

Lecture 09 - Object-Oriented Programming (OOP)

March 19, 2025

1 Lecture 9: 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.

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

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

Here's a simple example of a class:

```
[2]: 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"
```

```
[3]: hermes = Pet("Hermes", "cat", 16)
```

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

```
<__main__.Pet object at 0x1061e02f0>
```

```
[6]: hermes.fake
```

```
[6]: 'some value'
```

```
[7]: hermes.name
```

```
[7]: 'Hermes'
```

```
[8]: hermes.species
```

```
[8]: 'cat'
```

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

```
[9]: 16
```

```
[10]: type(hermes)
```

```
[10]: __main__.Pet
```

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

```
<__main__.Pet object at 0x1061e02f0>
```

“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.

```
[12]: class Job:

    def __init__(self, index, duration, deadline, profit):
        self.index = index
        self.duration = duration
        self.deadline = deadline
        self.profit = profit
```

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

```
2
```

```
[14]: J = Job(1, 2, 1, 5)
J.deadline
```

```
[14]: 1
```

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

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

```
[17]: # 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.

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

```
[18]: 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
```

```
[19]: hermes = Pet("Hermes", "cat", 16, "meow")
```

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

Hermes says meow.

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

Hermes is a cat whose age is 16.

```
[22]: hermes.age_in_human_years()
```

```
[22]: 112
```

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

```
<__main__.Pet object at 0x1061e1940>
```

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.

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

```
[24]: '<__main__.Pet object at 0x1061e1940>'
```

```
[25]: 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 self.species == "dog":
        return 11 * self.age
    elif self.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}"

```

```
[26]: hermes = Pet("Hermes", "cat", 16, "meow")
```

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

```
Hermes / cat / 16
```

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

```
[28]: 'Hermes / cat / 16'
```

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

```
[38]: 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 self.species == "dog":
        return 11 * self.age
    elif self.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}')"

```

```
[39]: hermes = Pet("Hermes", "cat", 16, "meow")
```

```
[40]: print(hermes) # uses __str__ method
```

Hermes / cat / 16

```
[41]: hermes # uses the __repr__ method
```

```
[41]: Pet('Hermes', 'cat', 16, 'meow')
```

```
[42]: new_hermes = Pet('Hermes', 'cat', 16, 'meow')
new_hermes.print_info()
```

Hermes is a cat whose age is 16.

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.

```
[43]: hermes1 = Pet("Hermes", "cat", 16, "meow")
hermes2 = Pet("Hermes", "cat", 16, "meow")
```

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

```
[44]: False
```

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

```
[45]: 4395610704
```

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

```
[46]: 4396970848
```

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

```
[47]: True
```

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

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

```
[49]: 4395610704
```

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

```
[50]: 4395610704
```

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

```
[51]: True
```

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

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

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

```
Hermes is a cat whose age is 16.
```

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

```
[56]: False
```

```
[57]: hermes4.age = 10
```

```
[59]: hermes1.print_info()
hermes4.print_info()
```

```
third_hermes.print_info()
```

```
Hermes is a cat whose age is 16.  
Hermes is a cat whose age is 10.  
Hermes is a cat whose age is 16.
```

```
[ ]:
```

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 self.species == "dog":  
            return 11 * self.age  
        elif self.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

```

```
[61]: hermes1 = Pet("Hermes", "cat", 16, "meow")
hermes2 = Pet("Hermes", "cat", 16, "meow")
```

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

```
4397606832
4399531344
```

```
[62]: True
```

```
[63]: hermes1 = Pet("Hermes", "cat", 16, "meow")
hermes2 = Pet("Hermes", "cat", 16, "growl")
```

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

```
[64]: True
```

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

```
[65]: hermes1 < hermes2
```

```

-----
TypeError                                Traceback (most recent call last)
Cell In[65], line 1
----> 1 hermes1 < hermes2

TypeError: '<' not supported between instances of 'Pet' and 'Pet'

```

```

[3]: 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 self.species == "dog":
            return 11 * self.age
        elif self.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

```

```

[84]: pet1 = Pet("A", "cat", 10, "meow")
      pet2 = Pet("B", "dog", 5, "woof")
      pet1 < pet2

```

[84]: False

```

[85]: animals = [
      Pet("Hermes", "cat", 16, "meow"),
      Pet("Leopold", "cat", 14, "growl"),
      Pet("Vaughn", "dog", 13, "woof"),
      Pet("Chicken Macaroni", "cat", 0, "purr")
      ]

