



Python II

UNIDAD I



# Material de lectura

## Contenido

INTRODUCCIÓN	2
INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS (POO)	3
CARACTERÍSTICAS PRINCIPALES DE LA POO	3
Clases y Objetos:	3
REFERENCIAS BIBLIOGRÁFICAS	20

## Introducción

La Programación Orientada a Objetos (POO) es un paradigma que permite organizar el desarrollo de software mediante la creación y manipulación de objetos que representan entidades del mundo real. Su uso en Python facilita la estructuración del código, promoviendo la reutilización, la escalabilidad y el mantenimiento de los programas.

En esta unidad, se explorarán los conceptos fundamentales de la POO, incluyendo clases y objetos, encapsulamiento, herencia, polimorfismo y abstracción. Además, se presentarán ejemplos prácticos y estrategias para implementar estos principios en Python utilizando herramientas como Visual Studio Code.

El objetivo es proporcionar una base sólida para comprender y aplicar la POO en el desarrollo de software, permitiendo a los estudiantes estructurar sus programas de manera eficiente y modular.

# Introducción a la Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el diseño y desarrollo de software en torno a objetos, que son representaciones de entidades del mundo real. Este paradigma es ampliamente utilizado debido a su capacidad para manejar sistemas complejos, promover la reutilización del código y facilitar el mantenimiento de los programas.

## Características principales de la POO

### Clases y Objetos:

1. Clase: Es una plantilla que define las propiedades (atributos) y comportamientos (métodos) comunes de un conjunto de objetos.
2. Objeto: Es una instancia concreta de una clase que puede tener valores específicos para sus atributos.
3. Encapsulamiento: Agrupa atributos y métodos en una clase, restringiendo el acceso directo a ciertos datos para proteger la integridad de los mismos.

Herencia: Permite que una clase (clase hija) reutilice atributos y métodos de otra clase (clase padre).

Polimorfismo: Habilidad de diferentes clases para responder a los mismos métodos de formas específicas.

Abstracción: Permite definir los atributos y métodos esenciales para una entidad, ignorando los detalles no relevantes.

#### Ventajas de la POO

Reutilización del código: Las clases y métodos pueden reutilizarse en otros proyectos o partes del programa.

Modularidad: Divide el programa en partes manejables, facilitando el mantenimiento y la colaboración en equipos.

Facilidad de mantenimiento: Permite realizar cambios en una clase sin afectar otras partes del sistema, siempre que se mantenga la interfaz de la clase.

Escalabilidad: Facilita la adaptación y ampliación del sistema para incorporar nuevas funcionalidades.

## Implementación de la POO en Python

Python es un lenguaje que soporta la POO de forma nativa y es ideal para enseñar y aprender este paradigma debido a su simplicidad. A continuación, se presenta un ejemplo básico:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

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

# Crear una instancia de la clase
persona1 = Persona("Juan", 30)
persona1.saludar()
```

```
class Persona:
```

- Define una clase llamada `Persona`.

Una clase es una plantilla para crear objetos. En este caso, `Persona` es el modelo que define cómo serán los objetos de tipo persona.

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

- `__init__`: Es el **constructor** de la clase, un método especial que se ejecuta automáticamente al crear un objeto de la clase.
- **Parámetros:**
  - `self`: Se refiere al propio objeto que se está creando. Es obligatorio como primer parámetro en todos los métodos de una clase.
  - `nombre`, `edad`: Son parámetros que se pasan al constructor para inicializar los atributos del objeto.
- **Atributos:**
  - `self.nombre`: Es un atributo del objeto que almacenará el nombre de la persona.
  - `self.edad`: Es otro atributo del objeto para almacenar la edad.

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

- Define un método llamado `saludar` :
  - `self` : Permite acceder a los atributos y métodos del objeto actual.
  - `print()` : Imprime un mensaje con el nombre y la edad de la persona utilizando una cadena formateada ( `f-string` ).

