VII. Programación orientada a objetos

Objetivo:
Introducción a POO con lenguaje de programación Python.

a. Clases
b. Objetos
c. herencia
d. herencia múltiple
e. encapsulación

POO

¿Qué es la Programación Orientada a Objetos?

La Programación Orientada a Objetos (POO) es un paradigma de programación, es decir, un modelo o un estilo de programación que nos da unas guías sobre cómo trabajar con él. Se basa en el concepto de clases y objetos. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (clases) para crear instancias individuales de objetos.

POO

Con el paradigma de **Programación Orientado a Objetos** lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. Esto nos ayuda muchísimo en sistemas grandes, ya que en vez de pensar en funciones, pensamos en las relaciones o interacciones de los diferentes componentes del sistema.

POO

La Programación Orientada a objetos permite que el código sea reutilizable, organizado y fácil de mantener. Sigue el principio de desarrollo de software utilizado por muchos programadores DRY (Don’t Repeat Yourself), para evitar duplicar el código y crear de esta manera programas eficientes. Además, evita el acceso no deseado a los datos o la exposición de código propietario mediante la encapsulación y la abstracción.

POO

¿Cómo se crean los programas orientados a objetos? En resumen, consiste en hacer clases y crear objetos a partir de estas clases. Las clases forman el modelo a partir del que se estructuran los datos y los comportamientos.

El primer y más importante concepto de la POO es la **distinción entre clase y objeto**.

Objetos y Clases

Una **clase** es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo. Por ejemplo, una clase para representar a animales puede llamarse ‘animal’ y tener una serie de **atributos**, como ‘nombre’ o ‘edad’ (que normalmente son propiedades), y una serie con los comportamientos que estos pueden tener, como caminar o comer, y que a su vez se implementan como métodos de la clase (funciones).

Clases y Objetos

Las **clases** proveen una forma de empaquetar datos y funcionalidad juntos. Al crear una nueva clase, se crea un nuevo *tipo* de objeto, permitiendo crear nuevas *instancias* de ese tipo. Cada instancia de clase puede tener atributos adjuntos para mantener su estado. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

Un **objeto** es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de variables llamadas atributos, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de métodos del objeto.

Sintaxis

Para crear una clase vamos a emplear la palabra reservada class seguido de un nombre escrito en minúscula, a excepción de la primera letra de cada palabra, que se escribe en mayúscula, y sin guiones bajos. (Clase vacía)

```
class Alumno:
    pass
```

Sintaxis

Sabemos que las clases pueden contener funciones, a las que llamamos *métodos*. Para ello vamos a usar la misma nomenclatura que aprendimos en el apartado anterior, con la diferencia que esta vez todo nuestro código estará indentado cuatro espacios, para indicar que queremos ubicarlo dentro de la clase.

```
class Alumno:

    def saludar(self):
        """Imprime un saludo en pantalla."""
        print("¡Hola, mundo!")
```

Todas las funciones definidas dentro de una clase deberán tener, al menos, un argumento, que por convención se le llama self y es una referencia a la *instancia* de la clase.

Sintaxis

Creamos una instancia de nuestra clase.

```
alumno = Alumno()

alumno.saludar()
```

Sintaxis

Una clase también puede contener variables, a las que se conoce con el nombre de *atributos*. Para crear atributos definimos un método especial llamado __init__(), que es invocado por Python automáticamente siempre que se crea una instancia de la clase (conocido también como *constructor* o *inicializador*).

```
class Alumno:

    def __init__(self):
        self.nombre = "Pablo"

    def saludar(self):
        """Imprime un saludo en pantalla."""
        print(f"¡Hola, {self.nombre}!")

alumno = Alumno()
```

Sintaxis

Para el código anterior es conveniente permitir que, al definir una instancia, se pase como argumento el nombre del alumno y éste se almacene en el atributo self.nombre.

```
class Alumno:

    def __init__(self, nombre):
        self.nombre = nombre

    def saludar(self):
        """Imprime un saludo en pantalla."""
        print(f"¡Hola, {self.nombre}!")


alumno = Alumno("Pablo")
alumno.saludar()
```

Herencia

La herencia define **relaciones jerárquicas entre clases**, de forma que atributos y métodos comunes puedan ser reutilizados. Las clases principales extienden atributos y comportamientos a las clases secundarias. A través de la definición en una clase de los atributos y comportamientos básicos, se pueden crear clases secundarias, ampliando así la funcionalidad de la clase principal y agregando atributos y comportamientos adicionales.

