una clase Coche calcula el consumo de combustible, puedes modificar el método correspondiente sin afectar a las demás clases o funciones del programa.

Facilidad para la colaboración:

La POO permite a los equipos de desarrollo trabajar en paralelo en diferentes partes del programa. Los desarrolladores pueden asignarse diferentes clases o módulos, trabajando de forma independiente.

```
1 class Coche:
2     def __init__(self, marca, consumo_por_km):
3         self.marca = marca
4         self.consumo_por_km = consumo_por_km
5
6     def calcular_consumo(self, distancia):
7         return self.consumo_por_km * distancia
```

En un proyecto de software grande, un desarrollador puede estar a cargo de las clases relacionadas con el front-end mientras otro trabaja en las clases del back-end.

3

Principios SOLID

Como ya te vas dando cuenta, en cada porción de código que escribimos tenemos reglas y principios a seguir...

Los principios SOLID proporcionan directrices para escribir código de calidad, flexible y fácil de mantener. Son especialmente útiles en el contexto de la Programación Orientada a Objetos.

POO: Principios SOLID

Concepto:

Los principios SOLID son un conjunto de cinco principios de diseño orientados a objetos que buscan mejorar la calidad y la mantenibilidad del software. Fueron introducidos por Robert C. Martin (también conocido como "Uncle Bob") y son fundamentales para crear sistemas de software bien estructurados y robustos.

S – Single Responsibility Principle

O – Open/Closed Principle

L – Liskov Substitution Principle

I – Interface Segregation Principle

D – Dependency Inversion Principle

POO: Principios SOLID

1. Principio de Responsabilidad Única (SRP):

Una clase debe tener **una sola** responsabilidad o razón para cambiar. Esto significa que una clase debe centrarse en una única función o tarea dentro del sistema, lo que facilita su comprensión, mantenimiento y modificación.

Supongamos que tenemos una clase que gestiona tanto la lógica de negocio de un Pedido como la presentación del mismo:

```
1 class Pedido:
2     def calcular_total(self):
3         # Lógica para calcular el total del pedido
4         pass
5
6     def imprimir_pedido(self):
7         # Lógica para imprimir el pedido
8         pass
```

Aplicando SRP, dividimos la clase en dos:

```
1 class Pedido:
2     def calcular_total(self):
3         # Lógica para calcular el total del pedido
4         pass
5
6 class ImpresoraDePedidos:
7     def imprimir(self, pedido):
8         # Lógica para imprimir el pedido
9         pass
```

POO: Principios SOLID

2. Principio de Abierto/Cerrado (OCP - Open/Closed Principle)

Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión pero cerradas para la modificación. Esto significa que el comportamiento de una entidad debe poder extenderse sin modificar su código fuente.

Supongamos que tenemos una clase `CalculadoraDeSalarios` que calcula salarios según el tipo de empleado:

```
1 class CalculadoraDeSalarios:
2     def calcular(self, empleado):
3         if empleado.tipo == 'comisionado':
4             return empleado.ventas * 0.10
5         elif empleado.tipo == 'asalariado':
6             return empleado.salario_fijo
```

Aplicando OCP, utilizamos herencia o composición para extender el comportamiento sin modificar el código existente:

```
1 class CalculadoraDeSalarios:
2     def calcular(self, empleado):
3         return empleado.calcular_salario()
4
5 class EmpleadoComisionado:
6     def calcular_salario(self):
7         return self.ventas * 0.10
8
9 class EmpleadoAsalariado:
10    def calcular_salario(self):
11        return self.salario_fijo
```

POO: Principios SOLID

3. Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle)

Los objetos de una clase derivada deben poder sustituir a los objetos de su clase base sin alterar el funcionamiento del programa. Este principio garantiza que las clases derivadas respeten el contrato establecido por sus clases base.

Considera una clase `Animal` con un método `hacer_sonido`:

```

1  class Animal:
2      def hacer_sonido(self):
3          pass
4
5  class Perro(Animal):
6      def hacer_sonido(self):
7          return "Ladra"
8
9  class Gato(Animal):
10     def hacer_sonido(self):
11         return "Maúlla"

```

Las clases `Perro` y `Gato` pueden sustituir a `Animal` sin problemas, ya que cumplen con el comportamiento esperado.

POO: Principios SOLID

4. Principio de Segregación de Interfaces (ISP - Interface Segregation Principle)

Los clientes no deben estar forzados a depender de interfaces que no utilizan. Es mejor tener varias interfaces pequeñas y específicas que una sola interfaz grande y genérica.

Supongamos que tenemos una interfaz Vehiculo con métodos que no todos los vehículos utilizan:

```
1 class Vehiculo:
2     def encender(self):
3         pass
4
5     def volar(self):
6         pass
7
8     def navegar(self):
9         pass
```

Aplicando ISP, dividimos la interfaz en varias más específicas:

```
1 class Vehiculo:
2     def encender(self):
3         pass
4
5 class Avion(Vehiculo):
6     def volar(self):
7         pass
8
9 class Barco(Vehiculo):
10    def navegar(self):
11        pass
```

POO: Principios SOLID

5. Principio de Inversión de Dependencias (DIP - Dependency Inversion Principle)

Las clases de alto nivel no deben depender de clases de bajo nivel; ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles; los detalles deben depender de abstracciones.

Supongamos que una clase ControlRemoto depende directamente de una clase Television:
(En este diseño, ControlRemoto está estrechamente acoplado a Television, lo que significa que cualquier cambio en la implementación de Television podría requerir cambios en ControlRemoto.)

```
1 class Television:
2     def encender(self):
3         pass
4
5 class ControlRemoto:
6     def __init__(self, tv):
7         self.tv = tv
8
9     def presionar_boton(self):
10        self.tv.encender()
```

Aplicando DIP, usamos una abstracción para eliminar la dependencia directa:

```
1 class Dispositivo:
2     def encender(self):
3         pass
4
5 class Television(Dispositivo):
6     def encender(self):
7         pass
8
9 class ControlRemoto:
10    def __init__(self, dispositivo):
11        self.dispositivo = dispositivo
12
13    def presionar_boton(self):
14        self.dispositivo.encender()
```



Lo que vimos hoy 🤔

Hasta acá pudimos ver el uso de POO en python y su gran utilidad a la hora de desarrollar nuestros programas!

Además aprendiste los principios fundamentales a seguir para escribir un código limpio y de calidad para tus próximos programas a desarrollar :)

Te espero en la siguiente clase para ver unos conceptos más que son importantes para la POO!



**¡Nos vemos
En la próxima
clase!**

