**WORKSHOP DE PROGRAMACIÓN**

Alumna: Nancy Salvatierra

Comisión 3

Profesor: Gianluca Piriz

**1. Revisión de Programación Imperativa**

* Pregunta teórica

¿Qué caracteriza a la programación imperativa y en qué se diferencia de la declarativa?

* La **programación imperativa** dice *cómo* se hacen las cosas: paso a paso, usando variables, bucles y condicionales.
* La **declarativa** dice *qué* se quiere lograr, sin especificar cómo lograrlo (por ejemplo, usar SQL o funciones como filter en Python).
* **Verdadero o Falso**

La programación imperativa se basa en describir qué se quiere lograr, no cómo.

❌ FALSO. Eso describe a la programación declarativa. La imperativa se enfoca en los pasos para lograr algo.

* ¿Qué estructuras básicas componen la programación imperativa?
* Asignación (guardar datos en variables)
* Condicionales (if, else)
* Bucles (for, while)
* Secuencias de instrucciones
* **Código con errores para corregir**

nums = [1, 2, 3, 4]

for i in nums:

if i % 2 == 0:

i = i \* 2

print(i)

Este código **modifica i pero no guarda los cambios**. Lo corregimos para que cree una nueva lista con solo los pares duplicados:

nums = [1, 2, 3, 4]

nueva\_lista = []

for i in nums:

if i % 2 == 0:

nueva\_lista.append(i \* 2)

print(nueva\_lista)

# Resultado: (4,8)

**Ejercicio práctico:** Crea una función que reciba una lista de números y devuelva otra lista con solo los números pares multiplicados por 3, usando solo estructuras imperativas (no comprensiones ni funciones como map o filter).

def procesar\_numeros(lista):

resultado = []

for num in lista:

if num % 2 == 0:

resultado.append(num \* 3)

return resultado

# Ejemplo:

print(procesar\_numeros([1, 2, 3, 4, 5, 6]))

# Resultado: [6, 12, 18]

**Reflexión individual:** Diferencia entre código imperativo y usar comprensiones o funciones como map o filter?

* En el imperativo, haces **paso a paso** el proceso.
* Con comprensiones, el código es más **compacto y declarativo**, pero menos claro.
* **Desafío opcional:** Convertí el mismo ejercicio imperativo a uno declarativo (usando list comprehension) y compará claridad y legibilidad.

# Versión declarativa usando list comprehension

def procesar\_numeros(lista):

return [num \* 3 for num in lista if num % 2 == 0]

print(procesar\_numeros([1, 2, 3, 4, 5, 6]))

# Resultado: [6, 12, 18]

**Pilares de la Programación Orientada a Objetos**

* **Pregunta teórica**

¿Cuáles son los 4 pilares de la POO y qué aporta cada uno?

1. **Abstracción**: Oculta los detalles innecesarios y muestra solo lo importante.
2. **Encapsulamiento**: Protege los datos, evitando accesos indebidos.
3. **Herencia**: Permite que una clase herede propiedades y comportamientos de otra.
4. **Polimorfismo**: Permite que objetos distintos respondan al mismo método de diferentes maneras.

* **Mini autoevaluación**

1. ¿Diferencia entre encapsulamiento y abstracción?

* **Encapsulamiento**: Oculta *los datos* y cómo se modifican.
* **Abstracción**: Oculta *los detalles* de implementación y solo muestra lo necesario.

2. ¿Qué pilar permite a las subclases sobrescribir métodos?

**Polimorfismo**.

* **Código con errores para corregir**

Código original:

class Dog:

def \_\_init\_\_(self, name):

name = name

def speak(self):

return "woof"

dog = Dog("Bobby")

print(dog.name)

El error es que no se guarda el nombre en el objeto. Se debe usar self.name.

* **Corrección:**

class Dog:

def \_\_init\_\_(self, name):

self.name = name # así se guarda en el objeto

def speak(self):

return "woof"

dog = Dog("Bobby")

print(dog.name)

# Resultado: Bobby

**Ejercicio práctico:** Define una jerarquía simple para vehículos con al menos una clase base y dos clases hijas. Cada clase hija debe tener un método propio sobrescrito que imprima información diferente. Crea una función que reciba un vehículo y llame a ese método.

