BACKEND CON PYTHON

PROGRAMACION ORIENTADA A OBJETOS

- Clases y Objetos
- UML
- Encapsulamiento

PROGRAMACION ORIENTADA A OBJETOS

- Clases y Objetos
- UML
- Encapsulamiento

CLASES Y OBJETOS

Python es un lenguaje de programación orientada a objetos.
Primero debemos comprender que es un objeto:

Los objetos son una abstracción de las entidades del mundo real. Los objetos se definen por medio de sus características y comportamiento relevantes.

- A las características las llamaremos atributos
- Al comportamiento los llamaremos métodos

CLASES Y OBJETOS

Para entender mejor que es un objeto utilicemos el ejemplo de un Perro.

Los perros tienen muchísimas características, pero dentro de la abstracción que estamos haciendo para representar el objeto solo nos vamos a quedar con las siguientes características (atributos):

- nombre: que será un str
- raza: que será un str
- peso (en kilos): que será un float
- edad (en años): que será un int

Dentro de todas las acciones que puede realizar un perro, en la abstracción solo decidimos tener dos comportamientos (métodos):

- ladrar
- modificar peso

CLASES Y OBJETOS

A partir de la definición anterior del objeto Perro, podemos tener varios objetos de tipo Perro (con nombre, raza, peso y edad)

Perro1:
Raza: Rottweiler
Nombre: Zeus
Edad: 3
Peso: 45.8

Perro2:
Raza: Golden R.
Nombre: Nala
Edad: 1
Peso: 8.5

Perro3:
Raza: Alabai
Nombre: Atila
Edad: 5
Peso: 58.9

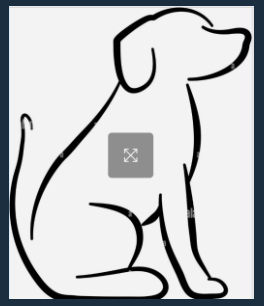
CLASES Y OBJETOS

Ahora que entendemos que un objeto es una abstracción de la realidad debemos comprender que es una Clase.

- Las clases son los planos de un objeto, en el que se definen atributo y métodos,
- Las clases definen atributos para representar las características de la abstracción
- Los métodos son funciones que representan las acciones que podrá realizar el objeto
- La clase es el plano y el objeto es dar valor a esos planos.
- La clase puede ser visto como el molde, o una unidad reutilizable para crear diferentes objetos.

CLASES Y OBJETOS

Clase Perro:
Tiene los atributos nombre, raza, edad y peso.
Tiene los métodos ladrar y modificar_peso



A partir de esa clase se pueden crear objetos dando valores a cada uno de sus atributos:

Perro1: nombre="Zeus", raza="Rottweiler", edad=3, peso=45.8

Perro2: nombre="Nala", raza="Golden R.", edad=0, peso=8.5

Perro2: nombre="Atila", raza= "Alabai", edad=5, peso= 58.9

CLASES Y OBJETOS

En Python para crear una clase, debemos primero crear un archivo .py, idealmente con el nombre de la clase, en este caso perro.py, podría tener otro nombre, pero por buena práctica es mejor ponerle el mismo.

Acá vemos como escribir una clase en Python.

- La palabra self, es un valor que se necesita para hablar de objetos en Python.
- La función __init__, es el constructor, es decir, la función que se encarga de darle valor a cada uno de los atributos.
- Los atributos siguen la forma self.atributo, como por ejemplo: self.raza
- Los métodos necesitan el parámetro self, seguido de los parámetros adicionales que necesite

perro.py*

```python
# -*- coding: utf-8 -*-

class Perro:
    def __init__(self, nombre, raza, peso, edad):
        self.nombre = nombre
        self.raza = raza
        self.peso = peso
        self.edad = edad

    def ladrar(self):
        return "¡Guau, guau!"

    def modificar_peso(self, nuevo_peso):
        self.peso = nuevo_peso
```

CLASES Y OBJETOS

Ya tenemos la clase perro

```
class Perro:
```

Los atributos

```
    self.nombre
    self.raza
    self.peso
    self.edad
```

El constructor

```
def __init__(self, nombre, raza, peso, edad):
    self.nombre = nombre
    self.raza   = raza
    self.peso   = peso
    self.edad   = edad
```

