



Universidad  
Tecnológica  
del Perú

**FACULTAD DE  
INGENIERÍA**

**Domina  
Python y  
Conquista el  
Mundo de los  
Datos.**



## SESIÓN 03

# PYTHON Y EL MUNDO DE LOS DATOS

## Herencia y encapsulamiento en Python



### Logros de Aprendizaje

#### General

Desarrollar competencias en programación con Python, desde fundamentos básicos hasta el manejo avanzado de datos con NumPy y Pandas. Los participantes aprenderán a implementar estructuras, funciones, conceptos de POO (herencia y encapsulamiento) y librerías de análisis, integrando estos conocimientos en un caso práctico para resolver problemas reales y presentar resultados de manera efectiva.

#### Sesión

Al finalizar la sesión, el estudiante desarrolla programas básicos con Python usando herencia y encapsulamiento, así como importación y creación de módulos para organizar el código.

# PYTHON Y EL MUNDO DE LOS DATOS



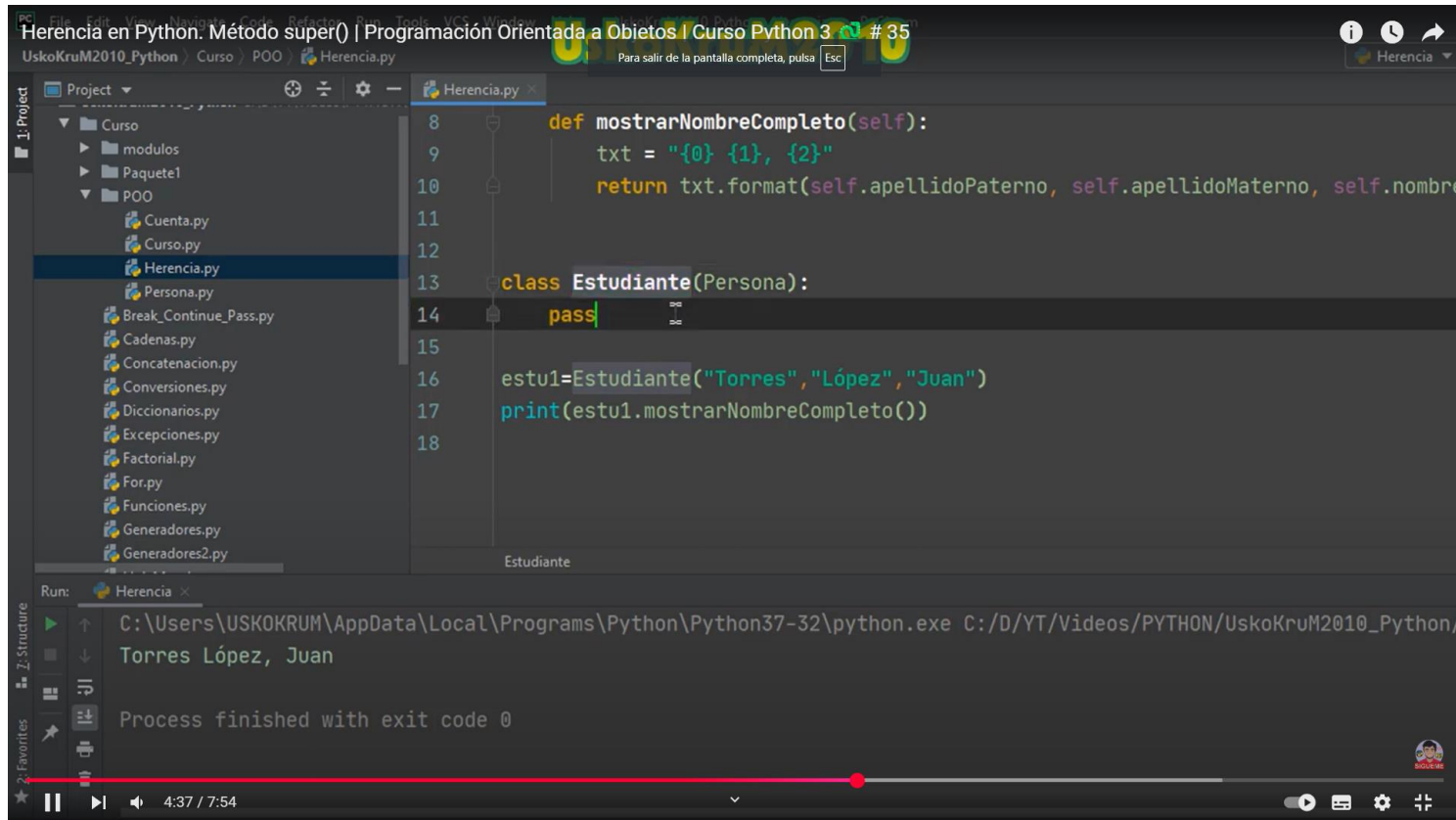
## SESIÓN 03

### Herencia y encapsulamiento en Python

- Herencia y encapsulamiento en POO.
- Importación y creación de módulos para organizar el código.

# INICIO - Conocimientos Previos

## ¿Cómo se implementa la herencia en Python?



The screenshot shows a Python IDE with a project named 'UskoKruM2010\_Python'. The left sidebar displays a file explorer with a directory structure: 'Curso' (containing 'modulos' and 'Paquete1'), and 'POO' (containing 'Cuenta.py', 'Curso.py', 'Herencia.py', 'Persona.py', 'Break\_Continue\_Pass.py', 'Cadenas.py', 'Concatenacion.py', 'Conversiones.py', 'Diccionarios.py', 'Excepciones.py', 'Factorial.py', 'For.py', 'Funciones.py', 'Generadores.py', and 'Generadores2.py'). The 'Herencia.py' file is selected. The main editor shows the following code:

```
8 def mostrarNombreCompleto(self):
9     txt = "{0} {1}, {2}"
10    return txt.format(self.apellidoPaterno, self.apellidoMaterno, self.nombre)
11
12
13 class Estudiante(Persona):
14     pass
15
16 estu1=Estudiante("Torres", "López", "Juan")
17 print(estu1.mostrarNombreCompleto())
18
```

The bottom panel shows the output of the program: 'Torres López, Juan'. The status bar indicates 'Process finished with exit code 0'.



<https://www.youtube.com/watch?v=0xt806gLghg>