Creamos una clase base y dos hijas, cada una con un método propio sobrescrito:

class Vehiculo:

def describir(self):

return "Soy un vehículo"

class Auto(Vehiculo):

def describir(self):

return "Soy un auto con 4 ruedas"

class Moto(Vehiculo):

def describir(self):

return "Soy una moto con 2 ruedas"

# Función que recibe cualquier vehículo

def mostrar\_descripcion(vehiculo):

print(vehiculo.describir())

# Probar

auto = Auto()

moto = Moto()

mostrar\_descripcion(auto) # → Soy un auto con 4 ruedas

mostrar\_descripcion(moto) # → Soy una moto con 2 ruedas

Esto demuestra **herencia** y **polimorfismo**.

* **Reflexión individual** ¿Qué pilar sentís que dominás mejor? ¿Cuál te cuesta más aplicar en la práctica?
* **Herencia** y **encapsulamiento** son más fáciles de entender.
* **Polimorfismo** y **abstracción** cuesten más al principio, porque no se ven tan directo.
* **Desafío opcional:** Agregá encapsulamiento con atributos privados y métodos get y set**.**

class Auto:

def \_\_init\_\_(self, marca):

self.\_\_marca = marca # atributo privado

def get\_marca(self):

return self.\_\_marca

def set\_marca(self, nueva\_marca):

self.\_\_marca = nueva\_marca

mi\_auto = Auto("Toyota")

print(mi\_auto.get\_marca()) # Toyota

mi\_auto.set\_marca("Ford")

print(mi\_auto.get\_marca()) # Ford

Usar \_\_atributo lo hace privado, y get/set son funciones para acceder o cambiar su valor sin modificarlo directamente.

**3. Cohesión y Acoplamiento**

* **Pregunta teórica** ¿Qué significa que una clase tenga alta cohesión y bajo acoplamiento? ¿Por qué es una buena práctica?
* **Alta cohesión**: La clase hace **una sola cosa** y lo hace bien.
* **Bajo acoplamiento**: La clase **no depende demasiado** de otras. Se puede cambiar una sin romper las demás.
* **Es buena práctica** porque hace el código más **mantenible** y fácil de probar.
* **Mini autoevaluación**

1. ¿Qué significa que una clase esté “altamente acoplada”?

Que **depende mucho** de otras clases para funcionar. Cambiar una puede romper otras.

**2. ¿Verdadero o falso? Una clase con alta cohesión tiene muchas responsabilidades distintas.**

❌ **FALSO**. Una clase con alta cohesión **tiene pocas responsabilidades**, bien relacionadas entre sí.

* **Código con errores para corregir**

Código original:

class InvoiceHandler:

def handle\_invoice(self, invoice):

print("Total:", invoice['amount'])

self.save\_to\_db(invoice)

def save\_to\_db(self, invoice):

print("Saving invoice…")

* Ahora cada clase tiene **una sola responsabilidad** → alta cohesión, bajo acoplamiento.
* **Ejercicio práctico** Diseña dos clases: una que calcule el total de una factura (InvoiceCalculator) y otra que solo se encargue de mostrarla (InvoiceDisplay). Usa un objeto invoice como intermediario para mantener bajo acoplamiento.
* InvoiceCalculator: calcula o muestra el total.
* InvoiceDisplay: muestra la factura.
* Ejemplo básico:

class InvoiceCalculator:

def calcular\_total(self, items):

return sum(items)

class InvoiceDisplay:

def mostrar\_factura(self, total):

print(f"El total de la factura es: ${total}")

# Uso

items = [100, 200, 50]

calc = InvoiceCalculator()

total = calc.calcular\_total(items)

display = InvoiceDisplay()

display.mostrar\_factura(total)

Usamos un objeto total como intermediario, sin que una clase dependa directamente de la otra.

* **Reflexión individual**

**¿Cómo podrías detectar un alto acoplamiento en tu propio código? ¿Que impacto tiene?**

* Cuando una clase **usa directamente muchas otras**.
* Cuando **un cambio en una clase rompe muchas otras**.
* Cuando una clase **hace muchas cosas** (baja cohesión).

Impacto: el código se vuelve **difícil de mantener y de probar**.

* **Desafío opcional**

**Convertir tus clases a un sistema basado en interfaces o servicios para reducir acoplamiento:**

Usar clases “interfaz” con métodos que otras clases implementen. Eso ayuda a mantener el bajo acoplamiento.

**4. Herencia Múltiple**

* **Pregunta teórica**

**¿Qué es la herencia múltiple y qué problema puede generar en lenguajes como Python?**

* **Herencia múltiple**: una clase **hereda de dos o más clases al mismo tiempo**.
* Problemas posibles:
  + Si dos clases tienen un método con el mismo nombre, puede haber confusión sobre cuál usar.
  + Python resuelve esto con el MRO (Method Resolution Order), que indica el orden en que busca los métodos.
* **Mini autoevaluación**

**1. ¿Qué es el orden de resolución de métodos (MRO)?**

Es el orden en que Python **busca métodos o atributos** cuando hay herencia múltiple.