Y los métodos

```
def ladrar(self):
    return "¡Guau, guau!"

def modificar_peso(self, nuevo_peso):
    self.peso = nuevo_peso
```

CLASES Y OBJETOS

Ahora podemos crear los 3 perros que hemos estado usando de ejemplo.
Para esto, hemos creado el módulo guarderia.py que se encargará de manejar los 3 perros. Al estar en un módulo separado, debemos importar la clase Perro como se muestra en la línea 3.

perro.py | guarderia.py*

```
1  # -*- coding: utf-8 -*-
2
3  from perro import Perro
4
5  perro_1 = Perro("Zeus", "Rottweiler", 45.8, 3)
6  perro_2 = Perro("Nala", "Golden R.", 8.5, 0)
7  perro_3 = Perro("Atila", "Alabai", 58.9, 5)
```

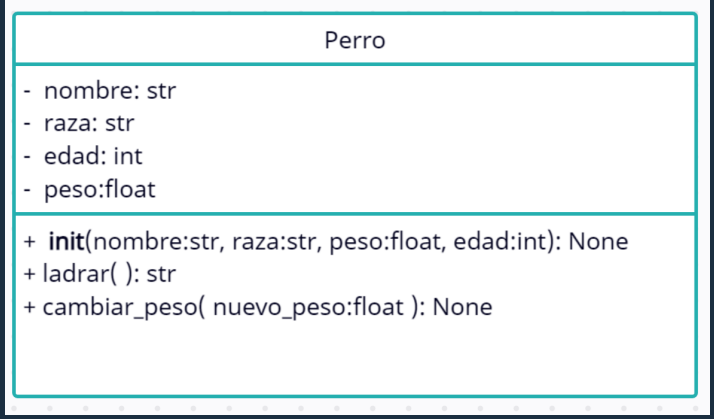
Ahora tenemos 3 objetos de tipo Perro en guarderia.py

PROGRAMACION ORIENTADA A OBJETOS

- Clases y Objetos
- UML
- Encapsulamiento

UML

Para representar tanto clases como objetos (y muchísimos otros elementos) se creó UML.
UML viene de Unified Modeling Language, es decir, lenguaje de modelado unificado.
UML nace por la necesidad de estandarizar la forma de representar gráficamente elementos de programación, redes, procesos, requerimientos, entre muchos otros.
A continuación, veremos la notación UML para diagramas de clases, a través del ejemplo del Perro.

