Propuestas para Workshop de Programación

1. Revisión de Programación Imperativa

Pregunta teórica:

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

1. ¿Verdadero o falso? La programación imperativa se basa en describir qué se quiere lograr, no cómo.
2. ¿Qué estructuras básicas componen la programación imperativa?

Código con errores para corregir

Código con mal uso del bucle y sin claridad

```
nums = [1,2,3,4]
for i in nums:
    if i % 2 == 0:
        i = i * 2
    print(i)
```

Corregir para que solo los pares se dupliquen y la salida sea una nueva lista.

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).

Reflexión individual
¿Qué diferencias notaste entre escribir código imperativo y usar comprensiones o funciones como `map` o `filter`?

Desafío opcional
Convertí el mismo ejercicio imperativo a uno declarativo (usando list comprehension) y compará claridad y legibilidad.

2. Pilares de la POO

Pregunta teórica:

¿Cuáles son los 4 pilares de la programación orientada a objetos y qué aporta cada uno?

✅Mini autoevaluación

1. ¿Cuál es la diferencia entre encapsulamiento y abstracción?
2. ¿Qué pilar permite a las subclases sobrescribir métodos?

🧪 Código con errores para corregir

```
class Dog:
    def __init__(self, name):
        name = name

    def speak(self):
        return "woof"

dog = Dog("Bobby")
print(dog.name)
```

Corregir: El nombre no se guarda correctamente.

Ejercicio:
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.

🧠 Reflexión individual
¿Qué pilar sentís que dominás mejor? ¿Cuál te cuesta más aplicar en la práctica?

🚀 Desafío opcional

Agregá encapsulamiento con atributos privados y métodos `get` y `set`.

3. Cohesión y Acoplamiento

Pregunta teórica:

¿Qué significa que una clase tenga alta cohesión y bajo acoplamiento? ¿Por qué es una buena práctica?

✅Mini autoevaluación

1. ¿Qué significa que una clase esté “altamente acoplada”?
2. ¿Verdadero o falso? Una clase con alta cohesión tiene muchas responsabilidades distintas.

🧩 Código con errores para corregir

```
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...")
```

Corregir: Separar responsabilidades en dos clases.

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.

🧠 Reflexión individual
¿Cómo podrías detectar un alto acoplamiento en tu propio código? ¿Qué impacto tiene?

🚀 Desafío opcional
Convertí tus clases a un sistema basado en interfaces o servicios para reducir acoplamiento.

4. Herencia Múltiple

Pregunta teórica:

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

✅Mini autoevaluación

1. ¿Qué es el orden de resolución de métodos (MRO)?
2. ¿Qué conflicto puede surgir si dos clases tienen el mismo método y una clase hereda de ambas?

🧩 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.

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?

🧠 Reflexión individual
¿En qué casos usarías herencia múltiple y en cuáles preferirías composición?

🚀 Desafío opcional
Agregá una tercera clase con un método compartido y analizá qué método se ejecuta usando `super()`.

5. Metaclases

Pregunta teórica:

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

✅Mini autoevaluación

1. ¿Qué es una metaclase y cuándo se ejecuta?
2. ¿Qué diferencia hay entre `__new__` y `__init__`?

🐞 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__`.

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.

🧠 Reflexión individual
¿Te resultó confuso el uso de metaclases? ¿Qué ventajas le ves frente a otras técnicas más simples?

🚀 Desafío opcional
Hacé que la metaclase prohíba crear clases sin un atributo llamado `name`.

6. Decoradores

Pregunta teórica:
¿Qué es un decorador en Python y para qué se utiliza comúnmente?

✅Mini autoevaluación

1. ¿Como se aplica a una función?
2. ¿Qué función interna suele tener un decorador?

🧪 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.

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.

🧠 Reflexión individual
¿Qué otras funciones de Python conocés que usan decoradores? ¿Te gustaría usarlos en tus propios proyectos?

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