Herencia

La herencia es una herramienta fundamental para la orientación a objetos. Permite definir jerarquías de clases que comparten diversos métodos y atributos. Por ejemplo, consideremos la siguiente clases:

```python
class Rectangulo:
    """
    Define un rectángulo según su base y su altura.
    """
    def __init__(self, b, h):
        self.b = b
        self.h = h

    def area(self):
        return self.b * self.h

rectangulo = Rectangulo(20, 10)
print("Área del rectángulo: ", rectangulo.area())
```

```python
class Triangulo:
    """
    Define un triángulo según su base y su altura.
    """
    def __init__(self, b, h):
        self.b = b
        self.h = h

    def area(self):
        return (self.b * self.h) / 2
```

Herencia

Los códigos son muy similares, a excepción del método area(). Pero dado que el método __init__() es el mismo, podemos abstraerlo en una *clase padre* de la cual *hereden* tanto Rectángulo como Triángulo.

```
class Poligono:
    """
    Define un polígono según su base y su altura.
    """
    def __init__(self, b, h):
        self.b = b
        self.h = h

class Rectangulo(Poligono):

    def area(self):
        return self.b * self.h

class Triangulo(Poligono):

    def area(self):
        return (self.b * self.h) /  2

rectangulo = Rectangulo( 20, 10)
triangulo = Triangulo( 20, 12)

print("Área del rectángulo: " , rectangulo.area())
print("Área del triángulo:" , triangulo.area())
```

Herencia

Una misma clase puede heredar de varias clases (*herencia múltiple*); en ese caso, se especifican los nombres separados por comas.

```
class ClaseA(ClaseB, ClaseC, ClaseD):

    pass
```

Encapsulación

La encapsulación contiene **toda la información importante de un objeto dentro del mismo** y solo expone la información seleccionada al mundo exterior.

Esta propiedad permite asegurar que la información de un objeto esté oculta para el mundo exterior, agrupando en una Clase las características o atributos que cuentan con un acceso privado, y los comportamientos o métodos que presentan un acceso público.

La encapsulación de cada objeto es responsable de su propia información y de su propio estado. La única forma en la que este se puede modificar es mediante los propios métodos del objeto. Por lo tanto, los atributos internos de un objeto deberían ser **inaccesibles desde fuera**, pudiéndose modificar sólo llamando a las funciones correspondientes. Con esto conseguimos mantener el estado a salvo de usos indebidos o que puedan resultar inesperados.

Usamos de ejemplo un coche para explicar la encapsulación. El coche comparte información pública a través de las luces de freno o intermitentes para indicar los giros (interfaz pública). Por el contrario, tenemos la interfaz interna, que sería el mecanismo propulsor del coche, que está oculto bajo el capó. Cuando se conduce un automóvil es necesario indicar a otros conductores tus movimientos, pero no exponer datos privados sobre el tipo de carburante o la temperatura del motor, ya que son muchos datos, lo que confundiría al resto de conductores.

Abstracción

La abstracción es cuando **el usuario interactúa solo con los atributos y métodos seleccionados de un objeto**, utilizando herramientas simplificadas de alto nivel para acceder a un objeto complejo.

En la programación orientada a objetos, los programas suelen ser muy grandes y los objetos se comunican mucho entre sí. El concepto de abstracción **facilita el mantenimiento de un código de gran tamaño**, donde a lo largo del tiempo pueden surgir diferentes cambios.

Así, la abstracción se basa en usar **cosas simples para representar la complejidad**. Los objetos y las clases representan código subyacente, ocultando los detalles complejos al usuario. Por consiguiente, supone una extensión de la encapsulación. Siguiendo con el ejemplo del coche, no es necesario que conozcas todos los detalles de cómo funciona el motor para poder conducirlo.

Poliformismo

El polimorfismo consiste en **diseñar objetos para compartir comportamientos**, lo que nos permite procesar objetos de diferentes maneras. Es la capacidad de presentar la misma interfaz para diferentes formas subyacentes o tipos de datos.

Al utilizar la herencia, los objetos pueden anular los comportamientos principales compartidos, con comportamientos secundarios específicos. El polimorfismo permite que el mismo método ejecute diferentes comportamientos de dos formas: anulación de método y sobrecarga de método.

Getters / Setters

Los “Getters” y “Setters” se utilizan en POO para garantizar el principio de la encapsulación de datos.
Claramente el **getter** se emplea para obtener los datos y el **setter** para cambiar el valor de los datos. Son decoradores y se identifican por tener un @.

Por lo general, estos se usan en Python:

- Si queremos añadir una lógica de validación para obtener y establecer un valor.
- Para evitar el acceso directo a un atributo de clase (un usuario externo no puede acceder directamente a las variables privadas ni modificarlas).

“Getters / Setters” Sintaxis

```
class Humano:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
        #self.valEdad()

    @property
    def listaNombre(self):
        return self.nombre

    @listaNombre.setter
    def listaNombre(self, nuevoNombre):
        self.nombre = nuevoNombre

    @property
    def listaEdad(self):
        return self.edad

    @listaEdad.setter
    def listaEdad(self,nuevaEdad):
        self.edad = nuevaEdad
        #self.valEdad()
```