# UTILIDAD - Sesión 03

¿Cuál es la utilidad de la herencia y el encapsulamiento en Python?

La utilidad de la herencia y el encapsulamiento en Python reside en su capacidad para promover el diseño estructurado, la reutilización de código, modular, escalabilidad y la seguridad en la programación orientada a objetos (POO).

```
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.")

class Estudiante(Persona):
    def __init__(self, nombre, edad, matricula):
        super().__init__(nombre, edad)
        self.matricula = matricula

    def mostrar_matricula(self):
        print(f"Mi matrícula es {self.matricula}")

estudiante = Estudiante("Carlos", 30, "A12345")
estudiante.saludar()
estudiante.mostrar_matricula()
```

## Herencia y encapsulamiento en POO.

- La **herencia** es un concepto de la POO mediante el cual se puede crear una clase **hija** (derivada o subclase) que hereda de una clase **padre** (base o superclase), compartiendo sus métodos y atributos.
- Además de ello, una clase hija puede sobrescribir los métodos o atributos, o incluso definir unos nuevos.
- Esto facilita la reutilización de código, la extensión de funcionalidades y la creación de jerarquías lógicas entre clases relacionadas.



# Características de la Herencia en Python.

1. **Reutilización de Código:** Una clase hija hereda los métodos y atributos de la clase padre, evitando duplicar código.
2. **Extensibilidad:** La clase hija puede agregar nuevos atributos y métodos o modificar (sobrescribir) los heredados de la clase padre.
3. **Relación Jerárquica:** Se establece una relación "es-un" (is-a) entre la clase hija y la clase padre. Ejemplo: Un "Perro" es un "Animal".
4. **Uso del Constructor de la Clase Padre:** La clase hija puede usar el constructor de la clase padre mediante el método `super()`.

# Características de la Herencia en Python.

5. **Polimorfismo:** Las clases hijas pueden sobrescribir métodos de la clase padre, permitiendo comportamientos diferentes con la misma interfaz.
6. **Herencia Múltiple:** Python admite que una clase hija herede de múltiples clases padres.
7. **Jerarquías Complejas:** Es posible crear cadenas de herencia donde una clase hija actúa como clase padre para otra clase.