**2. ¿Qué conflicto puede surgir si dos clases tienen el mismo método y una clase hereda de ambas?**

Python usará el método de la **clase que aparece primero** en la definición.

* **Código con errores para corregir**

class A:

def greet(self):

print("Hi from A")

class B:

def greet(self):

print("Hi from B")

class C(A, B):

pass

obj = C()

obj.greet()

* Explicar por qué imprime lo que imprime y cómo se puede modificar el orden de herencia.

Imprime: Hi from A  
Porque Python sigue el orden de herencia C(A, B), primero busca en A.

Si cambiás a:

class C(B, A):

pass

Ahora imprimiría: Hi from B.

* **Ejercicio práctico:** Define dos clases con métodos distintos (Walker, Runner) y una tercera clase (Athlete) que herede de ambas. Crea una instancia y demuestra que puede usar métodos de ambas clases. ¿Qué pasa si ambas clases tuvieran un método con el mismo nombre?

Creamos dos clases (Walker y Runner) y una tercera (Athlete) que hereda de ambas.

class Walker:

def walk(self):

print("Caminando...")

class Runner:

def run(self):

print("Corriendo...")

class Athlete(Walker, Runner):

pass

atleta = Athlete()

atleta.walk() # Caminando...

atleta.run() # Corriendo...

#Funciona: el objeto Athlete hereda los métodos de ambas.

**¿Qué pasa si ambas clases tienen un método con el mismo nombre?**

python

CopiarEditar

class Walker:

def move(self):

print("Caminando lentamente")

class Runner:

def move(self):

print("Corriendo rápido")

class Athlete(Walker, Runner):

pass

atleta = Athlete()

atleta.move()

# Resultado: Caminando lentamente, porque Python toma primero la clase Walker.

* **Reflexión individual:** ¿En qué casos usarías herencia múltiple y en cuáles preferirías composición?
* Usária **herencia múltiple** si quiero mezclar funcionalidades simples (como Loggable, Serializable, etc.).
* Preferíria **composición** si las clases son más complejas o independientes (por ejemplo, Auto con un motor).
* **Desafío opcional:** Agregá una tercera clase con un método compartido y analizá qué método se ejecuta usando super().

class Swimmer:

def move(self):

print("Nadando")

class Walker:

def move(self):

super().move()

print("Caminando")

class Athlete(Swimmer, Walker):

def move(self):

super().move()

print("Atleta moviéndose")

a = Athlete()

a.move()

* Resultado:

nginx

CopiarEditar

Nadando

Caminando

Atleta moviéndose

Python sigue el MRO: Athlete → Swimmer → Walker.

**5. Metaclases**

* **Pregunta teórica**

**¿Qué es una metaclase en Python y en qué se diferencia de una clase común?**

* Una **clase** crea **objetos**.
* Una **metaclase** crea **clases**.

En otras palabras:

* Una clase es como un plano para crear objetos.
* Una metaclase es como un plano para crear clases.

Se usan para **modificar cómo se crean o se comportan las clases**.

* **Mini autoevaluación**

1. ¿Qué es una metaclase y cuándo se ejecuta?

* Es una clase **especial** que define cómo se construyen otras clases.
* Se ejecuta cuando se **define una clase**, no cuando se crea una instancia.

**2. ¿Qué diferencia hay entre \_\_new\_\_ y \_\_init\_\_?**

* \_\_new\_\_: crea el objeto (es lo primero que se ejecuta).
* \_\_init\_\_: inicializa el objeto (se ejecuta después de \_\_new\_\_).
* **Código con errores para corregir**

class Meta(type):

def \_\_init\_\_(cls, name, bases, dct):

cls.added = True

class MyClass(metaclass=Meta):

pass

print(MyClass.added)

Corregir para que added se agregue correctamente usando \_\_new\_\_.

Esto **falla** porque \_\_init\_\_ en metaclases no es el lugar adecuado para agregar atributos.

