



BACKEND CON PYTHON



HERENCIA Y POLIMORFISMO

- Clases abstractas
- Herencia
- Interfaces
- Polimorfismo



HERENCIA Y POLIMORFISMO

- Clases abstractas
- Herencia
- Interfaces
- Polimorfismo



CLASES ABSTRACTAS

Retomemos el ejemplo del Perro, ahora la Guarderia también recibirá Gatos

```
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
9
10    def ladrar(self):
11        return "¡Guau, guau!"
12
13    def modificar_peso(self, nuevo_peso):
14        self.__peso = nuevo_peso
15
16    def dar_nombre(self):
17        return self.__nombre
18
```

```
3 class Gato:
4     def __init__(self, nombre, raza, peso, edad):
5         self.__nombre = nombre
6         self.__raza = raza
7         self.__peso = peso
8         self.__edad = edad
9
10    def maullar(self):
11        return "¡Miau, miau!"
12
13    def modificar_peso(self, nuevo_peso):
14        self.__peso = nuevo_peso
15
16    def dar_nombre(self):
17        return self.__nombre
18
```

Perro y Gato son dos clases con mucho en común, y si la Guarderia recibiera más animales, seguramente tendrían una estructura (atributos y métodos) muy similar.

CLASES ABSTRACTAS

Para evitar estar creando un archivo nuevo por cada animal con una clase nueva, que va a tener atributos y métodos que ya hemos declarado vamos a crear una clase en común llamada Animal que tenga todos los atributos y métodos que serán iguales para todos los animales.

Sin embargo, hay un método (ladrar/maullar) que no es igual. Es acá donde la clase abstracta juega su papel más importante, crea un método abstracto (que no tendrá implementación), pero que tiene un nombre común: `hacer_sonido`

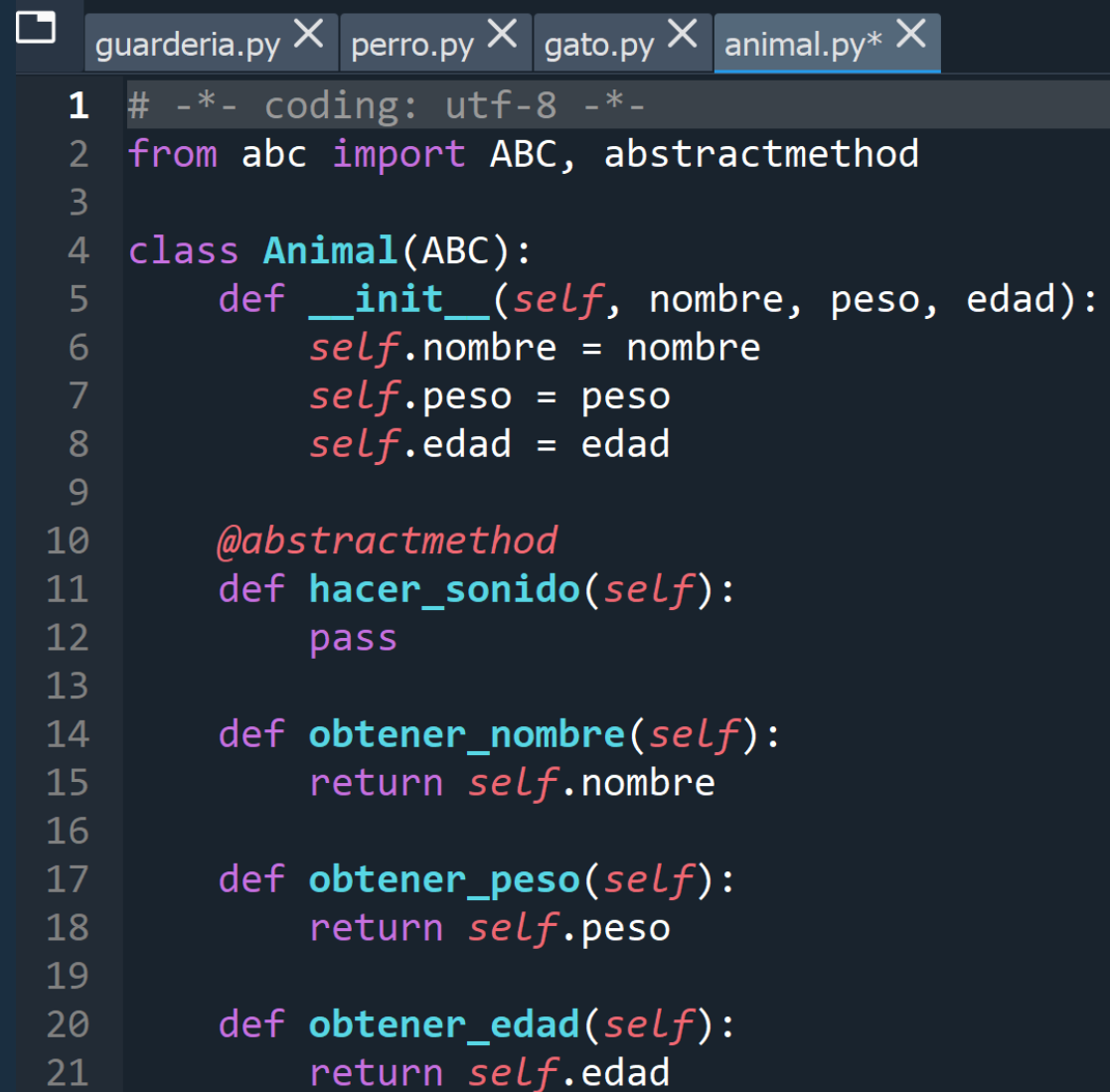
Para tener una clase abstracta, es decir una clase donde tenemos algunos métodos con implementación y otros no, usaremos una librería de Python llamada `abc` (Abstract Base Classes).