# Ejemplos de Herencia en Python.

## Ejemplo 01: Herencia Simple.

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

    def hacer_sonido(self):
        print("Este animal hace un sonido.")

class Perro(Animal):
    def hacer_sonido(self):
        print(f"{self.nombre} dice: ¡Guau!")

mi_perro = Perro("Max")
mi_perro.hacer_sonido()
```

← Clase Padre

← Clase Hija

← Sobrescritura del método

← Uso de las clases

Salida: Max dice: ¡Guau!

# Ejemplos de Herencia en Python.

**Ejemplo 02:** Uso de **super()** para acceder al constructor de la clase padre.

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

    def mostrar_info(self):
        print(f"Nombre: {self.nombre}, Edad: {self.edad}")

class Estudiante(Persona):
    def __init__(self, nombre, edad, matricula):
        super().__init__(nombre, edad)
        self.matricula = matricula

    def mostrar_info(self):
        super().mostrar_info()
        print(f"Matrícula: {self.matricula}")

estudiante = Estudiante("Ana", 20, "A00123")
estudiante.mostrar_info()
```

Accede al constructor de Persona

sobrescritura

Invoca al método de la clase padre

Salida: Nombre: Ana, Edad: 20  
Matrícula: A00123

# Ejemplos de Herencia en Python.

## Ejemplo 03: Herencia Múltiple.

```
class Vehiculo:
    def __init__(self, marca):
        self.marca = marca

class Motor:
    def __init__(self, tipo_motor):
        self.tipo_motor = tipo_motor

class Automovil(Vehiculo, Motor):
    def __init__(self, marca, tipo_motor, modelo):
        Vehiculo.__init__(self, marca)
        Motor.__init__(self, tipo_motor)
        self.modelo = modelo

    def mostrar_info(self):
        print(f"Marca: {self.marca}, Motor: {self.tipo_motor}, Modelo: {self.modelo}")

auto = Automovil("Toyota", "Híbrido", "Prius")
auto.mostrar_info()
```

Salida:

Marca: Toyota, Motor: Híbrido, Modelo: Prius

# Ventajas de la Herencia en Python.

---

1. **Reutilización de código:** Reduce la duplicación al permitir que las clases hijas reutilicen la lógica de las clases padres.
2. **Organización y jerarquías claras:** Facilita la organización del código en estructuras jerárquicas.
3. **Extensibilidad:** Permite agregar o modificar funcionalidades sin alterar el código base.

# Importación y creación de módulos para organizar el código.

## Definición

En Python, los módulos son archivos que contienen definiciones y funciones que pueden ser importadas y reutilizadas en otros archivos. Crear y usar módulos es fundamental para organizar el código, reducir la complejidad de programas grandes y fomentar la reutilización.

## Estructura General

Supongamos que queremos organizar un proyecto que realiza operaciones matemáticas básicas. Este se divide en:

1. Un módulo para operaciones matemáticas.
2. Un módulo principal que usa las funciones del módulo matemático.

# Importación y creación de módulos para organizar el código.

## Paso 01: Creación del Módulo.

Crea un archivo llamado *operaciones.py* con las funciones matemáticas.

```
1  # Archivo: operaciones.py
2
3  def sumar(a, b):
4      """Devuelve la suma de dos números."""
5      return a + b
6
7  def restar(a, b):
8      """Devuelve la resta de dos números."""
9      return a - b
10
11 def multiplicar(a, b):
12     """Devuelve el producto de dos números."""
13     return a * b
14
15 def dividir(a, b):
16     """Devuelve la división de dos números. Valida que el divisor no sea 0."""
17     if b == 0:
18         return "Error: División entre cero no permitida."
19     return a / b
20
```

