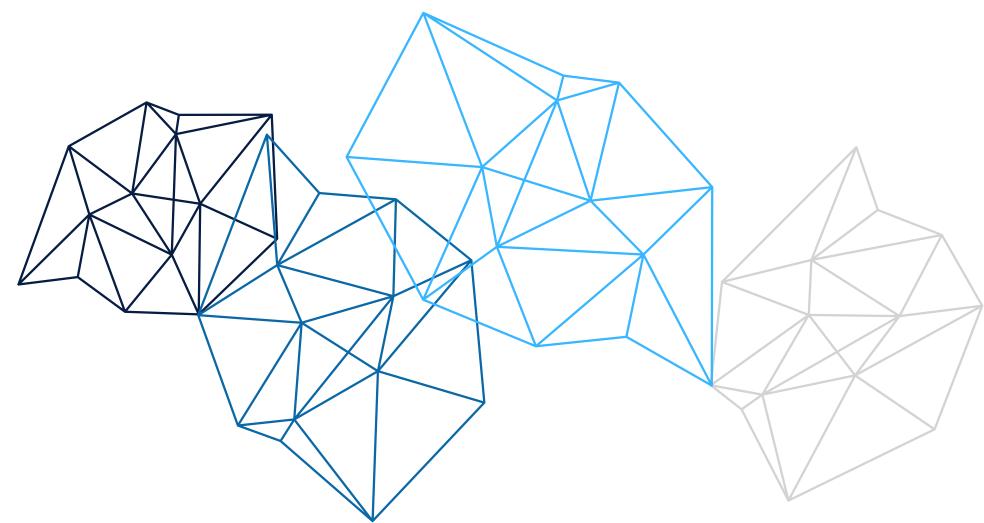


Programación Orientada a Objetos

La **programación orientada a objetos (POO)** es un paradigma de programación que se basa en el concepto de "objetos" y sus interacciones para resolver problemas y desarrollar software.

Python es un lenguaje de programación que admite POO de manera nativa y facilita la implementación de este enfoque.

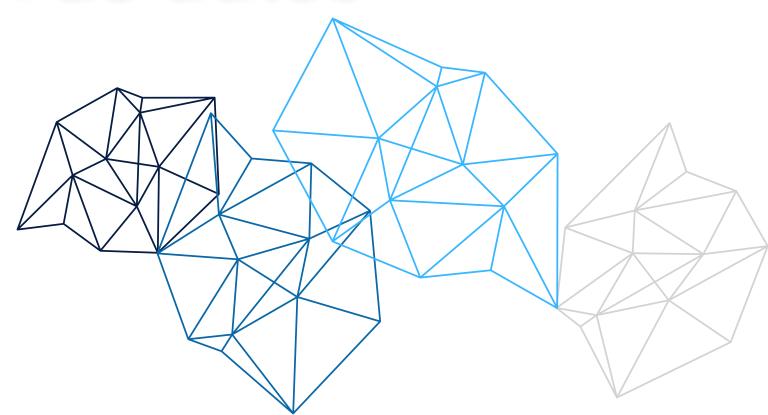
En POO, los **objetos** son **instancias** de una clase. Una **clase** es una **plantilla** o estructura que define las **propiedades (atributos)** y los **comportamientos (métodos)** de un objeto. Los atributos representan las características del objeto, mientras que los métodos definen las acciones que puede realizar el objeto.



¿Qué es un paradigma?

Un paradigma en la programación es una forma particular de enfoque o modelo que define cómo se deben diseñar, estructurar y organizar los programas de computadora. Representa una filosofía o conjunto de principios que guían la manera en que se desarrolla el software. Cada paradigma tiene sus propias reglas, conceptos y técnicas que determinan la forma en que se escriben, organizan y manipulan las instrucciones de programación. Algunos de los paradigmas más comunes son:

1. Programación estructurada: Se enfoca en dividir el programa en estructuras lógicas más pequeñas, como funciones y bloques de código, para mejorar la claridad y la mantenibilidad del código.
2. **Programación orientada a objetos (Poo)**: Se basa en la creación de objetos que encapsulan datos y comportamientos relacionados. Los objetos son instancias de clases, y la interacción entre ellos se realiza mediante mensajes y métodos.
3. Programación funcional: Se centra en el uso de funciones puras y evita el uso de estados y datos mutables. Se enfatiza en la composición de funciones y la manipulación de datos inmutables.