CLASES ABSTRACTAS

Como podemos ver, se parece mucho a una clase normal, solo que ahora en la declaración de la clase recibe como parámetro ABC.

Los métodos que no son comunes para Perro y Gato (y futuros animales), tienen la anotación `@abstractmethod`, además de que no tienen implementación, en lugar tienen la instrucción `pass`.

Sin embargo, hasta el momento Gato, Perro y Animal no están conectadas: acá aparece la herencia.



```
1  # -*- coding: utf-8 -*-
2  from abc import ABC, abstractmethod
3
4  class Animal(ABC):
5      def __init__(self, nombre, peso, edad):
6          self.nombre = nombre
7          self.peso = peso
8          self.edad = edad
9
10     @abstractmethod
11     def hacer_sonido(self):
12         pass
13
14     def obtener_nombre(self):
15         return self.nombre
16
17     def obtener_peso(self):
18         return self.peso
19
20     def obtener_edad(self):
21         return self.edad
```

HERENCIA Y POLIMORFISMO

- Clases abstractas
- Herencia
- Interfaces
- Polimorfismo



HERENCIA

Para poder relacionar las clases Perro y Gato con Animal, de manera que tanto Perro como Gato, puedan usar los métodos de Animal ahora deben heredar de la clase animal.

Cuando una clase (Perro/Gato), hereda de otra clase (Animal), puede usar los atributos y métodos como propios, sin que estén declarados en la clase. Es por esta razón que la herencia nos da herramientas de reutilización muy potentes, porque solo tenemos que implementar los atributos y métodos una vez.