```
# Crear una instancia de la clase  
persona1 = Persona("Juan", 30)
```

- Se crea un **objeto (instancia)** de la clase `Persona` llamado `persona1` .
- `"Juan"` y `30` se pasan como argumentos al constructor `__init__` para inicializar los atributos `nombre` y `edad` de la instancia.

```
persona1.saludar()
```

- Se llama al método `saludar()` del objeto `persona1` .
- El método accede a los atributos `nombre` y `edad` de `persona1` y muestra:  
`"Hola, mi nombre es Juan y tengo 30 años."`



## Ejemplo paso a paso en acción

### 1. Definición de la clase `Persona`:

- Describe cómo serán los objetos de tipo persona, con atributos `nombre` y `edad`, y el comportamiento `saludar`.

### 2. Creación del objeto `persona1`:

- El constructor `__init__` inicializa los atributos:
  - `persona1.nombre = "Juan"`
  - `persona1.edad = 30`

### 3. Llamada al método `saludar()`:

- El método `saludar` imprime un mensaje utilizando los valores de los atributos `nombre` y `edad` de `persona1`.

## Salida del programa

Al ejecutar el código, se imprime:

CSS

Hola, mi nombre es Juan y tengo 30 años.

## Ejemplo de Modularidad: Gestión de Hospital

```
class Paciente:
```

- Define una clase `Paciente` para representar a los pacientes en el sistema.

python

```
def __init__(self, nombre, edad, historial):  
    self.nombre = nombre  
    self.edad = edad  
    self.historial = historial
```

- Método especial `__init__` (constructor) que se ejecuta al crear una instancia.
- `self` se refiere al objeto actual.
- `nombre`, `edad` y `historial` son atributos de cada objeto `Paciente`.

python

```
def mostrar_datos(self):  
    return f"Paciente: {self.nombre}, Edad: {self.edad}"
```

- Método `mostrar_datos` devuelve una cadena con información del paciente.

- Método `mostrar_datos` devuelve una cadena con información del paciente.

python

```
class Doctor:
    def __init__(self, nombre, especialidad):
        self.nombre = nombre
        self.especialidad = especialidad
```

- Define una clase `Doctor` con atributos `nombre` y `especialidad`.

python

```
def mostrar_datos(self):
    return f"Doctor: {self.nombre}, Especialidad: {self.especialidad}"
```

- Método `mostrar_datos` devuelve información del doctor.

python

```
class CitaMedica:
    def __init__(self, paciente, doctor, fecha):
        self.paciente = paciente
        self.doctor = doctor
        self.fecha = fecha
```

- Clase `CitaMedica` representa una cita, asociando un paciente, un doctor y una fecha.

```
def detalles_cita(self):  
    return f"Cita: {self.fecha} - Paciente: {self.paciente.nombre}, Doctor:  
{self.doctor.nombre}"
```

- Método `detalles_cita` devuelve una descripción de la cita usando información de los objetos `Paciente` y `Doctor`.

## 2. Ejemplo de Reutilización: Sistema de Transporte

python

```
class Vehiculo:
```

- Define la clase base `Vehiculo` para representar vehículos genéricos.

python

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

- Constructor que inicializa los atributos `marca` y `modelo`.

python

```
def descripcion(self):  
    return f"Vehículo: {self.marca} {self.modelo}"
```

- Método `descripcion` devuelve información básica del vehículo.

python

```
class Auto(Vehiculo):  
    def __init__(self, marca, modelo, puertas):  
        super().__init__(marca, modelo)  
        self.puertas = puertas
```

- Clase `Auto` hereda de `Vehiculo` y agrega el atributo `puertas`.
- `super().__init__()` llama al constructor de la clase base para inicializar `marca` y `modelo`.