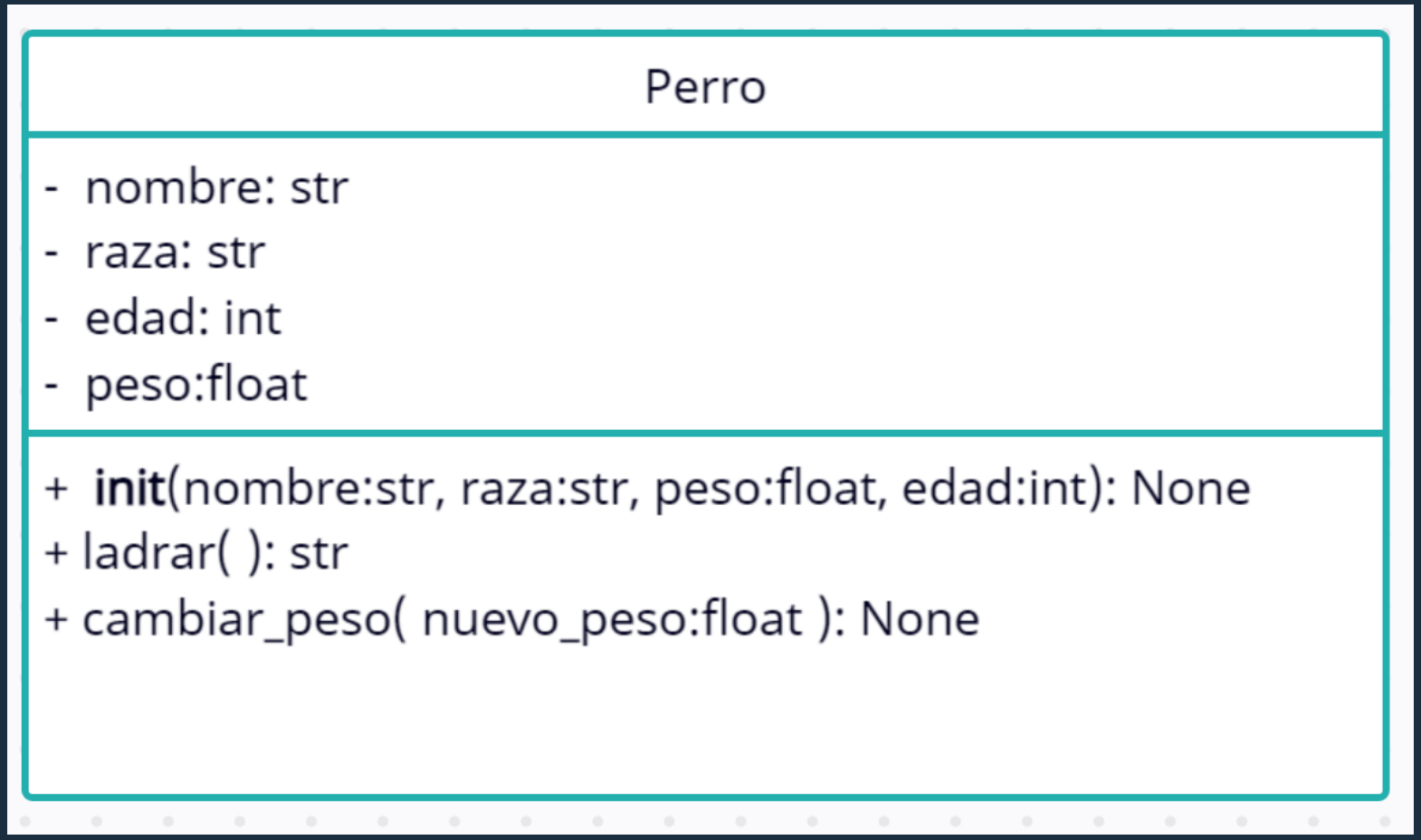


UML

Nombre de la clase->

Atributos –>

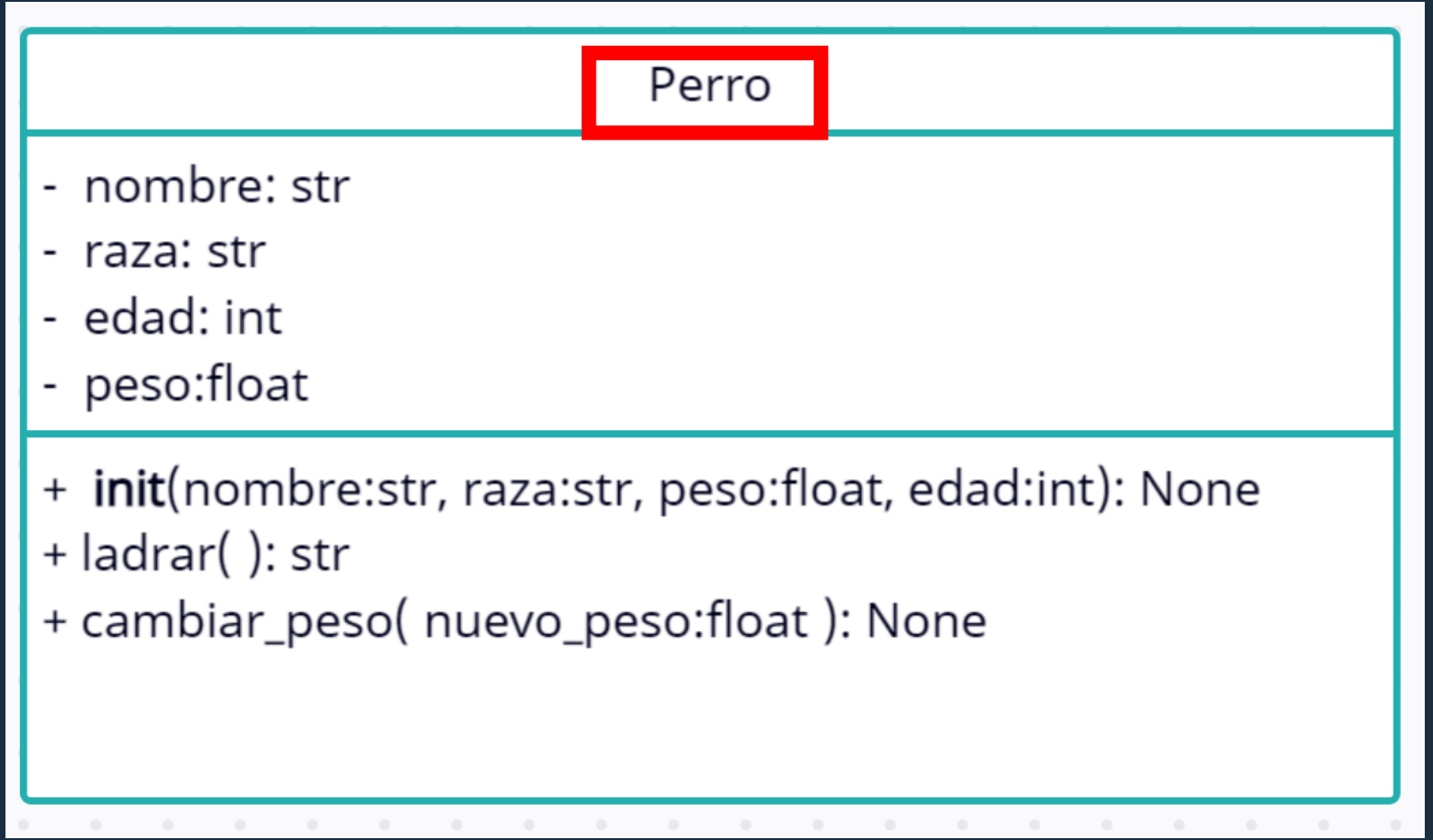
Métodos –>



UML

Nombre de la clase:

Puede ir acompañado de un estereotipo (si es interface o abstract). El ejemplo del Perro no tiene estereotipo pero lo veremos cuando hablemos de herencia.

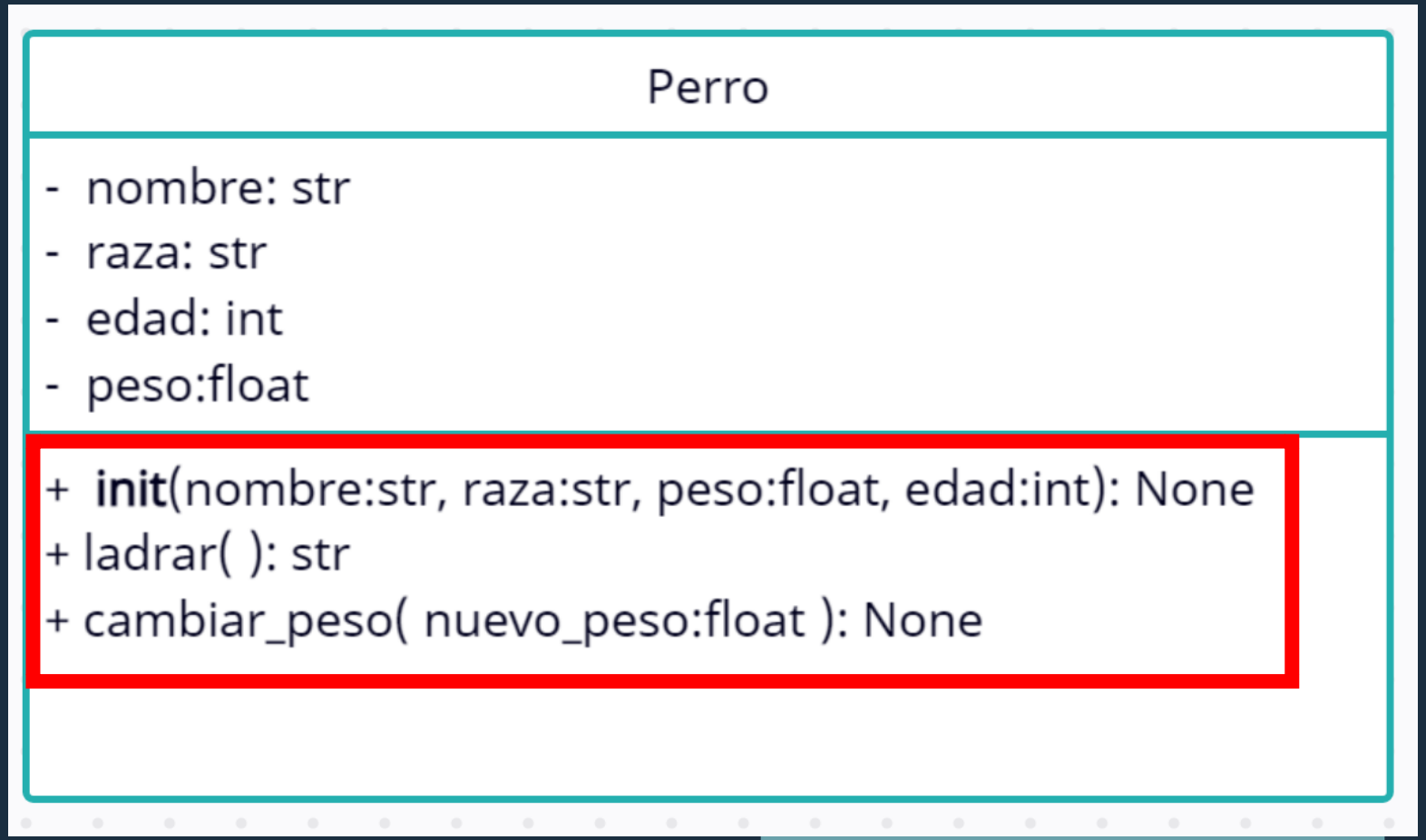


UML

Métodos:

Tienen la forma:
Visibilidad
nombre(parámetros): retorno

El método init se marca en negrilla porque al llevar __, es un método particular: el método constructor



