

Lecture 08 - Object-Oriented Programming (OOP)-done

March 23, 2024

1 Lecture 8: Object-Oriented Programming (OOP)

OOP is a way of thinking about algorithms, and a way of structuring and organizing your code.

You define *objects*, and you give those objects *variable* and *functions*, which you then use as needed.

You already do this without knowing it! For example `lists` are objects. They have a variable which stores the things in the list. They have functions like `append` and `pop` that you use.

```
[ ]: L = [1, 2, 3]
      L.append(4)
      print(L)
```

```
[1, 2, 3, 4]
```

Here's a simple example of a class:

```
[ ]: class Pet:
      # two underscores __ init __
      def __init__(self, animal_name, animal_species, animal_age):
          self.name = animal_name
          self.species = animal_species
          self.age = animal_age
          self.fake = "some value"
          print("You have just made a Pet named", self.name)
```

```
[ ]: hermes = Pet("Hermes", "cat", 15)
```

You have just made a Pet named Hermes

```
[ ]: test = Pet("Hermes", "cat")
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 test = Pet("Hermes", "cat")

TypeError: Pet.__init__() missing 1 required positional argument: 'animal_age'
```

```
[ ]: print(hermes)
<__main__.Pet object at 0x00000001285df408>
[ ]: hermes.fake
[ ]: 'some value'
[ ]: hermes.name
[ ]: 'Hermes'
[ ]: hermes.species
[ ]: 'cat'
[ ]: hermes.age
[ ]: 15
[ ]: type(hermes)
[ ]: __main__.Pet
[ ]: print(hermes)
```

```
<__main__.Pet object at 0x000000015926d558>
```

“dunder methods” - “double underscore”

Every class need a `__init__` function (that is two underscores on each side). This function is called automatically when you create a new *instance* of an object. (`hermes` is an *instance* of the class `Pet`.)

The first parameter to `__init__` in its definition always has to be `self`, which is how an object refers to itself. But when you’re creating an instance later, you don’t pass in a value for `self` – it’s automatically added.

In our `Pet` class, we take the three inputs, `name`, `species`, and `age`, and we assign those to *class variables* `self.name`, `self.species`, and `self.age` so they are remembered by the object.

Representing things with classes makes it easier to keep track of the meaning of different variables.

```
[ ]: class Job:
    def __init__(self, index, duration, deadline, profit):
        self.index = index
        self.duration = duration
        self.deadline = deadline
        self.profit = profit
```

```
[ ]: j = [2,1,5]
      print(j[0])
      J = Job(1, 2, 1, 5)
```

2

```
[ ]: J.deadline
```

```
[ ]: 1
```

```
[ ]: J.profit
```

```
[ ]: 5
```

```
[ ]: J.duration
```

```
[ ]: 2
```

```
[ ]: # f-string example
      x = 1
      y = 3
      z = x + y
      print(f"The sum of {x} and {y} is {x+y}.")
      print("The sum of " + str(x) + " and " + str(y) + " is " + str(z) + ".")
```

The sum of 1 and 3 is 4.

The sum of 1 and 3 is 4.

```
[ ]: test = 0.198719284798721
      print(f"{test:.2f}")
```

0.20

Now let's define some functions in the Pet class.

```
[ ]: class Pet:

      def __init__(self, name, species, age, noise):
          self.name = name
          self.species = species
          self.age = age
          self.noise = noise

      def speak(self):
          # string = ""
          # string += self.name
          # string += " says "
          # string += self.noise
```

```

    # string += "."
    # print(string)
    print(f"{self.name} says {self.noise}.")

def print_info(self):
    # f-string
    string = f"{self.name} is a {self.species} whose age is {self.age}."
    print(string)

def age_in_human_years(self):

    # "species" would be a variable that is local to this function
    # "self.species" refers to the "species" variable stored by the
    # whole object
    if self.species == "cat":
        return 7 * self.age
    elif self.species == "dog":
        return 11 * self.age
    elif self.species == "turtle":
        return 4 * self.age
    else:
        return None