# Importación y creación de módulos para organizar el código.

## Paso 2: Uso del Módulo

Ahora, crea un archivo principal que importe y use las funciones del módulo.

```
1  # Archivo: main.py
2
3  # Importa el módulo operaciones
4  import operaciones
5
6  def main():
7      print("Operaciones Matemáticas:")
8
9      # Solicitar entrada del usuario
10     a = float(input("Ingrese el primer número: "))
11     b = float(input("Ingrese el segundo número: "))
12
13     # Usar las funciones del módulo
14     print(f"Suma: {operaciones.sumar(a, b)}")
15     print(f"Resta: {operaciones.restar(a, b)}")
16     print(f"Multiplicación: {operaciones.multiplicar(a, b)}")
17     print(f"División: {operaciones.dividir(a, b)}")
18
19     # Llama a la función principal
20     if __name__ == "__main__":
21         main()
22
```



# Importación y creación de módulos para organizar el código.

## Ejemplo de ejecución.

1. Ejecuta el archivo main.py

```
>>> python main.py
```

2. Entrada del usuario

```
>>> ingrese el primer número: 10  
>>> ingrese el segundo número: 5
```

3. Salida esperada:

```
>>> Operaciones Matemáticas:  
Suma: 15.0  
Resta: 5.0  
Multiplicación: 50.0  
División: 2.0
```

# Importación y creación de módulos para organizar el código.

## Variaciones en la Importación

**Importar funciones específicas:** En lugar de importar todo el módulo, puedes importar funciones individuales:

```
1  from operaciones import sumar, dividir
2
3  # Uso directo de las funciones
4  print(sumar(10, 5))  # 15
5  print(dividir(10, 2))  # 5
6
```

# Importación y creación de módulos para organizar el código.

## Variaciones en la Importación

**Renombrar el módulo:** Puedes renombrar el módulo para usar un alias más corto:

```
1  import operaciones as op
2
3  print(op.sumar(10, 5))  # 15
4  print(op.restar(10, 5)) # 5
5
```

# Importación y creación de módulos para organizar el código.

---

## Beneficios de Usar Módulos:

1. **Organización del Código:** Divide el programa en partes manejables, lo que mejora la legibilidad.
2. **Reutilización:** Las funciones y clases de un módulo pueden ser reutilizadas en otros proyectos.
3. **Facilidad de Mantenimiento:** Los cambios en un módulo se reflejan automáticamente en todos los programas que lo usan.
4. **Colaboración:** Facilita el trabajo en equipo al permitir que diferentes programadores trabajen en módulos independientes.

# Importación y creación de módulos para organizar el código.

## Extensión: Uso de Paquetes

Si tienes varios módulos relacionados, puedes organizarlos en paquetes. Un paquete es una carpeta que contiene un archivo `__init__.py` y varios módulos.

Estructura del paquete:

```
1  calculadora/  
2      __init__.py  
3      operaciones.py  
4      estadisticas.py  
5
```

Ejemplo de Uso:

```
1  from calculadora.operaciones import sumar  
2  from calculadora.estadisticas import promedio  
3
```

# Importación y creación de módulos para organizar el código.

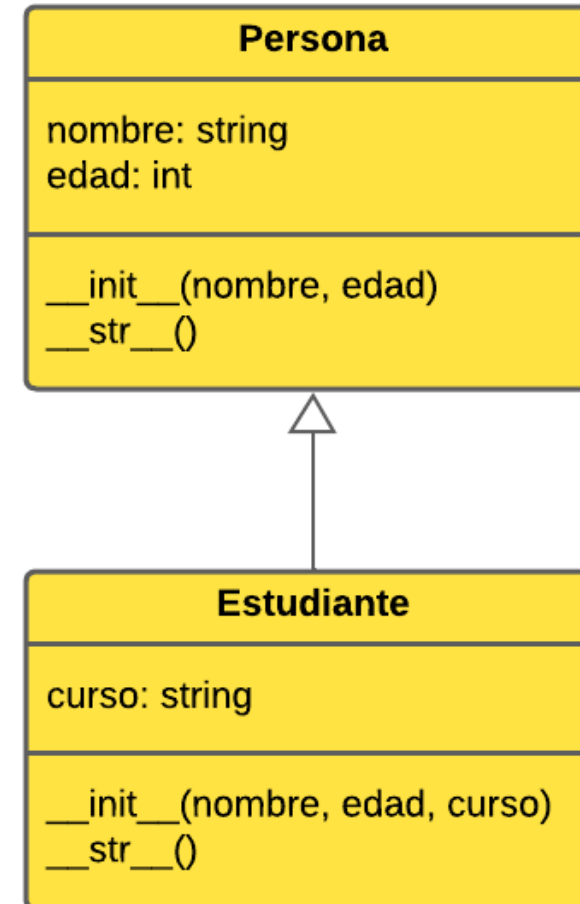
---

## Conclusión

- La creación e importación de módulos en Python es una práctica esencial para mantener un código limpio, reutilizable y escalable.
- A medida que los programas crecen, organizar el código en módulos y paquetes se vuelve indispensable para un desarrollo eficiente.

