Lecture 08 - Object-Oriented Programming (OOP)-in-progress

March 18, 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
```

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

```
<__main__.Pet object at 0x00000001209d8678>
```

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

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

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

[ ]: 'Hermes'

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

[ ]: 'cat'

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

[ ]: 15

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

[ ]: __main__.Pet

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

```
<__main__.Pet object at 0x00000001209d8678>
```

"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}.")
```

```
The sum of 1 and 3 is 4.
```

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)

    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.speak()
```

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

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

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

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 it's 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")
```

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

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

A closely related dunder method is `__repr__`, which tells Python how to show the object when you just use it's 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
```

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

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
```

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

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

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

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

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

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

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

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

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

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

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

[ ]:
```

```

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
        """
        return self.name == other.name and self.species == other.species and␣
↪self.age == other.age
```

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

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

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

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

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

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

[ ]:
```
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
```

[ ]:
```
sorted(animals)
```

[ ]:
```
animals
```

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

[ ]:
```
animals
```

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

[ ]:
```
animals
```

[ ]:
```

```

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 __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
```

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

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

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

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

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

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

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

[ ]:
```
random_rectangles
```

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

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

[ ]:
```

```

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

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

[ ]:
```
L = [hermes1, third_hermes]
```

[ ]:
```
L
```

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

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

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

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

[ ]:
```
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

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

[ ]:
```
hermes1 = Pet("Hermes", "cat", 13, "meow")
hermes2 = hermes1
```

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

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

[ ]:
```

```

[ ]:
```

```