```

```

[86]: animals

```

```

[86]: [Pet('Hermes', 'cat', 16, 'meow'),
      Pet('Leopold', 'cat', 14, 'growl'),
      Pet('Vaughn', 'dog', 13, 'woof'),
      Pet('Chicken Macaroni', 'cat', 0, 'purr')]

```

```

[87]: sorted(animals, key=lambda pet : pet.age)

```

```

[87]: [Pet('Chicken Macaroni', 'cat', 0, 'purr'),
      Pet('Vaughn', 'dog', 13, 'woof'),
      Pet('Leopold', 'cat', 14, 'growl'),
      Pet('Hermes', 'cat', 16, 'meow')]

```

```

[88]: animals

```

```

[88]: [Pet('Hermes', 'cat', 16, 'meow'),
      Pet('Leopold', 'cat', 14, 'growl'),
      Pet('Vaughn', 'dog', 13, 'woof'),
      Pet('Chicken Macaroni', 'cat', 0, 'purr')]

```

```

[89]: animals.sort()

```

```

[ ]:

```

```

[ ]:

```

```
[90]: animals
```

```
[90]: [Pet('Chicken Macaroni', 'cat', 0, 'purr'),  
       Pet('Vaughn', 'dog', 13, 'woof'),  
       Pet('Leopold', 'cat', 14, 'growl'),  
       Pet('Hermes', 'cat', 16, 'meow')]
```

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

```
[92]: animals
```

```
[92]: [Pet('Chicken Macaroni', 'cat', 0, 'purr'),  
       Pet('Hermes', 'cat', 16, 'meow'),  
       Pet('Leopold', 'cat', 14, 'growl'),  
       Pet('Vaughn', 'dog', 13, 'woof')]
```

```
[ ]:
```

Let's do one more example from scratch.

```
[93]: 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 __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})"
```

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

```
[95]: R
```

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

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

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

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

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

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

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

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

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

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

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

```
[100]: 12
```

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

```
[101]: 14
```

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

```
[103]: random_rectangles
```

```
[103]: [Rectangle(3, 5),
Rectangle(5, 2),
Rectangle(5, 4),
Rectangle(4, 1),
Rectangle(2, 1),
Rectangle(1, 1),
Rectangle(4, 1),
Rectangle(4, 5),
Rectangle(2, 5),
Rectangle(1, 6)]
```

```
[104]: 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
```

```
[105]: for R in sorted(random_rectangles):  
        print(R)  
        # print((R.perimeter(), R.area()))
```

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
```

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

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

(4, 1)

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

(6, 2)

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

(10, 4)

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

(10, 4)

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

o-----o

(14, 6)

o--o

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

(14, 10)

o-----o

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

(14, 10)

o-----o

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

(16, 15)

o-----o

o-----o

(18, 20)

o-----o

o-----o

(18, 20)

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

```
[5]: hermes1 = Pet("hermes", "cat", 16, "meow")
hermes2 = Pet("hermes", "cat", 16, "meow")
```

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

```
[6]: True
```

```
[7]: L = [hermes1, hermes2]
```

```
[8]: L
```

```
[8]: [Pet('hermes', 'cat', 16, 'meow'), Pet('hermes', 'cat', 16, 'meow')]
```

```
[9]: id(L[0]), id(L[1])
```

```
[9]: (4423186288, 4423181872)
```

```
[10]: set(L)
```

```
-----
TypeError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 set(L)

TypeError: unhashable type: 'Pet'
```

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

```
-----
TypeError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 hash([1,2,3])

TypeError: unhashable type: 'list'
```

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

```
[12]: 529344067295497451
```

```
[13]: hash(hermes1)
```

```
-----
TypeError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 hash(hermes1)
```

`TypeError: unhashable type: 'Pet'`

```
[16]: 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 self.species == "dog":
            return 11 * self.age
        elif self.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

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

```

[]:

[]:

[]:

```
[17]: hermes1 = Pet("Hermes", "cat", 16, "meow")
hermes2 = Pet("Hermes", "cat", 16, "meow")
```

```
[18]: hash(hermes1)
```

```
[18]: -987498866930948655
```

```
[19]: {hermes1, hermes2}
```

```
[19]: {Pet('Hermes', 'cat', 16, 'meow')}
```

```
[20]: L = [hermes1, hermes2]
```

```
[21]: L
```

```
[21]: [Pet('Hermes', 'cat', 16, 'meow'), Pet('Hermes', 'cat', 16, 'meow')]
```

```
[22]: set(L)
```

```
[22]: {Pet('Hermes', 'cat', 16, 'meow')}
```

[]: