Jose Revete
19-10040

Tarea 4

1. **Lenguaje escogido: Python**
    a. **Breve descripción**
        i. **Explique la manera de crear y manipular objetos que tiene el lenguaje, incluyendo: constructores, métodos, campos, etc.**

R: Sabiendo que en Python todo es un objeto, incluyendo tipos primitivos como enteros, cadenas y booleanos. Los objetos se crean y manipulan mediante clases y sus instancias.

- Constructores: El método **__init__** es el constructor de una clase. Se ejecuta automáticamente cuando se crea una nueva instancia de la clase.

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

p = Persona("José", 25)  # Instancia creada
```

- Métodos: Son funciones definidas dentro de una clase que operan sobre sus instancias.

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

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

p = Persona("José", 25)
p.saludar()  # Llama al método
```

- Campos (atributos): Representan los datos de un objeto. Pueden ser públicos, protegidos (**_nombre**) o privados (**__nombre**).
- Manipulación dinámica: Python permite agregar atributos y métodos a objetos en tiempo de ejecución:

```
p.apellido = "Revete"
print(p.apellido)  # Agregado dinámicamente
```

- Modificadores de acceso: Python no tiene modificadores de acceso explícitos como public, private o protected, pero sigue:

    - Público: Los atributos/métodos accesibles directamente. Ejemplo: **obj.atributo**.
    - Protegido: Convención mediante un guión bajo **_atributo**. Se usa para indicar que no debería accederse fuera de la clase.

- Privado: Convención mediante dos guiones bajos **__atributo**. Python lo "ofusca" para evitar colisiones de nombres en clases derivadas.

```
class Ejemplo:
    def __init__(self):
        self.publico = "Acceso público"
        self._protegido = "Acceso protegido"
        self.__privado = "Acceso privado"

obj = Ejemplo()
print(obj.publico)     # OK
print(obj._protegido)  # No recomendado
# print(obj.__privado)  # Error
```

- Encapsulación: Aunque Python no implementa una encapsulación estricta, utiliza las convenciones de atributos protegidos y privados ya mencionadas. La encapsulación puede combinarse con propiedades para controlar el acceso a los atributos.

```
class CuentaBancaria:
    def __init__(self, saldo):
        self.__saldo = saldo

    @property
    def saldo(self):  # Getter
        return self.__saldo

    @saldo.setter
    def saldo(self, monto):  # Setter
        if monto >= 0:
            self.__saldo = monto
        else:
            raise ValueError("El saldo no puede ser negativo")

cuenta = CuentaBancaria(100)
print(cuenta.saldo)  # Acceso controlado
cuenta.saldo = 200   # Actualización controlada
```

ii. Describa el funcionamiento del manejo de memoria, ya sea explícito (new/delete) o implícito (recolector de basura).

R: Python usa manejo de memoria implícito a través de un recolector de basura basado en conteo de referencias y un algoritmo de rastreo de ciclos.

- Conteo de referencias: Cada objeto tiene un contador de referencias que incrementa al ser referenciado y decrece al eliminar referencias.

```
a = [1, 2, 3]
b = a  # Incrementa referencia
del b  # Decrementa referencia
```

- Recolector de basura: Libera memoria ocupada por objetos que ya no son accesibles (por ejemplo, en ciclos de referencia). A pesar de que se ejecuta automáticamente, es posible forzarlo con el módulo gc:

```
import gc
gc.collect()  # Fuerza recolección
```

iii. Diga si el lenguaje usa asociación estática o dinámica de métodos y si hay forma de alterar la elección por defecto del lenguaje.

R:

- Asociación dinámica por defecto: Python resuelve los métodos en tiempo de ejecución según el tipo real del objeto. Si una clase derivada sobrescribe un método de la base, se ejecuta la versión de la clase derivada incluso si el objeto se accede como una instancia de la clase base.

```
class Animal:
    def sonido(self):
        print("Genérico")

class Perro(Animal):
    def sonido(self):
        print("Ladrido")

perro = Perro()
perro.sonido()  # Imprime "Ladrido"
```

- Alterar el comportamiento dinámico:
  - super(): Llama explícitamente a métodos de la clase base.

```
class Perro(Animal):
    def sonido(self):
        super().sonido()  # Llama al método base
        print("Ladrido")
```

  - Herencia múltiple y MRO: El orden de resolución de métodos (MRO) define cuál método se ejecuta. Consultable con Clase.__mro__.

- Asociación estática simulada: Python no usa asociación estática, pero los métodos estáticos (@staticmethod) no dependen de instancias ni de la clase.

```
class Calculadora:
    @staticmethod
    def sumar(a, b):
        return a + b
```

iv. Describa la jerarquía de tipos, incluyendo mecanismos de herencia múltiple (de haberlos), polimorfismo paramétrico (de tenerlo) y manejo de varianzas.