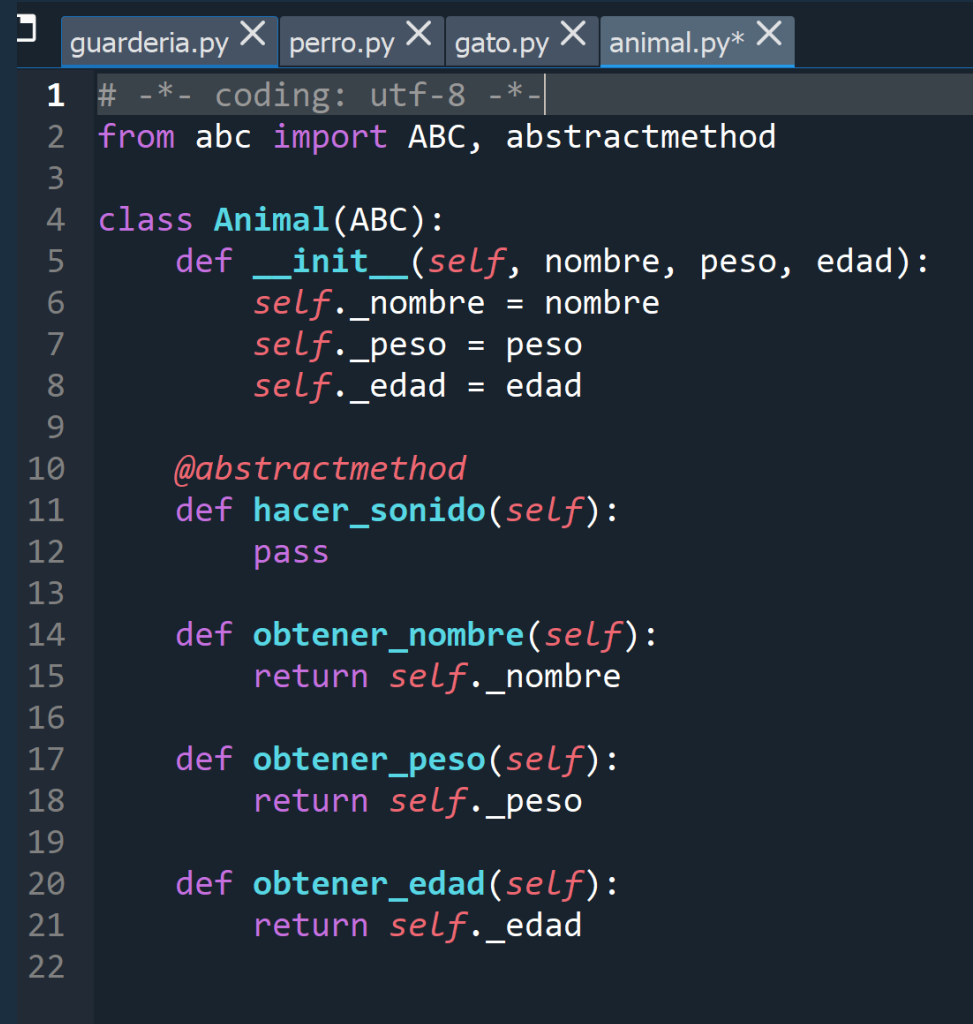
Las demás clases que puedan venir, como por ejemplo Conejo, Cerdo, entre otros, solo deben heredar de Animal y ya tendrán la mayoría de funcionalidades implementadas.

HERENCIA

Para que Perro (y Gato), pueda heredar de Animal, debemos hacer la siguiente modificación en Animal: los atributos y métodos que van a ser heredados deben ser protected: self.nombre ahora es self._nombre, lo mismo para peso y edad.

Si los dejamos como public, cualquier clase lo puede usar, inclusive las que no están en la herencia y esto incumpliría el encapsulamiento.

Si los dejamos private, nadie diferente a Animal lo puede usar.



```
guarderia.py X perro.py X gato.py X animal.py* X
1  # -*- coding: utf-8 -*-
2  from abc import ABC, abstractmethod
3
4  class Animal(ABC):
5      def __init__(self, nombre, peso, edad):
6          self._nombre = nombre
7          self._peso = peso
8          self._edad = edad
9
10     @abstractmethod
11     def hacer_sonido(self):
12         pass
13
14     def obtener_nombre(self):
15         return self._nombre
16
17     def obtener_peso(self):
18         return self._peso
19
20     def obtener_edad(self):
21         return self._edad
22
```

HERENCIA

Para que Perro (y Gato), pueda heredar de Animal, debemos hacer las siguientes modificaciones en Perro y Gato.

- Importar animal
- En la declaración de la clase recibir como parámetro Animal (la super clase de la cual se hereda)
- Quitar los métodos que ya está haciendo Animal, como obtener_nombre, obtener_edad, obtener_peso

El atributo raza no está en animal porque es exclusivo de Perro. Lo mismo para Gato y color.

```
3 from animal import Animal
4
5 class Perro(Animal):
6     def __init__(self, nombre, peso, edad, raza):
7         super().__init__(nombre, peso, edad)
8         self.__raza = raza
9
10    def hacer_sonido(self):
11        return "¡Guau, guau!"
12
13    def obtener_raza(self):
14        return self._raza
```

```
1 # -*- coding: utf-8 -*-
2 from animal import Animal
3
4 class Gato(Animal):
5     def __init__(self, nombre, peso, edad, color):
6         super().__init__(nombre, peso, edad)
7         self.__color = color
8
9     def hacer_sonido(self):
10        return "¡Miau, miau!"
11
12    def obtener_color(self):
13        return self._color
```

HERENCIA

Cuando hay herencia hay dos nuevos conceptos: superclase (padre), que es de la cual heredan, y subclase (hijo) que son las que heredan.