```
def descripcion(self):  
    return f"Auto: {self.marca} {self.modelo}, Puertas: {self.puertas}"
```

- Método `descripcion` sobrescribe el de la clase base para incluir `puertas`.

python

```
class Bicicleta(Vehiculo):
    def __init__(self, marca, modelo, tipo):
        super().__init__(marca, modelo)
        self.tipo = tipo
```

- Clase `Bicicleta` hereda de `Vehiculo` y agrega el atributo `tipo`.

python

```
def descripcion(self):
    return f"Bicicleta: {self.marca} {self.modelo}, Tipo: {self.tipo}"
```

- Método `descripcion` sobrescribe el de la clase base para incluir `tipo`.

python

```
auto = Auto("Toyota", "Corolla", 4)
bicicleta = Bicicleta("Trek", "Domane", "Carretera")
```

- Crea objetos `auto` y `bicicleta` con sus respectivas clases.

```
print(auto.descripcion())  
print(bicicleta.descripcion())
```

- Imprime las descripciones específicas para cada objeto.

### 3. Ejemplo de Escalabilidad: Tienda Online

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

- Clase `Producto` con atributos `nombre` y `precio`.

```
python  
  
def mostrar_info(self):  
    return f"Producto: {self.nombre}, Precio: ${self.precio}"
```



- Método `mostrar_info` devuelve información básica del producto.

python

```
class Electronico(Producto):  
    def __init__(self, nombre, precio, garantia):  
        super().__init__(nombre, precio)  
        self.garantia = garantia
```

- Clase `Electronico` hereda de `Producto` y agrega el atributo `garantia`.

```
def mostrar_info(self):  
    return f"Electrónico: {self.nombre}, Precio: ${self.precio}, Garantía:  
{self.garantia} años"
```

- Método sobrescrito para incluir la garantía.

python

```
class Alimento(Producto):  
    def __init__(self, nombre, precio, fecha_caducidad):  
        super().__init__(nombre, precio)  
        self.fecha_caducidad = fecha_caducidad
```

- Clase `Alimento` hereda de `Producto` y agrega el atributo `fecha_caducidad`.

```
def mostrar_info(self):  
    return f"Alimento: {self.nombre}, Precio: ${self.precio}, Caduca: {self.fecha_caducidad}"
```

- Método sobrescrito para incluir la fecha de caducidad.

```
python  
  
televisor = Electronico("Televisor", 500, 2)  
manzana = Alimento("Manzana", 1, "2025-12-31")
```

- Crea objetos `televisor` y `manzana` usando las clases derivadas.

```
python  
  
print(televisor.mostrar_info())  
print(manzana.mostrar_info())
```

- Imprime la información detallada de cada objeto, mostrando cómo las clases hijas amplían las funcionalidades de la clase base.

### Implementación en Python usando Visual Studio Code

Python es un lenguaje que soporta la programación orientada a objetos. Con herramientas como **Visual Studio Code**, puedes escribir, depurar y ejecutar programas orientados a objetos de manera eficiente.



**Ejemplo básico de clase y objeto:**



## Referencias Bibliográficas

Aquí tienes las referencias bibliográficas en formato APA para la información presentada:

- 
- Ceder, N. (2018). *The Quick Python Book* (3rd ed.). Manning Publications.  
Una guía práctica para aprender Python, con un enfoque claro en la programación orientada a objetos.
  - Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.

Proporciona una introducción detallada a Python, incluyendo los conceptos fundamentales de la programación orientada a objetos.

- Griffiths, D., & Griffiths, P. (2020). *Head First Python* (2nd ed.). O'Reilly Media.  
Una guía visual para aprender Python, que incluye ejemplos prácticos y explicaciones sobre el paradigma de programación orientado a objetos.
- Guttag, J. V. (2016). *Introduction to Computation and Programming Using Python* (2nd ed.). MIT Press.  
Cubre los fundamentos de la programación con Python y una sólida introducción a la programación orientada a objetos.
- Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.  
Un recurso integral para aprender Python, desde los fundamentos básicos hasta temas avanzados como la programación orientada a objetos.
- Python Software Foundation. (2023). *The Python Tutorial*. Recuperado de <https://docs.python.org/3/tutorial/>  
Documentación oficial de Python, que incluye una sección sobre clases y objetos con ejemplos prácticos.
- Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.  
Aunque centrado en Java, este libro es ideal para entender los principios de la programación orientada a objetos, aplicables a otros lenguajes como Python.