```

```
[ ]: hermes = Pet("Hermes", "cat", 15, "meow")
```

```
[ ]: hermes.age
```

```
[ ]: 15
```

```
[ ]: hermes.speak()
```

Hermes says meow.

```
[ ]: hermes.print_info()
```

Hermes is a cat whose age is 15.

```
[ ]: hermes_age_in_human_years = hermes.age_in_human_years()
print(hermes_age_in_human_years)
```

105

```
[ ]: print(hermes)
```

<__main__.Pet object at 0x00000001309046e8>

It would be better, especially for debugging, if we could print the object and have it show us useful information.

This is where the Python magic comes in. In addition to defining class functions we want to be able to call, we can also define some “dunder methods” that automatically change some behavior

of the objects.

They called “dunder methods” because their names start and end with double underscores.

The first one we’ll see is `__str__`. Whenever you try to print an object or get its string representation, it secretly calls `obj.__str__()` in the background.

```
[ ]: class Pet:

    def __init__(self, name, species, age, noise):
        self.name = name
        self.species = species
        self.age = age
        self.noise = noise

    def speak(self):
        string = ""
        string += self.name
        string += " says "
        string += self.noise
        string += "."
        print(string)

    def print_info(self):
        string = f"{self.name} is a {self.species} whose age is {self.age}."
        print(string)

    def age_in_human_years(self):

        # "species" would be a variable that is local to this function
        # "self.species" refers to the "species" variable stored by the
        # whole object
        if self.species == "cat":
            return 7 * self.age
        elif species == "dog":
            return 11 * self.age
        elif species == "turtle":
            return 4 * self.age
        else:
            return None

    def __str__(self):
        # return "this would be a string"
        return f"{self.name} - {self.species} - {self.age}"
```

```
[ ]: hermes = Pet("Hermes", "cat", 15, "meow")
```

```
[ ]: hermes.__str__()
```

```
[ ]: 'Hermes - cat - 15'
```

```
[ ]: str(hermes)
```

```
[ ]: 'Hermes - cat - 15'
```

A closely related dunder method is `__repr__`, which tells Python how to show the object when you just use its name (without `print`)

```
[ ]: class Pet:

    def __init__(self, name, species, age, noise):
        self.name = name
        self.species = species
        self.age = age
        self.noise = noise

    def speak(self):
        string = ""
        string += self.name
        string += " says "
        string += self.noise
        string += "."
        print(string)

    def print_info(self):
        string = f"{self.name} is a {self.species} whose age is {self.age}."
        print(string)

    def age_in_human_years(self):

        # "species" would be a variable that is local to this function
        # "self.species" refers to the "species" variable stored by the
        # whole object
        if self.species == "cat":
            return 7 * self.age
        elif species == "dog":
            return 11 * self.age
        elif species == "turtle":
            return 4 * self.age
        else:
            return None

    def __str__(self):
        return f"{self.name} - {self.species} - {self.age}"

    def __repr__(self):
```

```
        return f"Pet('{self.name}', '{self.species}', {self.age}, '{self.
↳noise}')
```

```
[ ]: hermes = Pet("Hermes", "cat", 15, "meow")
```

```
[ ]: print(hermes)
```

```
Hermes - cat - 15
```

```
[ ]: hermes
```

```
[ ]: Pet('Hermes', 'cat', 15, 'meow')
```

```
[ ]: new_hermes = Pet('Hermes', 'cat', 15, 'meow')
new_hermes.print_info()
```

```
Hermes is a cat whose age is 15.
```

```
[ ]: L1 = [1,2,3]
L2 = [1,2,3]

L1 == L2
```

```
[ ]: True
```

```
[ ]: id(L1), id(L2)
```

```
[ ]: (4970933544, 4970933496)
```

One very important thing to keep in mind is that, by default, two objects are equal (==) only if they are literally the same object at the same memory location.

```
[ ]: hermes1 = Pet("Hermes", "cat", 15, "meow")
hermes2 = Pet("Hermes", "cat", 15, "meow")
```

```
[ ]: hermes1 == hermes2
```

```
[ ]: False
```

```
[ ]: id(hermes1)
```

```
[ ]: 5102133336
```

```
[ ]: id(hermes2)
```

```
[ ]: 5102133392
```

```
[ ]: hermes1 == hermes1
```

```
[ ]: True
```

```
[ ]: third_hermes = hermes1
```

```
[ ]: id(hermes1)
```

```
[ ]: 5102133336
```

```
[ ]: id(third_hermes)
```

```
[ ]: 5102133336
```

```
[ ]: hermes1 == third_hermes
```

```
[ ]: True
```

```
[ ]: L3 = list(L1)
      id(L1), id(L3)
```

```
[ ]: (4970933544, 5265568432)
```

```
[ ]: L1 = [[1,2,3], [4,5,6]]
      L2 = list(L1)
      id(L1), id(L2)
      print(id(L1[0]), id(L2[0]))
```

```
5266073168 5266073168
```

```
[ ]: print(L1)
      print(L2)
```

```
[[1, 2, 3], [4, 5, 6]]
[[1, 2, 3], [4, 5, 6]]
```

```
[ ]: from copy import deepcopy
```

```
[ ]: hermes4 = deepcopy(hermes1)
```

```
[ ]: hermes4.print_info()
```

```
Hermes is a cat whose age is 15.
```

```
[ ]: hermes4 == hermes1
```

```
[ ]: False
```

```
[ ]: id(hermes1), id(hermes4)
```



```
[ ]: (5102133336, 5804001336)
```

You can change this with the `__eq__` dunder method, which redefines when two objects are equal, but you should consider whether you **should**. If you are writing patient management software for a veterinary clinic, do you want to consider two animals to be the same animal if they have the same name / species / age? Probably not!

```
[ ]: class Pet:

    def __init__(self, name, species, age, noise):
        self.name = name
        self.species = species
        self.age = age
        self.noise = noise

    def speak(self):
        string = ""
        string += self.name
        string += " says "
        string += self.noise
        string += "."
        print(string)

    def print_info(self):
        string = f"{self.name} is a {self.species} whose age is {self.age}."
        print(string)

    def age_in_human_years(self):

        # "species" would be a variable that is local to this function
        # "self.species" refers to the "species" variable stored by the
        # whole object
        if self.species == "cat":
            return 7 * self.age
        elif species == "dog":
            return 11 * self.age
        elif species == "turtle":
            return 4 * self.age
        else:
            return None

    def __str__(self):
        return f"{self.name} / {self.species} / {self.age}"

    def __repr__(self):
        return f"Pet('{self.name}', '{self.species}', {self.age}, '{self.
↵noise}')"

```

```

def __eq__(self, other):
    """
    return True if [self] and [other] have identical names, species, and
↪ages
    """
    # import random
    # return bool(random.randint(0,1))
    # are_they_equal = (self.name == other.name and self.species == other.
↪species and self.age == other.age)
    # return are_they_equal
    # if self.name == other.name and self.species == other.species and self.
↪age == other.age:
    #     return True
    # return False
    return self.name == other.name and self.species == other.species and
↪self.age == other.age

```

```

[ ]: pet1 = Pet("Hermes", "cat", 15, "meow")
pet2 = Pet("Vaughn", "dog", 13, "woof")
pet1 == pet2

```

[]: False

```

[ ]: hermes1 = Pet("Hermes", "cat", 15, "meow")
hermes2 = Pet("Hermes", "cat", 15, "meow")

```

```

[ ]: print(id(hermes1))
print(id(hermes2))
hermes1 == hermes2

```

```

5805535704
5805535648

```

[]: True

```

[ ]: hermes1 = Pet("Hermes", "cat", 15, "meow")
hermes2 = Pet("Hermes", "cat", 15, "growl")

```

```

[ ]: hermes1 == hermes2

```

[]: True

One last dunder method for now: if you want to be able to compare two objects with < and >, define `__lt__`.

```

[ ]: hermes1 < hermes2