# PRÁCTICA – Sesión 3

- **Ejercicio 01:** Crear una clase estudiante que herede de Persona y agregue el atributo “curso”.





# PRÁCTICA – Sesión 3

- Solución:

*clase\_persona.py*

```
class Persona:
    nombre = ""
    edad = 0

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

    def __str__(self):
        return f"{self.nombre} ({self.edad})"
```



*clase\_estudiante.py*

```
from clase_persona import Persona

class Estudiante(Persona):
    curso = ''

    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)
        self.curso = curso

    def __str__(self):
        return f"{super().__str__()} - {self.curso}"
```

Importamos el módulo  
donde se encuentra  
definida la clase Persona

```
estudiante1 = Estudiante("Sandra Villar", 22, "Python")
print(estudiante1)
```

Salida: Sandra Villar (22) - Python

# PRÁCTICA – Sesión 3

- **Ejercicio 02:** proteger los atributos “edad” y “nombre” para que solo puedan modificarse a través de los métodos de la clase.

Persona
- nombre: string - edad: int
/* Getters & Setters */ __init__(nombre, edad) __str__()

# PRÁCTICA – Sesión 3

- Solución:


clase\_persona.py

con **@property** definimos  
propiedades para nombre y edad

```
class Persona:
    @property
    def nombre(self): # getter
        return self.__nombre

    @nombre.setter # setter
    def nombre(self, valor):
        if len(valor) > 0:
            self.__nombre = valor
        else:
            raise ValueError("Nombre no válido")

    @property
    def edad(self): # getter
        return self.__edad
```



```
@edad.setter # setter
def edad(self, valor):
    if valor >= 0:
        self.__edad = valor
    else:
        raise ValueError("Edad debe ser positiva")

def __init__(self, nombre, edad):
    self.__nombre = nombre
    self.__edad = edad

def __str__(self):
    return f"{self.__nombre} ({self.__edad})"
```



Usar el doble guion bajo antes del nombre del atributo (\_\_edad) hace que Python realice un *name mangling*, que renombra internamente el atributo (a algo como `_Persona__edad`). Esto hace que sea más difícil acceder al atributo directamente desde fuera de la clase, aunque no lo haga completamente inaccesible.

# PRÁCTICA – Sesión 3

- Solución:

```
persona1 = Persona("Sandra", 44)
print(persona1)
print(persona1.nombre)
print(persona1.edad)

persona1.nombre = 'María Teresa'
persona1.edad = 32
print(persona1)
print(persona1.nombre)
print(persona1.edad)
```