R:

- Tipos base: Todos los objetos heredan de object. Python tiene una jerarquía unificada.
- Herencia múltiple: Python soporta herencia múltiple y utiliza el MRO (Method Resolution Order) para resolver conflictos.

```
class A:
    def metodo(self):
        print("A")
class B:
    def metodo(self):
        print("B")
class C(A, B):
    pass
obj = C()
obj.metodo()  # Resuelve según el MRO
```

- Polimorfismo paramétrico: Se implementa mediante tipos genéricos con el módulo typing:

```
from typing import List

def sumar_lista(numeros: List[int]) -> int:
    return sum(numeros)
```

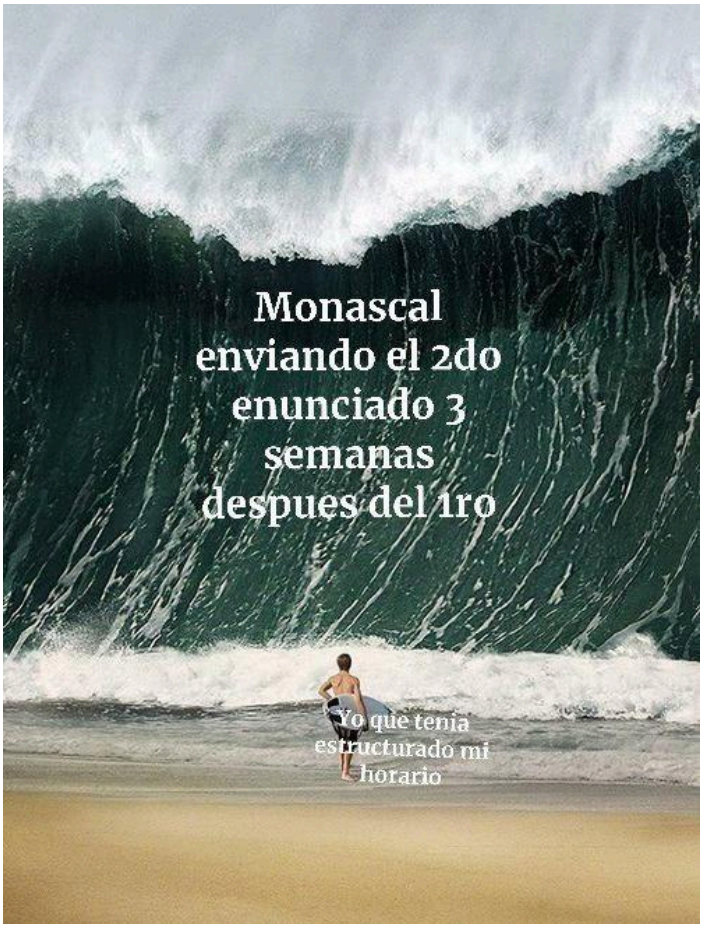
- Varianza: Python admite la covarianza y contravarianza a través del módulo typing.

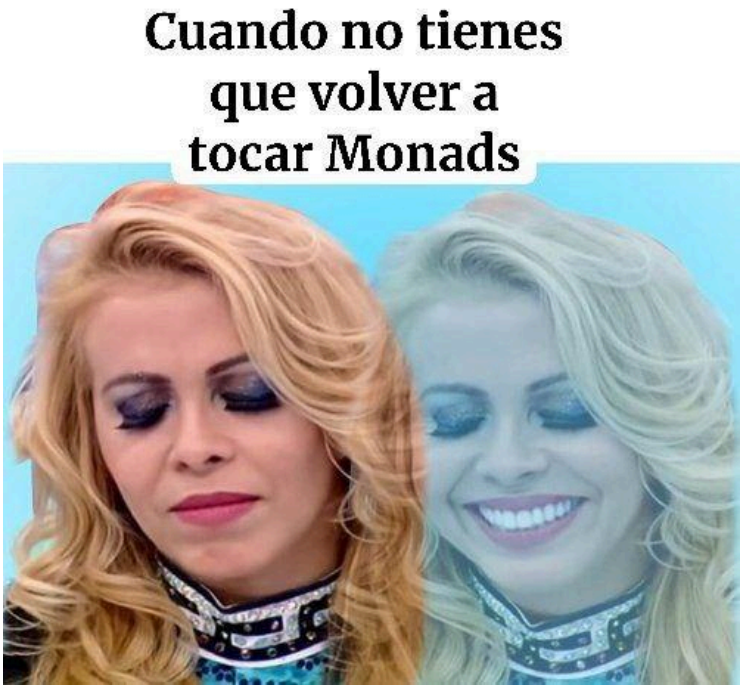
```
from typing import List

def procesar_lista(lista: List[object]):
    print(lista)

procesar_lista([1, 2, 3])  # List[int] es compatible con List[object]
```

EXTRA:


Monascal enviando el 2do enunciado 3 semanas despues del 1ro
Yo que tenia estructurado mi horario


Cuando no tienes que volver a tocar Monads