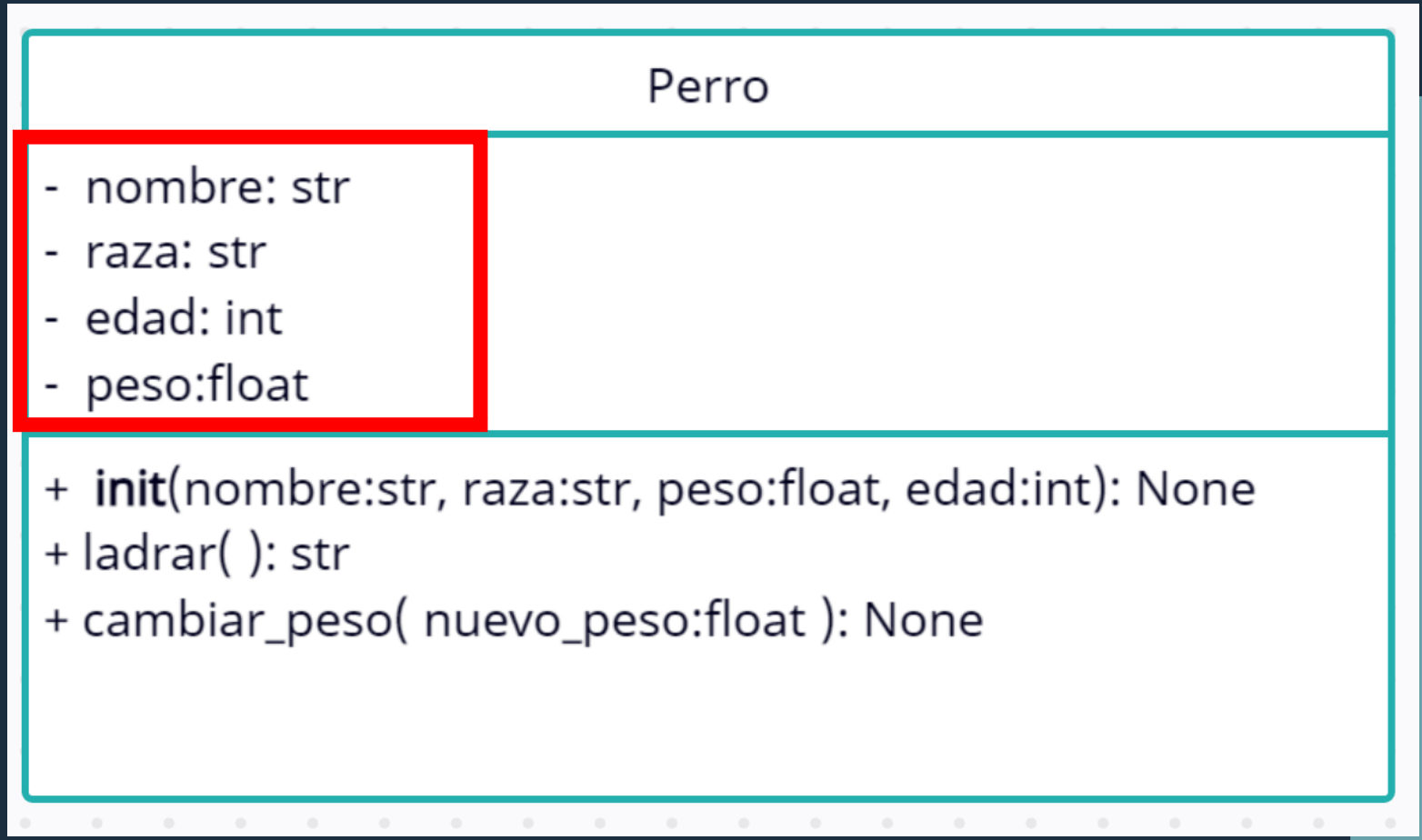
UML

Atributos:

Tienen la forma:
Visibilidad nombre: tipo

Hay 3 visibilidades:
public: +
private: -
protected: #

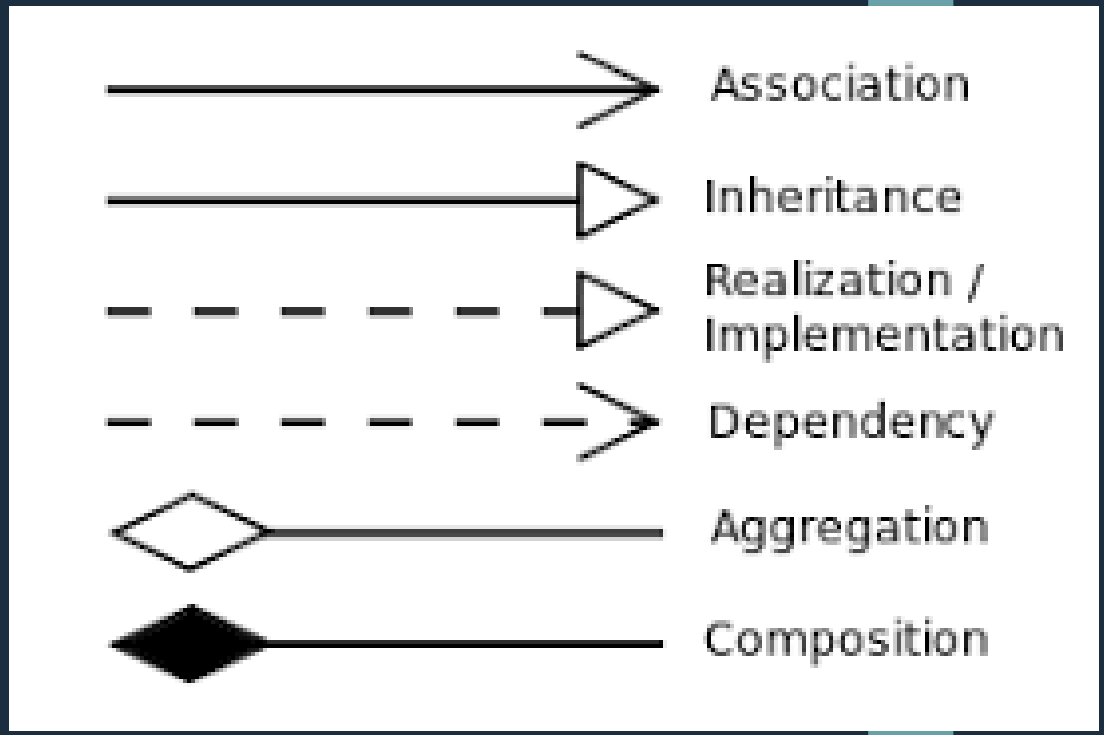
La visibilidad es una herramienta que ayuda al acoplamiento y hablaremos mas adelante