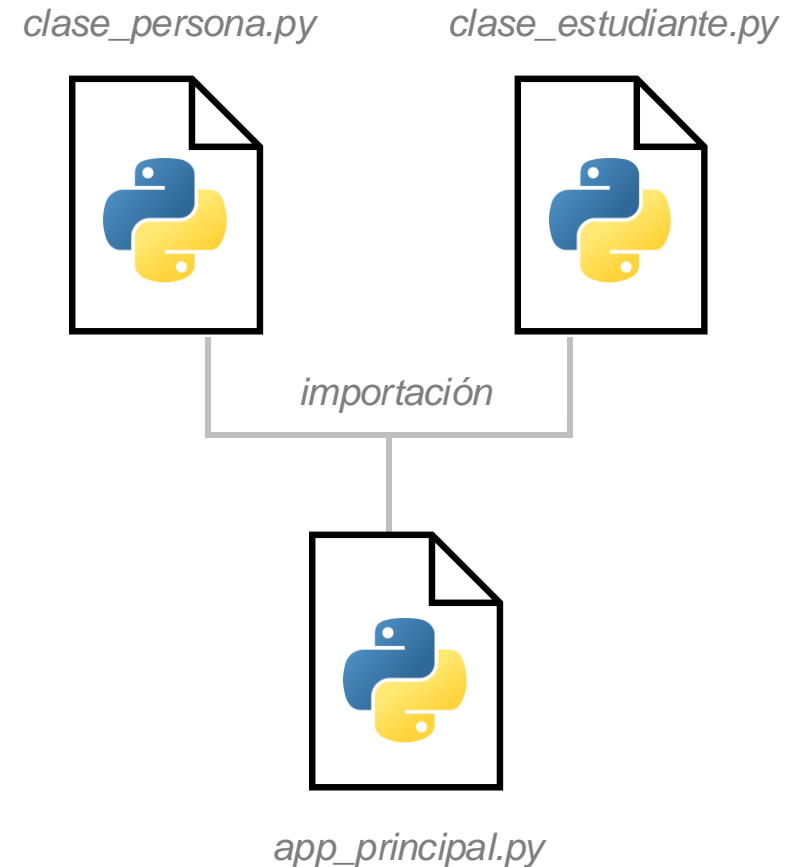


Salida:

```
Sandra (44)
Sandra
44
María Teresa (32)
María Teresa
32
```

# PRÁCTICA – Sesión 3

- **Ejercicio 03:** crear módulos para las clases *Persona* y *Estudiante*, importarlos en un programa principal y probar sus funcionalidades.




# PRÁCTICA – Sesión 3

- Solución:

 practica\_sesión\_03

 clase\_persona.py

 clase\_estudiante.py

 app\_clases.py

```
from clase_persona import Persona
from clase_estudiante import Estudiante
```

```
persona = Persona("Felipe Contreras", 28)
print(persona)
persona.nombre = "Felipe Carrasco"
persona.edad = 22
print(persona)
```

```
estudiante = Estudiante("Marisela Díaz", 32, "HTML5")
print(estudiante)
estudiante.nombre = "Mariela Díaz"
estudiante.edad = 25
estudiante.curso = "SQL Server"
print(estudiante)
```



# ESPACIO PRÁCTICO (Tarea) – Sesión 3

¡Ahora inténtalo tú!

- **Ejercicio propuesto:** Crea la clase *Producto* con los atributos: código, nombre, precio y stock. Agregar el constructor, getters & setters y el método para representar el objeto como cadena. Crea finalmente la clase *GestorProductos* que contenga una lista de objetos *Producto*. Implementa métodos para agregar, modificar, eliminar y mostrar productos.





# ESPACIO PRÁCTICO (Tarea) – Sesión 3

clase\_product.py

```
class Producto(object):  
    def __init__(self, codigo, nombre, precio, stock):  
        self.__codigo = codigo  
        self.__nombre = nombre  
        self.__precio = precio  
        self.__stock = stock
```

```
@property  
def codigo(self):  
    return self.__codigo
```

```
@codigo.setter  
def codigo(self, codigo):  
    self.__codigo = codigo
```

```
@property  
def nombre(self):  
    return self.__nombre
```

```
@nombre.setter  
def nombre(self, nombre):  
    self.__nombre = nombre
```

```
@property  
def precio(self):  
    return self.__precio
```

```
@precio.setter  
def precio(self, precio):  
    if (precio > 0):  
        self.__precio = precio
```

```
@property  
def stock(self):  
    return self.__stock
```

```
@stock.setter  
def stock(self, stock):  
    if (stock > 0):  
        self.__stock = stock
```

```
def __str__(self):  
    return f"{self.__codigo}: {self.__nombre} " + \  
           f"[S/{self.__precio}] ({self.__stock})"
```



