Programación 2 — Actividad Autónoma 1

Unidad 1: Fundamentos de la Programación Orientada a Objetos (POO)
Tema 1: Introducción a la POO, Clases y Objetos

Nombre: Pablo Andrés Terán Corrales

Objetivo de la actividad

Diseñar e implementar una clase orientada a objetos que modele instancias del dataset de **Iris**, aplicando principios de **encapsulamiento** y representando la solución con un **diagrama UML**.

Recursos previos sugeridos

- Conceptos fundamentales de POO en Python.
- Definición y uso de clases y objetos.
- Atributos privados y encapsulamiento mediante `@property` y `@<prop>.setter`.
- Método mágico `__init__()`.
- Representación con `__repr__()`.
- Estructura básica de un diagrama UML de clases.
- Sintaxis de Python en Jupyter Notebook.

1) Implementación de la clase `Sample`

La clase representa una flor del dataset Iris con los atributos solicitados:

- `sepal_length: float`
- `sepal_width: float`
- `petal_length: float`
- `petal_width: float`
- `species: str` *(encapsulado)*

In [2]:
```
class Sample:
    """Representa una muestra (flor) del dataset Iris."""
    def __init__(self, sepal_length, sepal_width, petal_length, petal_width,
        # Conversión defensiva a float para asegurar tipos numéricos correct
        self.sepal_length = float(sepal_length)
```

```
        self.sepal_width = float(sepal_width)
        self.petal_length = float(petal_length)
        self.petal_width = float(petal_width)
        # Atributo encapsulado (nombre con doble guion bajo para name mangli
        self.__species = species

    # Getter del atributo species
    @property
    def species(self):
        """Obtiene la especie (cadena)."""
        return self.__species

    # Setter del atributo species con validación
    @species.setter
    def species(self, value):
        if not isinstance(value, str):
            raise ValueError("El valor de 'species' debe ser una cadena de t
        if value.strip() == "":
            raise ValueError("'species' no puede ser una cadena vacía.")
        self.__species = value

    def __repr__(self):
        """Devuelve una representación concisa y legible del objeto."""
        return (f"Sample(sepal_length={self.sepal_length}, sepal_width={self
                f"petal_length={self.petal_length}, petal_width={self.petal_
                f"species='{self.__species}')")

    def describe(self):
        """Devuelve una descripción en lenguaje natural de la muestra."""
        return (f"La flor de especie {self.__species} tiene sépalos de "
                f"{self.sepal_length}×{self.sepal_width} cm y pétalos de "
                f"{self.petal_length}×{self.petal_width} cm.")

    def matches(self, species):
        """Retorna True si la especie coincide con el valor dado (comparació
        return self.__species.lower() == str(species).lower()
```

Ejemplos de uso

Las siguientes celdas muestran ejemplos simples de creación, representación, descripción y verificación de coincidencia de especie.

In [3]:
```
# Crear una instancia de Sample
sample = Sample(5.1, 3.5, 1.4, 0.2, "Iris-setosa")
sample
```

Out[3]:
```
Sample(sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_width=0.
2, species='Iris-setosa')
```

In [4]:
```
# Describir la flor
sample.describe()
```

Out[4]:
```
'La flor de especie Iris-setosa tiene sépalos de 5.1×3.5 cm y pétalos de 1.4×0.2 cm.'
```

In [5]:
```python
# Comprobar coincidencia de especie (True esperado)
sample.matches("iris-setosa")
```

Out[5]:
```
True
```

In [6]:
```python
# Actualizar species usando el setter (con validación)
sample.species = "Iris-versicolor"
sample, sample.describe(), sample.matches("Iris-virginica")  # Coincidencia
```

Out[6]:
```
(Sample(sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_width=0.2, species='Iris-versicolor'),
 'La flor de especie Iris-versicolor tiene sépalos de 5.1×3.5 cm y pétalos de 1.4×0.2 cm.',
 False)
```

Pruebas básicas (sanity checks)

Pequeño conjunto de pruebas con `assert` para verificar comportamientos esperados.

In [8]:
```python
# Tipos y representación
s = Sample(6.3, 2.9, 5.6, 1.8, "Iris-virginica")
assert isinstance(s.sepal_length, float)
assert isinstance(s.sepal_width, float)
assert isinstance(s.petal_length, float)
assert isinstance(s.petal_width, float)
assert isinstance(s.species, str)
r = repr(s)
assert r.startswith("Sample(") and "species='Iris-virginica'" in r

# matches (case-insensitive)
assert s.matches("IRIS-VIRGINICA") is True
assert s.matches("Iris-versicolor") is False

# setter validaciones
try:
    s.species = 123  # Debe fallar por tipo
    assert False, "Setter species debía lanzar ValueError por tipo"
except ValueError:
    pass

try:
    s.species = "  "  # Debe fallar por vacío
    assert False, "Setter species debía lanzar ValueError por cadena vacía"
except ValueError:
    pass

"Pruebas básicas OK"
```

Out[8]:
```
'Pruebas básicas OK'
```

2) Uso de `@property` y `@<prop>.setter` para `species`

En la clase anterior, `species` está **encapsulado** mediante `__species` (name-mangling) y expuesto de forma controlada con un *getter* (`@property`) y un *setter* (`@species.setter`) que valida el tipo y prohíbe cadenas vacías.

3) Métodos requeridos incluidos

- `__init__()` — inicializa todos los atributos.
- `__repr__()` — representación concisa y legible.
- `describe()` — redacción comprensible de medidas.
- `matches(species)` — compara especies (insensible a mayúsculas/minúsculas).

4) Diagrama UML de la clase `Sample`

> Representación textual (puedes replicarla en **draw.io / diagrams.net** y exportar como imagen PNG para tu entrega).

```
+----------------------------+
|          Sample            |
+----------------------------+
| + sepal_length: float      |
| + sepal_width: float       |
| + petal_length: float      |
| + petal_width: float       |
| - __species: str           |
+----------------------------+
| + __init__(...)            |
| + species(): str           |
| + species(value): void     |
| + __repr__(): str          |
| + describe(): str          |
| + matches(species): bool   |
+----------------------------+
```

Convenciones UML: `+` público, `-` privado.

5) Preguntas (respuestas breves)

a) ¿Qué significa encapsular un atributo?
Encapsular significa **proteger los datos internos de una clase** para impedir su acceso/modificación directa desde fuera del objeto y permitir un **acceso controlado** mediante propiedades o métodos. En Python se apoya en el *name mangling* (`__atributo`) y en `@property` / `@setter` para validar y mantener la coherencia del estado.

b) ¿Por qué es importante encapsular `species` en lugar de dejarlo público?
Porque `species` representa una clasificación que debe ser **consistente y válida**. Si fuera público, podría recibir valores no deseados (p. ej., números o cadenas vacías). Con encapsulamiento, se centraliza la **validación** y se reduce el riesgo de errores o estados inválidos.

c) ¿Qué ventajas ofrece `@property` frente a acceder directamente al atributo?

1. **Control:** permite validaciones, transformaciones y lógica adicional al leer/escribir.
2. **Compatibilidad futura:** el código cliente no cambia si luego agregas lógica en el *getter/setter*.
3. **Mantenibilidad:** facilita documentar y auditar cambios sobre atributos sensibles.

Link GitHub:
https://github.com/Pabloateran/Actividad_Autonoma1_Programacion2_Teran_Pablo/blob/6c13db4211a63fbedf101a3a1f557e7bc77076b8/Act_Autonoma1_Teran_Pablo.ipynb