```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[58], line 1  
----> 1 hermes1 < hermes2  
  
TypeError: '<' not supported between instances of 'Pet' and 'Pet'
```

```
[ ]: class Pet:  
  
    def __init__(self, name, species, age, noise):  
        self.name = name  
        self.species = species  
        self.age = age  
        self.noise = noise  
  
    def speak(self):  
        string = ""  
        string += self.name  
        string += " says "  
        string += self.noise  
        string += ". "  
        print(string)  
  
    def print_info(self):  
        string = f"{self.name} is a {self.species} whose age is {self.age}."  
        print(string)  
  
    def age_in_human_years(self):  
  
        # "species" would be a variable that is local to this function  
        # "self.species" refers to the "species" variable stored by the  
        #   whole object  
        if self.species == "cat":  
            return 7 * self.age  
        elif species == "dog":  
            return 11 * self.age  
        elif species == "turtle":  
            return 4 * self.age  
        else:  
            return None  
  
    def __str__(self):  
        return f"{self.name} / {self.species} / {self.age}"  
  
    def __repr__(self):
```

```

        return f"Pet('{self.name}', '{self.species}', {self.age}, '{self.
↳noise}')"

    def __eq__(self, other):
        """
        return True if [self] and [other] have identical names, species, and
↳ages
        """
        return self.name == other.name and self.species == other.species and
↳self.age == other.age

    def __lt__(self, other):
        """
        return True if the age of self is less than the age of other
        """
        return self.age < other.age

```

```

[ ]: animals = [
    Pet("Hermes", "cat", 15, "meow"),
    Pet("Leopold", "cat", 13, "growl"),
    Pet("Vaughn", "dog", 13, "woof"),
    Pet("Malcolm", "cat", 11, "wheeze")
]

```

```

[ ]: animals

```

```

[ ]: [Pet('Hermes', 'cat', 15, 'meow'),
    Pet('Leopold', 'cat', 13, 'growl'),
    Pet('Vaughn', 'dog', 13, 'woof'),
    Pet('Malcolm', 'cat', 11, 'wheeze')]

```

```

[ ]: sorted(animals)

```

```

[ ]: [Pet('Malcolm', 'cat', 11, 'wheeze'),
    Pet('Leopold', 'cat', 13, 'growl'),
    Pet('Vaughn', 'dog', 13, 'woof'),
    Pet('Hermes', 'cat', 15, 'meow')]

```

What if we want to sort by species?

```

[ ]: class Pet:

    def __init__(self, name, species, age, noise):
        self.name = name
        self.species = species
        self.age = age
        self.noise = noise

```

```

def speak(self):
    string = ""
    string += self.name
    string += " says "
    string += self.noise
    string += "."
    print(string)

def print_info(self):
    string = f"{self.name} is a {self.species} whose age is {self.age}."
    print(string)

def age_in_human_years(self):

    # "species" would be a variable that is local to this function
    # "self.species" refers to the "species" variable stored by the
    # whole object
    if self.species == "cat":
        return 7 * self.age
    elif species == "dog":
        return 11 * self.age
    elif species == "turtle":
        return 4 * self.age
    else:
        return None

def __str__(self):
    return f"{self.name} / {self.species} / {self.age}"

def __repr__(self):
    return f"Pet('{self.name}', '{self.species}', {self.age}, '{self.
↪noise}')"

def __eq__(self, other):
    """
    return True if [self] and [other] have identical names, species, and
↪ages
    """
    return self.name == other.name and self.species == other.species and
↪self.age == other.age

def __lt__(self, other):
    """
    return True if the age of self is less than the age of other
    """
    return self.species < other.species

```

```
[ ]: "h" < "efg"
```

```
[ ]: False
```

```
[ ]: help(str.__lt__)
```

```
Help on function __lt__:
```

```
__lt__(self, other)  
    x.__lt__(y) <=> x<y
```

```
[ ]: animals
```

```
[ ]: [Pet('Hermes', 'cat', 15, 'meow'),  
      Pet('Leopold', 'cat', 13, 'growl'),  
      Pet('Vaughn', 'dog', 13, 'woof'),  
      Pet('Malcolm', 'cat', 11, 'wheeze')]
```

```
[ ]: animals.sort()
```

```
[ ]: animals
```

```
[ ]: [Pet('Malcolm', 'cat', 11, 'wheeze'),  
      Pet('Leopold', 'cat', 13, 'growl'),  
      Pet('Vaughn', 'dog', 13, 'woof'),  
      Pet('Hermes', 'cat', 15, 'meow')]
```

```
[ ]: animals.sort(key=lambda pet: pet.name)
```

```
[ ]: animals
```

```
[ ]: [Pet('Hermes', 'cat', 15, 'meow'),  
      Pet('Leopold', 'cat', 13, 'growl'),  
      Pet('Malcolm', 'cat', 11, 'wheeze'),  
      Pet('Vaughn', 'dog', 13, 'woof')]
```

```
[ ]: class A:  
      def __init__(self, value):  
          self.value = value  
  
      def __eq__(self, other):  
          return True  
  
class B:  
      def __init__(self, value):  
          self.value = value
```

```
def __eq__(self, other):  
    return False
```

```
[ ]: thing1 = A(10)  
thing2 = B(10)
```

```
[ ]: thing1 == thing2
```

```
[ ]: True
```

```
[ ]: thing2 == thing1
```

```
[ ]: False
```

```
[ ]: thing1.__eq__(thing2)
```

```
[ ]: True
```

```
[ ]: thing1 < thing2
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[81], line 1  
----> 1 thing1 < thing2  
  
TypeError: '<' not supported between instances of 'A' and 'B'
```

Let's do one more example from scratch.

```
[ ]: class Rectangle:  
  
    def __init__(self, h, w):  
        self.height = h  
        self.width = w  
  
    def perimeter(self):  
        return 2 * self.height + 2 * self.width  
  
    def area(self):  
        return self.height * self.width  
  
    def double_dimensions(self):  
        return Rectangle(2 * self.height, 2 * self.width)  
  
    def double_this_rectangle(self):  
        self.height *= 2
```

```

        self.width *= 2

    def __eq__(self, other):
        return self.height == other.height and self.width == other.width

    def __lt__(self, other):
        return self.area() < other.area()

    def __str__(self):
        top = "o" + ("-" * self.width) + "o\n"
        side = "|" + (" " * self.width) + "|\n"
        return top + (side * self.height) + top

    def __repr__(self):
        return f"Rectangle({self.height}, {self.width})"

```

```
[ ]: R = Rectangle(3,4)
```

```
[ ]: R
```

```
[ ]: Rectangle(3, 4)
```

```
[ ]: print(R)
```

```

o----o
|    |
|    |
|    |
o----o

```

```
[ ]: R.double_dimensions()
```

```
[ ]: Rectangle(6, 8)
```

```
[ ]: R
```

```
[ ]: Rectangle(3, 4)
```

```
[ ]: print(R.double_dimensions())
```

```

o-----o
|       |
|       |
|       |
|       |
|       |
|       |

```



```
o-----o
```

```
[ ]: print(R.double_dimensions().double_dimensions())
```

```
o-----o
|
|
|
|
|
|
|
|
o-----o
```

```
[ ]: R
```

```
[ ]: Rectangle(3, 4)
```

```
[ ]: R.double_this_rectangle()
```

```
[ ]: R
```

```
[ ]: Rectangle(6, 8)
```

```
[ ]: R.area()
```

```
[ ]: 48
```

```
[ ]: R.perimeter()
```

```
[ ]: 28
```

```
[ ]: import random
random_rectangles = [Rectangle(random.randint(1,6), random.randint(1,6)) for i_
↳in range(10)]
```

```
[ ]: random_rectangles
```

```
[ ]: [Rectangle(6, 2),
Rectangle(6, 1),
Rectangle(4, 5),
Rectangle(2, 6),
```