# ESPACIO PRÁCTICO (Tarea) – Sesión 3

clase\_gestor\_productos.py

```
from clase_producto import Producto

class GestorProductos(object):
    def __init__(self):
        self.__lista_productos = []

    def agrega_producto(self, nuevo_producto):
        if isinstance(nuevo_producto, Producto):
            self.__lista_productos.append(nuevo_producto)

    def buscar_producto(self, codigo):
        for i in range(len(self.__lista_productos)):
            if self.__lista_productos[i].codigo == codigo:
                return i
        return -1
```



```
    def modificar_producto(self, codigo, nuevo_producto):
        indice = self.buscar_producto(codigo)
        if indice >= 0:
            self.__lista_productos[indice] = nuevo_producto

    def remover_producto(self, codigo):
        indice = self.buscar_producto(codigo)
        if indice >= 0:
            del(self.__lista_productos[indice])

    def mostrar_productos(self):
        for producto in self.__lista_productos:
            print(producto)
```

# ESPACIO PRÁCTICO (Tarea) – Sesión 3

app\_productos.py (*aquí se pone a prueba la implementación*)

```
from clase_producto import Producto
from clase_gestor_productos import GestorProductos

gestor = GestorProductos()

gestor.agrega_producto(Producto('DT254', 'Detergente Potente', 25.0, 120))
gestor.agrega_producto(Producto('AC254', 'Aceite Pureza', 35.50, 87))
gestor.agrega_producto(Producto('MR014', 'Mermelada Dulzura', 18.30, 50))
gestor.agrega_producto(Producto('LE988', 'Leche Mi Vaquita', 6.25, 265))
gestor.agrega_producto(Producto('QS820', 'Queso Fresco', 14.10, 22))

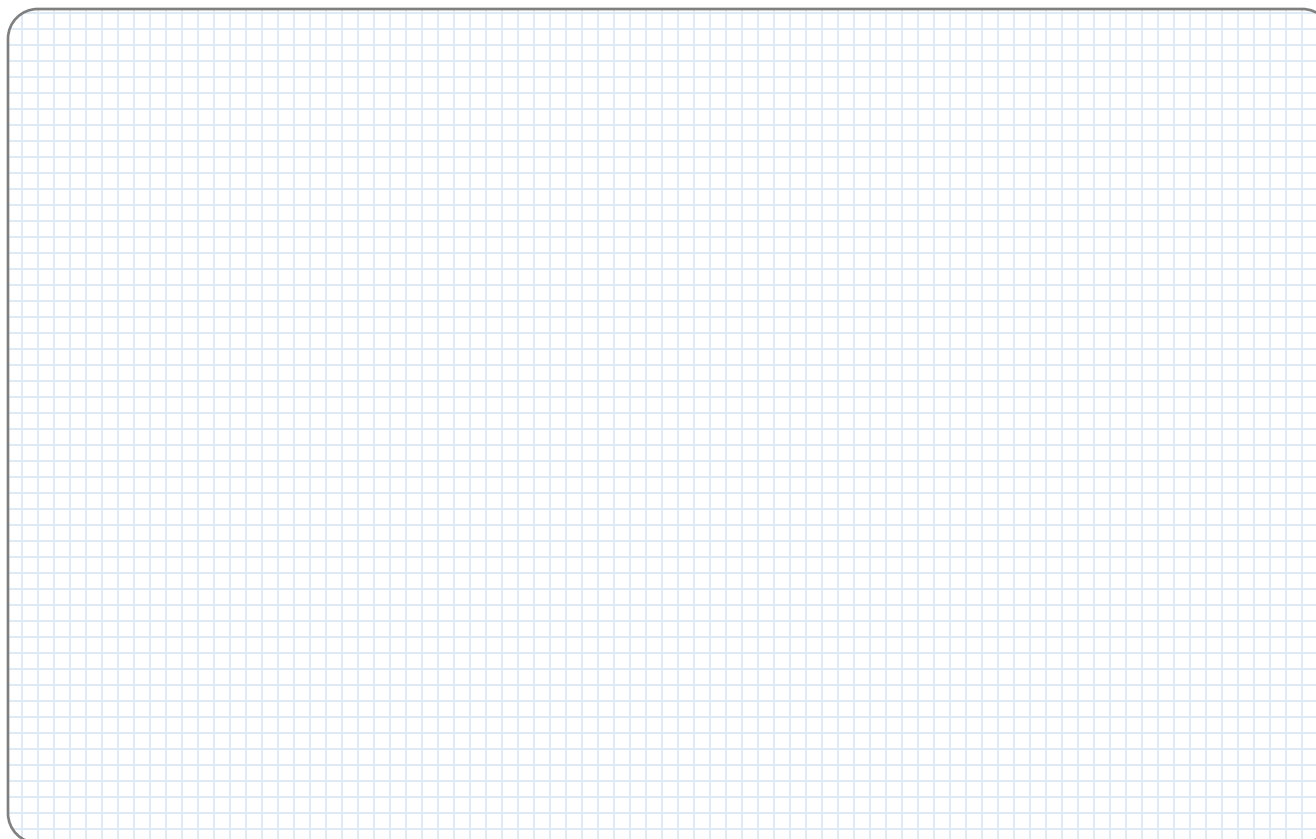
gestor.modificar_producto('MR014', Producto('MR014', 'Mermelada Dulzura', 14.30, 75))
gestor.remover_producto('AC254')

gestor.mostrar_productos()
```