En nuestro ejemplo la superclase es Animal y las subclases son Perro y Gato.

Los métodos que estén en la superclase, solo deben ser implementados en las subclases si tienen la anotación de @abstractmethod, como es el caso de hacer_sonido. **Sonidos diferentes!**

Las subclases pueden sobrescribir la implementación de la superclase (como __init__):

Para extenderla, deben declararla y la primera línea debe ser super().método_sobrescrito, para heredar la función de la super clase (línea 6) y además de eso agregar líneas adicionales, como asignar raza/color (línea 7).

```
3 from animal import Animal
4
5 class Perro(Animal):
6     def __init__(self, nombre, peso, edad, raza):
7         super().__init__(nombre, peso, edad)
8         self.__raza = raza
9
10    def hacer_sonido(self):
11        return "¡Guau, guau!"
12
13    def obtener_raza(self):
14        return self._raza
15
16 from animal import Animal
17
18 class Gato(Animal):
19     def __init__(self, nombre, peso, edad, color):
20         super().__init__(nombre, peso, edad)
21         self.__color = color
22
23     def hacer_sonido(self):
24         return "¡Miau, miau!"
25
26     def obtener_color(self):
27         return self._color
```

HERENCIA

Veamos la implementación de Perro (aplica también para Gato) y Animal y la siguiente instrucción de Guarderia

```
3 from animal import Animal
4
5 class Perro(Animal):
6     def __init__(self, nombre, peso, edad, raza):
7         super().__init__(nombre, peso, edad)
8         self.__raza = raza
9
10    def hacer_sonido(self):
11        return "¡Guau, guau!"
12
13    def obtener_raza(self):
14        return self.__raza
```

```
guarderia.py X perro.py X gato.py X animal.py* X
1 # -*- coding: utf-8 -*-
2 from abc import ABC, abstractmethod
3
4 class Animal(ABC):
5     def __init__(self, nombre, peso, edad):
6         self.__nombre = nombre
7         self.__peso = peso
8         self.__edad = edad
9
10    @abstractmethod
11    def hacer_sonido(self):
12        pass
13
14    def obtener_nombre(self):
15        return self.__nombre
16
17    def obtener_peso(self):
18        return self.__peso
19
20    def obtener_edad(self):
21        return self.__edad
22
```

```
guarderia.py X perro.py X gato.py X animal.py X
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)
8
9 print(perro_1.obtener_nombre()+":"+str(perro_1.obtener_edad()))
10 print(perro_2.obtener_nombre()+":"+str(perro_2.obtener_edad()))
11 print(perro_3.obtener_nombre()+":"+str(perro_3.obtener_edad()))
```

En guarderia, los objetos de tipo Perro (perro_1,perro_2 y perro_3), están haciendo un llamado a funciones que no están en Perro pero si en Animal. Pero como tenemos la herencia correctamente implementada el resultado por consola es el siguiente:

```
In [47]: runfile('C:\Users\alejo\Documents\Python\guarderia.py', wdir='C:\Users\alejo\Documents\Python')
Reloaded modules: animal, perro
Zeus:45.8
Nala:8.5
Atila:58.9
```

HERENCIA Y POLIMORFISMO

- Clases abstractas
- Herencia
- Interfaces
- Polimorfismo



INTERFACES

Ya vimos como las clases abstractas nos permitían tomar unas características y un comportamiento que podían tener en común unas clases y agruparlas para implementarla sólo una vez y poder reutilizarlas.

Ahora veremos como asegurarnos que las clases tengan siempre algunos métodos con un nombre específico.

Es aquí donde aparecen las interfaces: son clases que no tienen atributos y ninguno de sus métodos tiene implementación:

- Nos dicen el que, no el como.
- Son contratos funcionales: obligan que las clases concretas o abstractas tengan que implementar dichas funcionalidades.
- No confundir con interfaz de consola o interfaz web.

INTERFACES

Ya vimos como las clases abstractas nos permitían tomar unas características y un comportamiento que podían tener en común unas clases y agruparlas para implementarla sólo una vez y poder reutilizarlas.

Ahora veremos como asegurarnos que las clases tengan siempre algunos métodos con un nombre específico.