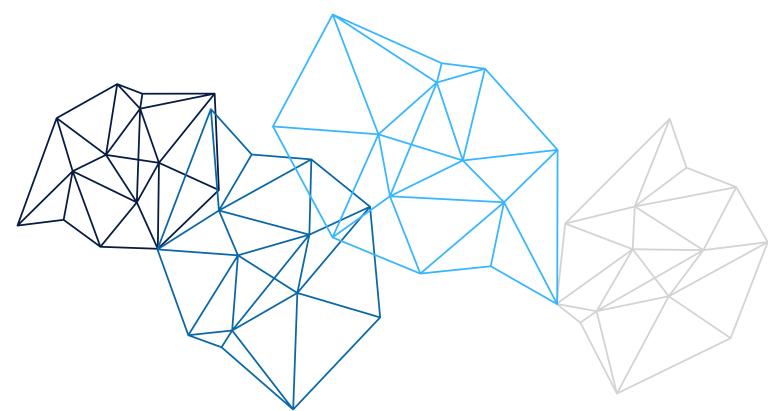
Estos son solo algunos ejemplos de paradigmas de programación, pero existen otros enfoques y combinaciones de ellos. Cada paradigma tiene sus ventajas y desventajas, y la elección del paradigma adecuado depende del problema a resolver y las características del proyecto. Algunos lenguajes de programación son más adecuados para un paradigma específico, mientras que otros admiten múltiples paradigmas.

¿Qué es una CLASE?

Una clase es un modelo que define las propiedades y comportamiento de un tipo de objeto concreto, en la instanciación se hace la lectura de estas definiciones y se crea un objeto a partir de ellas. Las clases son abstracciones que representan a un conjunto de objetos con un comportamiento e interfaz común.

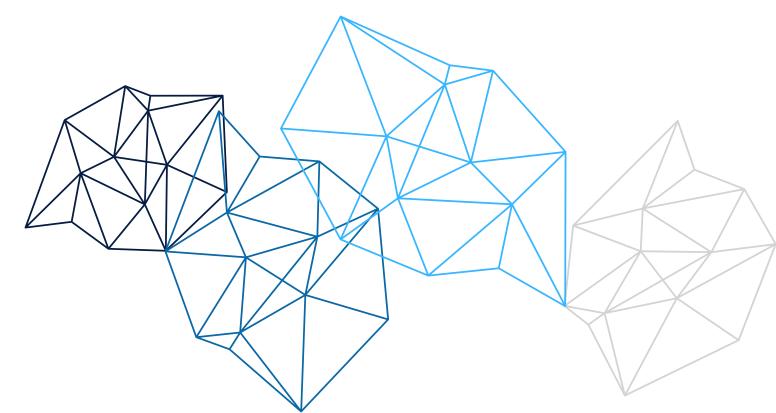
Una clase se compone de dos partes:

- Atributos: esto es, los datos que se refieren al estado del objeto.
- Métodos: son funciones que pueden aplicarse a objetos.



¿Qué es un objeto?

Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa). En realidad, un objeto es una instancia de una clase, por lo que se pueden intercambiar los términos objeto o instancia. Un ejemplo de objeto podría ser un automóvil, en el que tendríamos atributos como la marca, el número de puertas o el color y métodos como arrancar y parar. O bien cualquier otra combinación de atributos y métodos según lo que fuera relevante para nuestro programa.



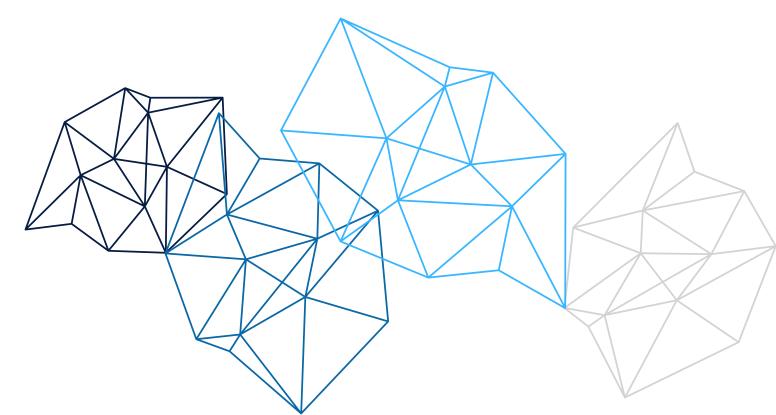
¿Qué es un método?

Un método es una función que pertenece a una clase o a un objeto. Los métodos definen los comportamientos o las acciones que pueden realizar los objetos de una clase específica.

Los métodos representan las acciones que los objetos pueden realizar o las operaciones que se pueden realizar sobre ellos.