UML

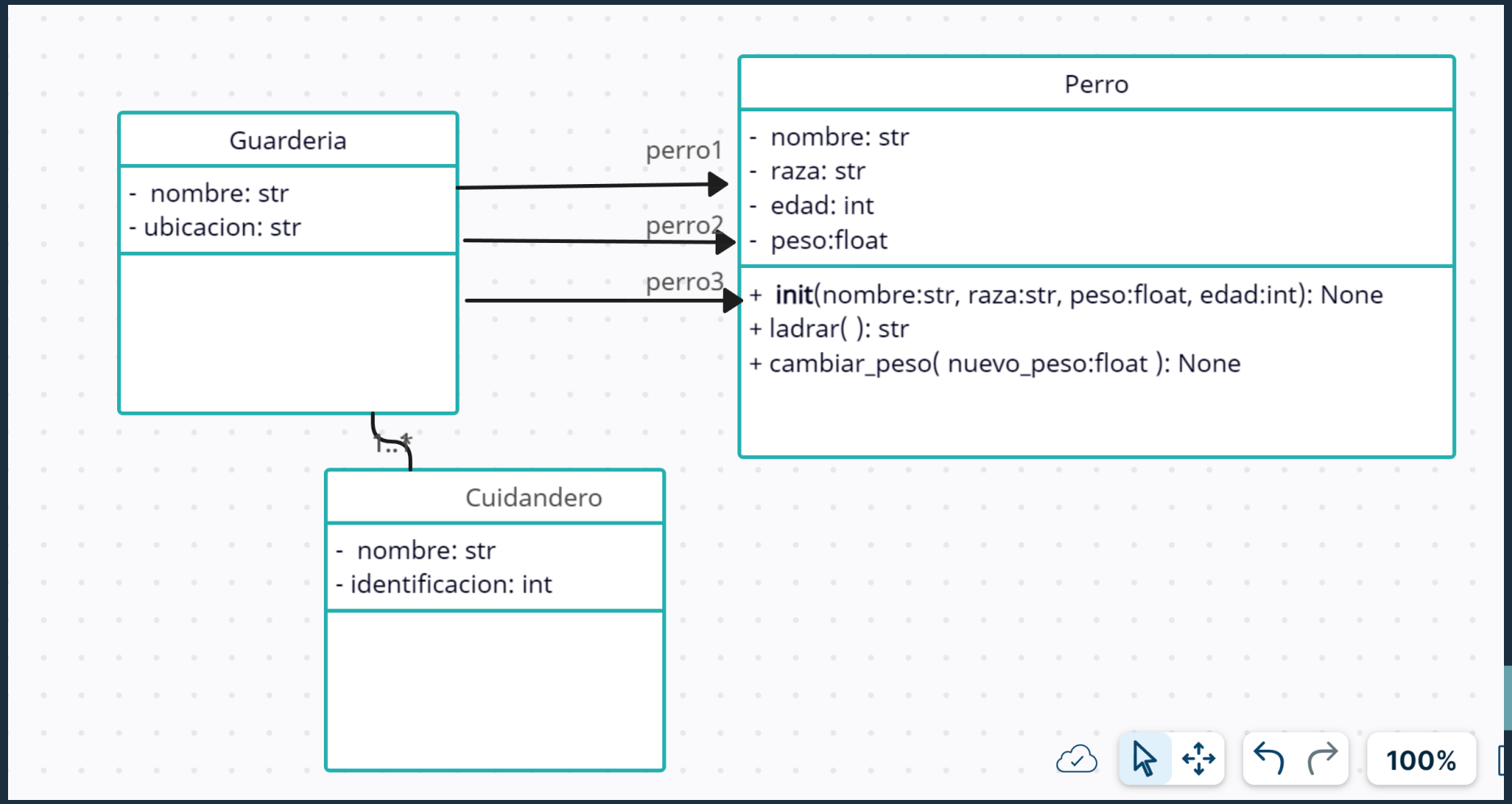
Asociaciones:

Cuando hay más de una clase en la aplicación, en el diagrama UML se utilizan las asociaciones para relacionar las diferentes clases.
Association: es la relación más básica e indica que una clase conoce a otra
Inheritance: relación de herencia
Realization: indica que la clase implementa una interface
Dependencia: una clase depende de otra para cumplir algunas funcionalidades
Aggregation: Asociación de uno a mucho donde la parte depende del todo
Composition: Relación uno a muchos donde la parte no depende del todo



UML

Para mostrar un ejemplo de asociaciones, supongamos que ahora tendremos una clase Guarderia y un Cuidandero



PROGRAMACION ORIENTADA A OBJETOS

- Clases y Objetos
- UML
- Encapsulamiento

ENCAPSULAMIENTO

El encapsulamiento es la medida de que tan acopladas están las clases dentro del programa, es decir que tan dependiente es una clase de las demás.