Es aquí donde aparecen las interfaces: son clases que no tienen atributos y ninguno de sus métodos tiene implementación:

- Nos dicen el que, no el como.
- Son contratos funcionales: obligan que las clases concretas o abstractas tengan que implementar dichas funcionalidades.
- No confundir con interfaz de consola o interfaz web.

INTERFACES

Vamos a pensar que ahora tenemos Animal, que representa los animales domésticos y Animal_granja que representa los animales rurales.

Ambos tipos de animales tendrán algo en común y es que llevarán un registro de cuantos kilos come el animal, pero tienen diferencias, como por ejemplo que el animal de granja no tiene nombre, ni edad (no son relevantes). Además la forma de contar los kilos comidos es diferente: cada vez que comen los animales domésticos aprovechan el 100%, mientras que los de granja solo el 80%.

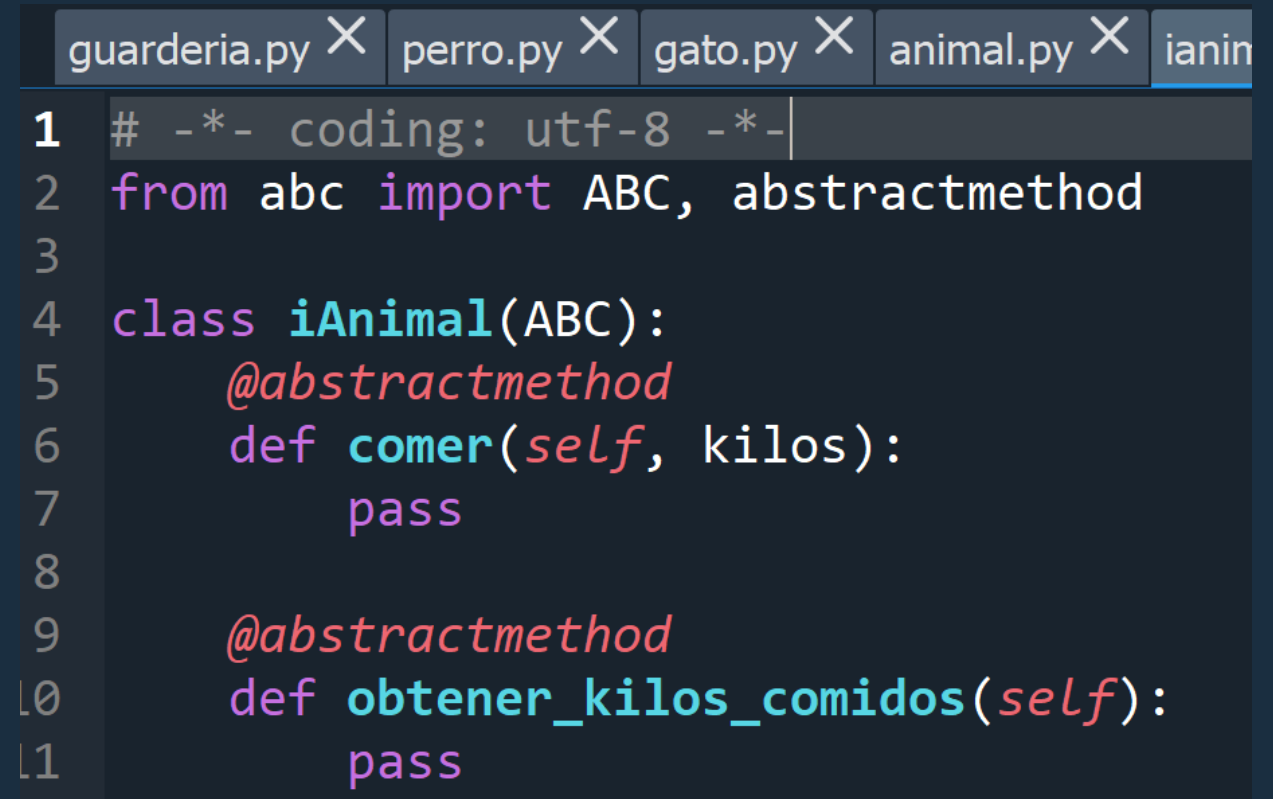
Para asegurarnos que en la guardería podamos manejar una lista con todos los animales (tanto domésticos como de granja) y podamos darles comida y consultar la cantidad total, vamos a crear una interface llamada ianimal.