✅ Lo corregimos usando \_\_new\_\_:

class Meta(type):

def \_\_new\_\_(cls, name, bases, dct):

dct['added'] = True

return super().\_\_new\_\_(cls, name, bases, dct)

class MyClass(metaclass=Meta):

pass

print(MyClass.added) # True

* **Ejercicio práctico** Crea una metaclase que agregue automáticamente un método describe() a cualquier clase que la use. El método debe imprimir el nombre de la clase. Crea una clase que la use y prueba que se puede llamar describe() sin haberla definido manualmente.

class AutoDescribe(type):

def \_\_new\_\_(cls, name, bases, dct):

def describe(self):

print(f"Soy una instancia de {self.\_\_class\_\_.\_\_name\_\_}")

dct['describe'] = describe

return super().\_\_new\_\_(cls, name, bases, dct)

class Persona(metaclass=AutoDescribe):

def \_\_init\_\_(self, nombre):

self.nombre = nombre

p = Persona("Ana")

p.describe() # → Soy una instancia de Persona

* **Reflexión individual**

¿Te resultó confuso usar metaclases? ¿Qué ventajas le ves frente a otras técnicas más simples?

* Sí, **son avanzadas**. No se usan todos los días.
* Pero son útiles si querés **modificar muchas clases** de forma automática o controlar su creación.
* **Desafío opcional:** Hacé que la metaclase prohíba crear clases sin un atributo llamado name.

class RequireName(type):

def \_\_new\_\_(cls, name, bases, dct):

if 'name' not in dct:

raise TypeError("La clase debe tener un atributo 'name'")

return super().\_\_new\_\_(cls, name, bases, dct)

# Esta funciona:

class Correcta(metaclass=RequireName):

name = "Clase correcta"

# Esta falla:

# class Incorrecta(metaclass=RequireName):

# pass # Levanta error: "La clase debe tener un atributo 'name'"

**6. Decoradores**

* **Pregunta teórica**

¿Qué es un decorador en Python y para qué se utilisa comúnmente?

Un decorador es una función que recibe otra función como argumento, la modifica o la envuelve, y devuelve una nueva función.

* Se usan para:
* Agregar **funcionalidades extras** a funciones ya existentes.
* Evitar repetir código (ej: chequeos, logs, permisos).
* **Mini autoevaluación**

**1. ¿Cómo se aplica un decorador a una función?**

Con @nombre\_del\_decorador antes de la definición de la función.

@mi\_decorador

def funcion():

pass

**2. ¿Qué función interna suele tener un decorador?**

Una función interna llamada comúnmente wrapper que **envuelve** la original y le agrega algo antes o después de ejecutarla.

* **Código con errores para corregir**

def decorator(func):

print("Decorating...")

return func

@decorator

def greet():

print("Hi!")

greet()

Corregir: Mostrar cómo se aplica realmente un decorador con un wrapper.

Esto imprime "Decorating..." apenas se carga el decorador, no cuando se llama greet().

✅ Versión correcta con wrapper:

def decorator(func):

def wrapper():

print("Decorating...")

func()

return wrapper

@decorator

def greet():

print("Hi!")

greet()

# Resultado:

Decorating...

Hi!

* **Ejercicio práctico** Crea un decorador @authorize que solo permita ejecutar una función si un parámetro user tiene el atributo is\_admin=True. Si no, debe imprimir “Acceso denegado”. Prueba el decorador con una función de ejemplo.

def authorize(func):

def wrapper(user):

if user.get("is\_admin"):

return func(user)

else:

print("Acceso denegado")

return wrapper

@authorize

def ver\_panel\_admin(user):

print("Bienvenido al panel de administrador")

# Prueba

admin = {"nombre": "Ana", "is\_admin": True}

invitado = {"nombre": "Pepe", "is\_admin": False}

ver\_panel\_admin(admin) # Bienvenido al panel de administrador

ver\_panel\_admin(invitado) # Acceso denegado

* **Reflexión individual** ¿Qué otras funciones de Python conocés que usan decoradores? ¿Te gustaría usarlos en tus propios proyectos?
* Decoradores como @staticmethod, @classmethod, @property son comunes en clases.
* También se usan en frameworks web (como Flask: @app.route()).

Son útiles para **limpiar y reutilizar código**, aunque puede costar al principio.

* **Desafío opcional** Creá un decorador que mida el tiempo que tarda en ejecutarse una función.

import time

def medir\_tiempo(func):

def wrapper(\*args, \*\*kwargs):

inicio = time.time()

resultado = func(\*args, \*\*kwargs)

fin = time.time()

print(f"Tiempo de ejecución: {fin - inicio:.4f} segundos")

return resultado

return wrapper

@medir\_tiempo

def tarea\_lenta():

time.sleep(1)

print("Tarea completada")

tarea\_lenta()

✅ Resultado:

Tarea completada

Tiempo de ejecución: 1.000x segundos