Cada método está definido dentro de una clase y se puede acceder a él a través de una instancia (objeto) de esa clase. Los métodos pueden acceder a los atributos de la instancia y manipularlos según sea necesario. También pueden tener parámetros y pueden devolver un valor.

Los métodos son fundamentales en la POO, ya que permiten que los objetos interactúen entre sí y realicen acciones específicas de acuerdo con su comportamiento definido en la clase.



Un ejemplo para empezar a verlo en código:

#Crear la clase para luego poder instanciar Personas

class Persona:

 #Constructor

 def __init__(self, nombre, edad):

 self.nombre = nombre

 self.edad = edad

 #Método

 def saludar(self):

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

Crear una instancia de la clase Persona

personal1 = Persona("Juan", 25)

Acceder a los atributos de la instancia

print(personal1.nombre) # Imprime: Juan

print(personal1.edad) # Imprime: 25

Llamar a un método de la instancia

personal1.saludar() # Imprime: Hola, mi nombre es Juan y tengo 25 años.



En el ejemplo anterior, se define la clase **Persona** con el método especial `__init__()` que se llama constructor. El constructor se ejecuta automáticamente cuando se crea una nueva instancia de la clase y se utiliza para inicializar los atributos de la instancia. En este caso, el constructor toma dos parámetros (**nombre** y **edad**) y asigna esos valores a los atributos correspondientes (**self.nombre** y **self.edad**).

Como vemos el primer parámetro de `__init__` y del resto de métodos de la clase es siempre **self** que sirve para referirse al objeto actual. Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local `mi_var` de un atributo del objeto **self.mi_var**.

El método **saludar()** es un método regular de la clase **Persona** que imprime un saludo utilizando los atributos de la instancia (`self.nombre` y `self.edad`).

Luego, se crea una instancia de la clase **Persona** llamada `personal` pasando los valores "Juan" y 25 como argumentos al constructor. Se puede acceder a los atributos de la instancia utilizando la notación de punto (**personal.nombre**, **personal.edad**). También se puede llamar a los métodos de la instancia utilizando la misma notación de punto (**personal.saludar()**).

Abstracción

La abstracción en Python, al igual que en otros lenguajes de programación orientada a objetos, es un concepto clave que permite representar entidades del mundo real o conceptos complejos como objetos con propiedades y comportamientos específicos.

Implica enfocarse en los aspectos esenciales y relevantes de un objeto y ocultar los detalles innecesarios. Permite crear clases y objetos que encapsulan datos y comportamientos relacionados, proporcionando una interfaz clara y simplificada para interactuar con ellos.

La técnica de la abstracción consiste en **aislar** un determinado elemento sobre su contexto o el resto de elementos que lo acompañan y que posiblemente no sean válidos para nuestro propósito. En POO solo necesita saber cómo interaccionar con los objetos, no necesita conocer los detalles de cómo está implementada la clase a partir de la cual se instancia el objeto. Sólo necesita conocer su interfaz pública. El nivel de abstracción puede ser bajo (en un objeto se manipulan datos y métodos individualmente), o alto (en un objeto solo se usan sus métodos de servicio).

Encapsulación

La encapsulación es una forma de abstracción, además es un mecanismo para llevar a la práctica la abstracción. Cuando una clase existe (se define), se crean objetos a partir de ella, y se usan dichos objetos llamando los métodos necesarios. Es decir, crea objetos para usar los servicios que nos proporciona la clase a través de sus métodos. No necesita saber cómo trabaja el objeto, ni saber las variables que usa, ni el código que contiene. Este principio otorga beneficios importantes para el desarrollo de software:

- **Modularidad:** El código fuente para un objeto puede ser escrito y mantenido independientemente del código fuente para otros objetos. Además, un objeto puede ser fácilmente pasado de un lugar a otro del sistema.
- **Ocultamiento de información:** Un objeto tiene una interfaz pública que otros objetos pueden utilizar para comunicarse con él. El objeto puede mantener información y métodos privados que pueden ser modificados a cualquier tiempo sin afectar los otros objetos que dependen de él. No es necesario entender cómo usar el mecanismo de los engranajes de una bicicleta para poder utilizarlo.

Encapsulación

La encapsulación describe el hecho de que los objetos se usan como cajas negras. Así, un objeto encapsula datos y métodos, que están dentro del objeto.

- **Interfaz pública de una clase:** Es el conjunto de métodos (métodos de servicio) que sirve para que los objetos de una clase proporcionen sus servicios. Estos servicios son los que pueden ser llamados por un cliente.
- **Métodos de soporte:** Son métodos adicionales en un objeto que no definen un servicio utilizable por un cliente, pero que ayudan a otros métodos en sus tareas.