INTERFACES

Para asegurarnos que en la guardería podamos manejar una lista con todos los animales (tanto domésticos como de granja) y podamos darles comida y consultar la cantidad total, vamos a crear una interface llamada ianimal.

Las interfaces, al igual que las clases abstractas usan la librería ABC, por lo que debemos importarla y ponerla como parámetro en la declaración de la clase.

Adicionalmente, todos los métodos son abstractos.



The screenshot shows a code editor with five tabs: guarderia.py, perro.py, gato.py, animal.py, and ianimal.py. The ianimal.py tab is active. The code defines an abstract class iAnimal that inherits from ABC. It includes two abstract methods: comer and obtener_kilos_comidos. The code is as follows:

```
1 # -*- coding: utf-8 -*-
2 from abc import ABC, abstractmethod
3
4 class iAnimal(ABC):
5     @abstractmethod
6     def comer(self, kilos):
7         pass
8
9     @abstractmethod
10    def obtener_kilos_comidos(self):
11        pass
```

INTERFACES

Ahora, debemos modificar animal y animal_granja para asegurarnos que implementen la interface ianimal.

Deben importar iAnimal y modificar el parámetro de la clase de ABC por iAnimal, adicionalmente ahora deben implementar los métodos comer y obtener_kilos_comidos

La clase animal_granja es mucho mas sencilla y no tiene los otros atributos que si tiene animal

La implementación de comer es diferente en ambos tipos de animales

```
guarderia.py X perro.py X gato.py X animal.py* X ianimal.py
1 # -*- coding: utf-8 -*-
2 from iAnimal import iAnimal
3
4 class Animal(iAnimal):
5     def __init__(self, nombre, peso, edad):
6         self._nombre = nombre
7         self._peso = peso
8         self._edad = edad
9         self._kilos_comidos=0
10
11     def comer(self, kilos):
12         self._kilos_comidos+=kilos
13
14     def obtener_kilos_comidos(self):
15         return self._kilos_comidos
16
17     def hacer_sonido(self):
18         pass
19
20     def obtener_nombre(self):
21         return self._nombre
22
23     def obtener_peso(self):
24         return self._peso
25
26     def obtener_edad(self):
27         return self._edad
28
```

```
guarderia.py X perro.py X gato.py X animal.py* X ianimal.py X anima
1 # -*- coding: utf-8 -*-
2 from iAnimal import iAnimal
3
4 class Animal_Granja(iAnimal):
5     def __init__(self, nombre, peso, edad):
6
7         self._kilos_comidos=0
8
9     def comer(self, kilos):
10         self._kilos_comidos+=(kilos*0.8)
11
12     def obtener_kilos_comidos(self):
13         return self._kilos_comidos
14
```

HERENCIA Y POLIMORFISMO

- Clases abstractas
- Herencia
- Interfaces
- Polimorfismo



POLIMORFISMO

El polimorfismo es la capacidad de un objeto de tomar muchas formas. En la programación orientada a objetos se da gracias a la herencia con clases abstractas e interfaces: un objeto (Perro), puede ser considerado como instancia de su clase, como instancia de la superclase (Animal) o de las interfaces que implementa (iAnimal).

A continuación mostramos como un `perro_1` es instancia de `Perro`, `Animal` e `iAnimal`, pero no de `Animal_Granja`

```
guarderia.py X perro.py X gato.py X animal.py X ianimal.py X animal_gra
1 # -*- coding: utf-8 -*-
2
3 from perro import Perro
4 from ianimal import iAnimal
5 from animal import Animal
6 from animal_granja import Animal_Granja
7
8 perro_1 = Perro("Zeus", "Rottweiler", 45.8, 3)
9
10 print("Es perro")
11 print(isinstance(perro_1, Perro))
12 print("Es animal")
13 print(isinstance(perro_1, Animal))
14 print("Es ianimal")
15 print(isinstance(perro_1, iAnimal))
16 print("Es animal_granja")
17 print(isinstance(perro_1, Animal_Granja))
```

```
In [55]: runfile('C:/U
4/guarderia.py', wdir=
clase 4')
Reloaded modules: iani
Es perro
True
Es animal
True
Es ianimal
True
Es animal_granja
False
```

MANOS A LA OBRA

Taller 4 disponible en BloqueNeon

