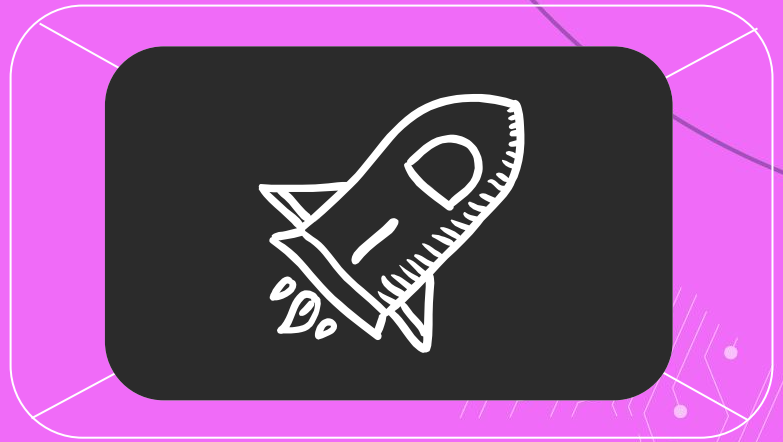
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

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

¿Cuáles son sus ventajas?

→ **Modularidad:** Facilita la división del software en componentes pequeños y manejables (concepto que venimos viendo).

→ **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 variable con su nombre, un paréntesis de apertura y uno de cierre (en realidad, podemos crear tantos objetos como necesitemos).

2

Atributos y métodos

Imagináte 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

Atributos o *características*

Los atributos son variables que pertenecen a una clase o a sus instancias (objetos). Los atributos pueden ser de clase o de instancia.
Entonces, ¿cómo haríamos para darle características y funcionalidades a mi coche?

```python
class Coche:
    # Constructor
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        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__ ()*** .

```python
class Coche:
    ruedas = 4

    # Constructor
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        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, podés 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, pero 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 vés, 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).

```python
class Coche:
    ruedas = 4

    # Constructor
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

# Crear un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla", "Rojo")
otro_coche = Coche("Ford","Ranger", "Azúl")

# Acceder al estado de los atributos directamente
print('Atributos del primer coche:')
print(mi_coche.marca)  # Toyota
print(mi_coche.modelo) # Corolla
print(mi_coche.color)  # Rojo
print(mi_coche.ruedas)  # 4

print('Atributos del segundo coche:')
print(otro_coche.marca)  # Ford
print(otro_coche.modelo) # Ranger
print(otro_coche.color)  # Azúl
print(otro_coche.ruedas)  # 4
```

POO: Métodos

Métodos o *acciones*

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()*.

```python
class Coche:
    ruedas = 4

    # Constructor
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    #Método
    def acelerar(self):
        print(f"El {self.marca} {self.modelo} está acelerando.")

    #Método
    def frenar(self):
        print(f"El {self.marca} {self.modelo} está frenando.")


# Crear un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla", "Rojo")
otro_coche = Coche("Ford","Ranger", "Azúl")

# Usar métodos del objeto
mi_coche.acelerar() #El Toyota Corolla está acelerando.
mi_coche.frenar() #El Toyota Corolla está frenando.

otro_coche.acelerar() #El Ford Ranger está acelerando.
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 imaginás para qué lo usarías ?

¡Ojo! Porque cuando lo usés, vas a modificar el valor del atributo en todos los objetos creados a partir de esa clase.

```python
class Coche:
    ruedas = 4

    # Constructor
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    @classmethod
    def incrementar_ruedas(cls):
        cls.ruedas += 1


# Crear un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla", "Rojo")
otro_coche = Coche("Ford","Ranger", "Azúl")

print(mi_coche.ruedas) # 4
# Usar métodos de clase
Coche.incrementar_ruedas()
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.

```python
class Coche:
    ruedas = 4

    # Constructor
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    @staticmethod
    def calcular_distancia(velocidad,tiempo):
        '''Calcula una distancia recorrida dadas la velocidad y el tiempo'''
        return velocidad * tiempo

print(Coche.calcular_distancia(80,2)) # 160
```

POO: Método constructor

Método *constructor*

Vamos a hacer una especial referencia a este método, ya que es un método especial y es el encargado de dar vida a los objetos.
Y, para que lo comprendas lo mejor posible, vamos a tomar como ejemplo un juego, o, la creación de un juego.

Entonces, vamos a imaginar que estamos creando un personaje para un juego, o videojuego. Ese personaje necesita un nombre, una edad, una apariencia y habilidades especiales, ¿no?

El método ___init___ es como el creador de personajes del videojuego.

Así que para crear el personaje del juego, tenés que darle toda la información para que sea único.

El *método* ___init___ hace lo mismo con los objetos en programación.

Por lo que, vamos a decir que, una clase es como un molde que te permite definir cómo querés que sea un objeto cuando lo creás, y con ___init___ vas a darle vida a ese objeto.