La encapsulación es un mecanismo de control que restringe la capacidad de modificar el estado (el conjunto de propiedades, atributos o datos) de un objeto sólo por medio de los métodos del propio objeto, permitiendo que los atributos de un objeto puedan ocultarse (superficialmente) para que no sean accedidos desde fuera de la definición de una clase. Para ello, es necesario nombrar los atributos con un prefijo de doble subrayado: atributo.

Encapsulamiento y niveles de privacidad

En encapsulamiento nos permite definir niveles de privacidad para señalar qué métodos y atributos no deben utilizarse fuera de la clase y así evitar exponer excesivamente los detalles de la implementación de una clase o un objeto. Esto se consigue en otros lenguajes de programación como Java utilizando modificadores de acceso que definen si cualquiera puede acceder a esa función o variable(`public`) o si está restringido el acceso a la propia clase (`private`). En Python no existen los modificadores de acceso, y lo que se suele hacer es usar convenciones a la hora de nombrar métodos y atributos, de forma que se señale su carácter privado, para su exclusión del espacio de nombres, o para indicar una función especial, normalmente asociada a funcionalidades estándar del lenguaje.

- `_nombre`: Los nombres que comienzan con un único guión bajo indican de forma débil un uso interno. Además, estos nombres no se incorporan en el espacio de nombres de un módulo al importarlo con `"from ... import *`.
- `__nombre`: Los nombres que empiezan por dos guiones bajos (y no termina también con dos guiones bajos) indican su uso privado en la clase.

Encapsulamiento y niveles de privacidad

- `__nombre__`: Los nombres que empiezan y acaban con dos guiones bajos indican atributos "mágicos", de uso especial y que residen en espacios de nombres que puede manipular el usuario. Solamente deben usarse en la manera que se describe en la documentación de Python y debe evitarse la creación de nuevos atributos de este tipo.

Algunos ejemplos de nombres "singulares" de este tipo son:

- `__init__`, método de inicialización de objetos
- `__del__`, método de destrucción de objetos
- `__doc__`, cadena de documentación de módulos, clases...
- `__class__`, nombre de la clase
- `__str__`, método que devuelve una descripción de la clase como cadena de texto
- `__repr__`, método que devuelve una representación de la clase como cadena de texto
- `__module__`, módulo al que pertenece la clase

En el siguiente ejemplo sólo se imprimirá la cadena correspondiente al método publico(), mientras que al intentar llamar al método __privado() Python lanzará una excepción debido a que no existe, evidentemente existe pero no lo podemos ver porque es privado.

```
class Ejemplo:
```

```
    def publico(self):
        print("Publico")
    def __privado(self):
        print("Privado")
```

```
ej = Ejemplo()
```

```
ej.publico()
```

```
ej.__privado()
```

Resultado

Público

Traceback (most recent call last): File "", line 11, in AttributeError: 'Ejemplo' object has no attribute '__privado'



Este mecanismo se basa en que los nombres que comienzan con un doble guión bajo se renombran para incluir el nombre de la clase (característica que se conoce con el nombre de name mangling). Esto implica que el método o atributo no es realmente privado, y podemos acceder a él mediante una pequeña trampa:

ej._Ejemplo__privado()

En ocasiones también puede suceder que queramos permitir el acceso a algún atributo de nuestro objeto, pero que este se produzca de forma controlada. Para esto podemos escribir métodos cuyo único cometido sea este, métodos que normalmente, por convención, tienen nombres como getVariable y setVariable; de ahí que se conozcan también con el nombre de getters y setters.

class Fecha():

```
def __init__(self):
    self.__dia = 1
def getDia(self):
    return self.__dia
def setDia(self, dia):
    if dia > 0 and dia < 31:
        self.__dia = dia
    else:
        print("Error")
```



```
mi_fecha = Fecha()  
mi_fecha.setDia(33)  
mi_fecha.__dia= 30 #Acá podríamos usar una trampa con mi_fecha._Fecha__dia =  
30  
print(mi_fecha.getDia())  
#Resultado  
#Error  
#1
```

Esto se podría simplificar mediante propiedades(decorador property), que abstraen al usuario del hecho de que se está utilizando métodos accesores para obtener y modificar los valores del atributo

Ej: dia = property(getDia, setDia)

Luego:

mi_fecha.dia = 30



INFORMATARIO

Hacia una mejor industria informática