```
Rectangle(4, 6),
Rectangle(4, 2),
Rectangle(5, 5),
Rectangle(5, 5),
Rectangle(2, 5),
Rectangle(3, 3)]
```

```
[ ]: for R in random_rectangles:
      print(R)
```

```
o--o
|  |
|  |
|  |
|  |
|  |
|  |
o--o
```

```
o-o
| |
| |
| |
| |
| |
| |
o-o
```

```
o-----o
|       |
|       |
|       |
|       |
o-----o
```

```
o-----o
|       |
|       |
o-----o
```

```
o-----o
|       |
|       |
|       |
o-----o
```

```
o--o
```

```
| |  
| |  
| |  
| |  
o--o
```

```
o-----o  
|         |  
|         |  
|         |  
|         |  
o-----o
```

```
o-----o  
|         |  
|         |  
|         |  
|         |  
o-----o
```

```
o-----o  
|         |  
|         |  
o-----o
```

```
o---o  
|   |  
|   |  
|   |  
o---o
```

```
[ ]: for rectangle in sorted(random_rectangles):  
      print(repr(rectangle))  
      print(rectangle)
```

```
Rectangle(6, 1)
```

```
o-o  
| |  
| |  
| |  
| |  
| |  
| |  
o-o
```

Rectangle(4, 2)

```
o--o
|  |
|  |
|  |
|  |
o--o
```

Rectangle(3, 3)

```
o---o
|   |
|   |
|   |
o---o
```

Rectangle(2, 5)

```
o-----o
|       |
|       |
o-----o
```

Rectangle(6, 2)

```
o--o
|  |
|  |
|  |
|  |
|  |
|  |
o--o
```

Rectangle(2, 6)

```
o-----o
|       |
|       |
o-----o
```

Rectangle(4, 5)

```
o-----o
|       |
|       |
|       |
|       |
o-----o
```

Rectangle(4, 6)

```
o-----o
|       |
```

```
| |
| |
| |
o-----o
```

Rectangle(5, 5)

```
o-----o
| |
| |
| |
| |
o-----o
```

Rectangle(5, 5)

```
o-----o
| |
| |
| |
| |
o-----o
```

[]:

[]:

```
[ ]: for R in sorted(random_rectangles, key=lambda rect: rect.perimeter()):
      print(repr(R))
      print(R)
      # print((R.perimeter(), R.area()))
```

Rectangle(4, 2)

```
o--o
| |
| |
| |
| |
o--o
```

Rectangle(3, 3)

```
o---o
| |
| |
| |
o---o
```

Rectangle(6, 1)

```
o-o
| |
| |
| |
| |
| |
| |
o-o
```

Rectangle(2, 5)

```
o-----o
|       |
|       |
o-----o
```

Rectangle(6, 2)

```
o--o
| |
| |
| |
| |
| |
| |
o--o
```

Rectangle(2, 6)

```
o-----o
|       |
|       |
o-----o
```

Rectangle(4, 5)

```
o-----o
|       |
|       |
|       |
o-----o
```

Rectangle(4, 6)

```
o-----o
|       |
|       |
|       |
o-----o
```

Rectangle(5, 5)

```
o-----o
|       |
|       |
|       |
|       |
|       |
o-----o
```

Rectangle(5, 5)

```
o-----o
|       |
|       |
|       |
|       |
|       |
o-----o
```

```
[ ]: # sort first by perimeter, then if tied by area
      for R in sorted(random_rectangles, key=lambda rect: (rect.perimeter(), rect.
        ↪area())):
          print(repr(R))
          print(R)
```

Rectangle(4, 2)

```
o--o
| |
| |
| |
| |
o--o
```

Rectangle(3, 3)

```
o---o
|   |
|   |
|   |
o---o
```

Rectangle(6, 1)

```
o-o
| |
| |
| |
| |
| |
```

```
| |  
o-o
```

Rectangle(2, 5)

```
o-----o  
|       |  
|       |  
o-----o
```

Rectangle(6, 2)