Para dar un ejemplo, este juego va a ser de animales con superpoderes, y para este ejemplo, el tipo de animal que vamos a crear va a ser un perro. 🐕

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

catdog = Perro('CatDog', 'Teckel', 'Súper elasticidad')
print(catdog.nombre)
print(catdog.raza)
print(catdog.poder)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS
CatDog
Teckel
Super elasticidad
```

POO: Método constructor

→ **class Perro()**: Creamos el molde.
→ **__init__**: Le damos vida a cada perro nuevo que creemos.
→ **nombre, raza, poder** : Para hablar muy "en criollo", estos van a ser los espacios en blanco a rellenar en un formulario.
Una vez llenado el "formulario", como en el ejemplo, ya tenemos nuestro personaje del juego, que es un super perro llamado "CatDog", su raza es "Teckel" y de superpoder tiene súper elasticidad.

¿Lo vas comprendiendo mejor? Ahora, lo que te habrás estado preguntando es ¿qué es ***self***?

→ **self** : En el constructor, self hace autorreferencia al objeto que se está creando, como un espejo podríamos decir. Cuando estamos diciendo "self.nombre = 'CatDog' ", estamos diciendo: "En este objeto que estoy construyendo ahora mismo, el atributo *nombre* tendrá el valor "CatDog".

¿Por qué se llama *self*?
Es simplemente una convención. Podría llamarse de otra manera, pero *self* es la más común y ayuda a recordar que se refiere al objeto en sí mismo.

Y ¿Por qué se llama __init__?
También es una convención establecida en Python para nombrar al método constructor. Los dos guiones bajos al principio y al final indican que es un método especial, reservado para un propósito específico.
La palabra "init" es una abreviatura de "initialize", que significa "inicializar". Y eso es exactamente lo que hace este método: inicializa el objeto.

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, podés 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.

```python
class Usuario:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email

    def mostrar_info(self):
        print(f"Usuario: {self.nombre}, Email: {self.email}")

class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

    def mostrar_info(self):
        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, evitá definir *setters* para sus atributos o modificá los atributos para que sean constantes (aunque en Python esto se basa más en convenciones).

```python
class Punto:
    def __init__(self, x, y):
        # Atributos "privados" para sugerir que no deben modificarse directamente
        self._x = x
        self._y = y

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

    def __str__(self):
        return f"Punto({self.x}, {self.y})"

# Crear un objeto de la clase Punto
p = Punto(3, 4)

# Acceder a sus atributos
print(p.x)  # 3
print(p.y)  # 4

# Intentar modificar los atributos lanzará un error
# 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.

```python
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self._saldo = saldo_inicial

    def depositar(self, cantidad):
        self._saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self._saldo:
            self._saldo -= cantidad
        else:
            print("Fondos insuficientes")

    def mostrar_saldo(self):
        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 necesitás cambiar la forma en que una *clase Coche* calcula el consumo de combustible, podés 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.

```python
class Coche:
    def __init__(self, marca, consumo_por_km):
        self.marca = marca
        self.consumo_por_km = consumo_por_km

    def calcular_consumo(self, distancia):
        return self.consumo_por_km * distancia
```

En un proyecto de software grande, un desarrollador puede estar a cargo de las clases relacionadas con el frontend mientras otro trabaja en las clases del backend.

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:

```python
class Pedido:
    def calcular_total(self):
        # Lógica para calcular el total del pedido
        pass

    def imprimir_pedido(self):
        # Lógica para imprimir el pedido
        pass
```

Aplicando SRP, dividimos la clase en dos:

```python
class Pedido:
    def calcular_total(self):
        # Lógica para calcular el total del pedido
        pass

class ImpresoraDePedidos:
    def imprimir(self, pedido):
        # Lógica para imprimir el pedido
        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:

```python
class CalculadoraDeSalarios:
    def calcular(self, empleado):
        if empleado.tipo == 'comisionado':
            return empleado.ventas * 0.10
        elif empleado.tipo == 'asalariado':
            return empleado.salario_fijo
```

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

```python
class CalculadoraDeSalarios:
    def calcular(self, empleado):
        return empleado.calcular_salario()

class EmpleadoComisionado:
    def calcular_salario(self):
        return self.ventas * 0.10

class EmpleadoAsalariado:
    def calcular_salario(self):
        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.

Considerá una *clase Animal* con un método *hacer_sonido*:

```python
class Animal:
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "Ladra"

class Gato(Animal):
    def hacer_sonido(self):
        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:

```python
class Vehiculo:
    def encender(self):
        pass

    def volar(self):
        pass

    def navegar(self):
        pass
```

Aplicando ISP, dividimos la interfaz en varias más específicas:

```python
class Vehiculo:
    def encender(self):
        pass

class Avion(Vehiculo):
    def volar(self):
        pass

class Barco(Vehiculo):
    def navegar(self):
        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*.)

```python
class Television:
    def encender(self):
        pass

class ControlRemoto:
    def __init__(self, tv):
        self.tv = tv

    def presionar_boton(self):
        self.tv.encender()
```

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

```python
class Dispositivo:
    def encender(self):
        pass

class Television(Dispositivo):
    def encender(self):
        pass

class ControlRemoto:
    def __init__(self, dispositivo):
        self.dispositivo = dispositivo

    def presionar_boton(self):
        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 esperamos en la siguiente clase para ver otros conceptos importantes para la ¡POO!

¡Nos vemos En la próxima clase!