Los principios de diseño para la programación orientada a objetos como SOLID y GRASP siempre sugieren respetar el encapsulamiento para que el acoplamiento entre las clases sea el más bajo posible, y por ende, la cohesión alta.

Entre menor sea el acoplamiento entre las clases, más fácil de modificar y mantener será, dado que si por el contrario, el acoplamiento es alto, al realizar cambios en una clase, implicará cambios en las demás clases.

Si el acoplamiento es bajo, los cambios deben estar localizados en solo 1 clase.

Recordemos la explicación de porque separamos lógica e interfaz.

ENCAPSULAMIENTO

Para explicar el encapsulamiento hablemos primero de la visibilidad (que aplica tanto para atributos como para métodos).

En la programación orientada a objetos tenemos 3 visibilidades:

- public: puede ser accedido directamente desde fuera de la clase donde se encuentra declarado.
  En Python se pueden ver así: `self.nombre`
- private: puede ser accedido únicamente por la clase donde se encuentra declarado.
  En Python se marca con __ (doble) así: `self.__nombre`
- protected: puede ser accedido por la clase donde se encuentra declarado y las clases que lo extiendan (herencia)
  En Python se marca con _ así: `self._nombre`

ENCAPSULAMIENTO

La mejor manera de respetar el encapsulamiento es impedir que las clases pueden acceder o modificar los atributos de las otras clases, es decir, cada clase es responsable únicamente de sus propios atributos.
Para esto, lo más indicado es que los atributos de las clases sean privados, o si estamos utilizando herencia pueden ser protegidos, pero idealmente no deberían ser públicos.
Si una clase necesita la información que está en un atributo de otra clase, lo debe hacer a través de algún método. Por ejemplo, si necesita consultar el valor, se puede crear el método dar_atributo (conocido como getter), si se necesita modificar el valor, se puede crear el método modificar_atributo (conocido como setter), como el modificar_peso en el ejemplo del Perro.

ENCAPSULAMIENTO

Para respetar el encapsulamiento, vamos a modificar la clase Perro para que los atributos ahora sean privados.

```
3  class Perro:
4      def __init__(self, nombre, raza, peso, edad):
5          self.__nombre = nombre
6          self.__raza   = raza
7          self.__peso   = peso
8          self.__edad   = edad
```

Para que la Guarderia pueda conocer la información de los perros, debemos crear métodos getter (dar), en este ejemplo se creó el dar_nombre y dar_información, que agrupa toda la información en un único método.

```
16     def dar_nombre(self):
17         return self.__nombre
18
19     def dar_información(self):
20         return self.__nombre + "(" + self.__raza + "):"+str(self.__peso)+"/"+str(self.__edad)
```

ENCAPSULAMIENTO

Ahora comprobemos desde Guarderia que a través de los métodos dar, podemos conocer la información de los atributos de perro, para los 3 perros de la guarderia.

El resultado fue el siguiente:

perro.py | guarderia.py*

```
1  # -*- coding: utf-8 -*-
2
3  from perro import Perro
4
5  class Guarderia:
6      def __init__(self):
7          self._nombre="German"
8
9  perro_1 = Perro("Zeus", "Rottweiler", 45.8, 3)
10 perro_2 = Perro("Nala", "Golden R.", 8.5, 0)
11 perro_3 = Perro("Atila", "Alabai", 58.9, 5)
12
13
14 print(perro_1.dar_información())
15 print(perro_2.dar_información())
16 print(perro_3.dar_información())
```

Console 1/A

```
In [40]: runfile('C:/Users/(
guarderia.py', wdir='C:/User
Reloaded modules: perro
Zeus(Rottweiler):45.8|3
Nala(Golden R.):8.5|0
Atila(Alabai):58.9|5
```

MANOS A LA OBRA

Taller 3 disponible en BloqueNeon