```
o--o  
|  |  
|  |  
|  |  
|  |  
|  |  
|  |  
o--o
```

Rectangle(2, 6)

```
o-----o  
|       |  
|       |  
o-----o
```

Rectangle(4, 5)

```
o-----o  
|       |  
|       |  
|       |  
|       |  
o-----o
```

Rectangle(4, 6)

```
o-----o  
|       |  
|       |  
|       |  
|       |  
o-----o
```

Rectangle(5, 5)

```
o-----o  
|       |  
|       |  
|       |  
|       |  
|       |
```



```

o-----o

Rectangle(5, 5)
o-----o
|       |
|       |
|       |
|       |
|       |
o-----o

```

Advanced Topic: Hashability (how to make objects that can be put in sets) (See me if you have any questions!)

```

[ ]: hermes
[ ]: Pet('Hermes', 'cat', 15, 'meow')
[ ]: leopold = Pet('Leopold', 'cat', 13, 'meow')
[ ]: [hermes, leopold]
[ ]: [Pet('Hermes', 'cat', 15, 'meow'), Pet('Leopold', 'cat', 13, 'meow')]
[ ]: set([hermes, leopold])

```

```

-----
TypeError                                Traceback (most recent call last)
Cell In[118], line 1
----> 1 set([hermes, leopold])

TypeError: unhashable type: 'Pet'

```

```

[ ]: hash([1,2,3])

```

```

-----
TypeError                                Traceback (most recent call last)
Cell In[119], line 1
----> 1 hash([1,2,3])

TypeError: unhashable type: 'list'

```

```

[ ]: hash((1,2,3))

```

```

[ ]: 529344067295497451

```

```
[ ]: hash(leopold)
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[122], line 1  
----> 1 hash(leopold)  
  
TypeError: unhashable type: 'Pet'
```

```
[ ]: class Pet:
```

```
    def __init__(self, name, species, age, noise):  
        self.name = name  
        self.species = species  
        self.age = age  
        self.noise = noise  
  
    def speak(self):  
        string = ""  
        string += self.name  
        string += " says "  
        string += self.noise  
        string += ". "  
        print(string)  
  
    def print_info(self):  
        string = f"{self.name} is a {self.species} whose age is {self.age}."  
        print(string)  
  
    def age_in_human_years(self):  
  
        # "species" would be a variable that is local to this function  
        # "self.species" refers to the "species" variable stored by the  
        # whole object  
        if self.species == "cat":  
            return 7 * self.age  
        elif species == "dog":  
            return 11 * self.age  
        elif species == "turtle":  
            return 4 * self.age  
        else:  
            return None  
  
    def __str__(self):  
        return f"{self.name} / {self.species} / {self.age}"
```

```

def __repr__(self):
    return f"Pet('{self.name}', '{self.species}', {self.age}, '{self.
↳noise}')"

def __eq__(self, other):
    """
    return True if [self] and [other] have identical names, species, and
↳ages
    """
    if not isinstance(other, Pet):
        return False
    return self.name == other.name and self.species == other.species and
↳self.age == other.age and self.noise == other.noise

def __lt__(self, other):
    """
    return True if the age of self is less than the age of other
    """
    return self.age < other.age

def __hash__(self):
    return hash((self.name, self.species, self.age, self.noise))

```

```
[ ]: hermes = Pet("Hermes", "cat", 15, "meow")
leopold = Pet("Leopold", "cat", 13, "meow")
extra = Pet("Hermes", "cat", 15, "meow")
```

```
[ ]: hash(hermes), hash(leopold), hash(extra)
```

```
[ ]: (-3859940691860368505, -7136126380760223751, -3859940691860368505)
```

```
[ ]: {hermes, leopold, extra}
```

```
[ ]: {Pet('Leopold', 'cat', 13, 'meow'), Pet('Hermes', 'cat', 15, 'meow')}
```

If you give you give your object a hash function, you need to treat it like it's immutable.

```
[ ]:
```