¿Qué es  
herencia en  
POO y cómo  
implementarla  
en Python?

Proporciona un ejemplo de código:

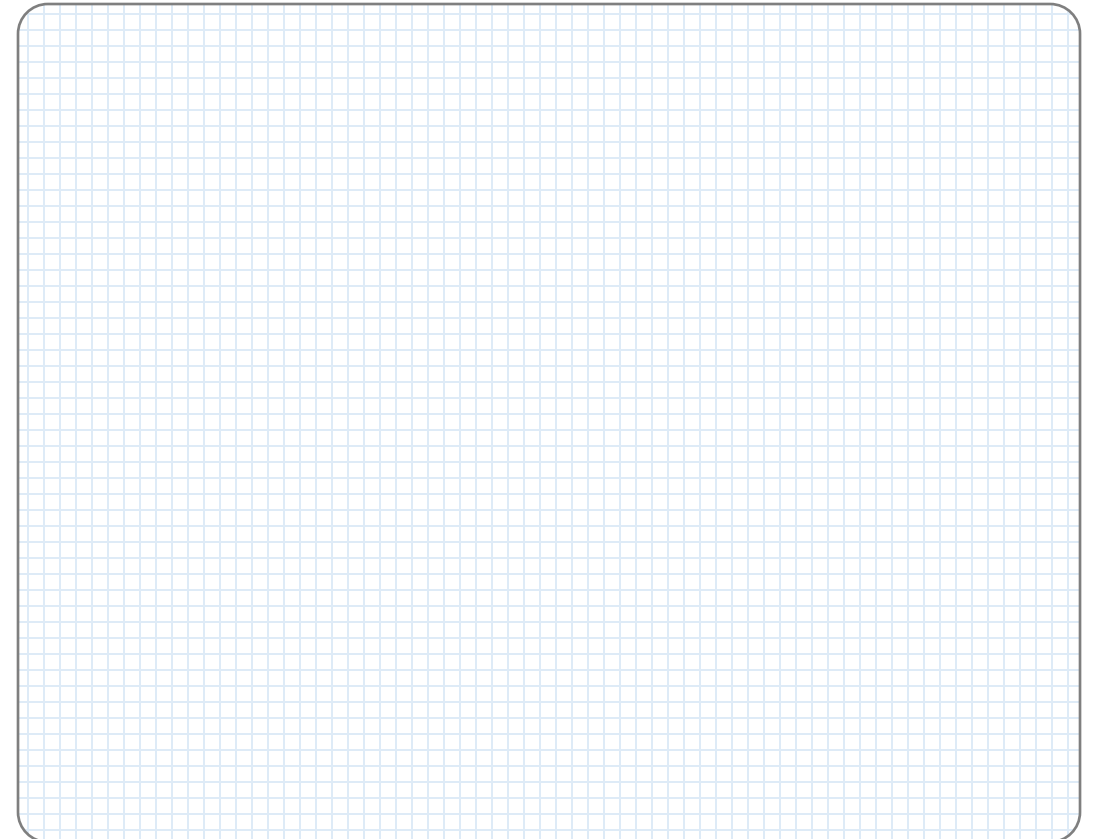


# CIERRE - Sesión 03

¿Qué es un  
módulo en  
Python y cuál  
es su  
utilidad?

Explique  
cómo crear e  
importar  
módulos en  
Python

Proporciona un ejemplo de código:





# Gracias por su atención