Python Classes & Objects Tutorial

Playlist

https://www.youtube.com/playlist?list=PLzMcBGfZo4-l1MqB1zoYfqzlj_HH-ZzXt

Python OOP Tutorial (Object Orientated Programming) - Intro

Video

https://youtu.be/v_Jp11xqCzg?si=yW_r0aVu7qQa7xwo

Notes

- Everything is an object
  - `print(type("something"))` would show that the text is a ‘str’ class
  - By being an object, functionality (methods) can be created for that thing
- `help()` will fetch documentation
  - `help(int)`
  - `help(str)`

Video

https://youtu.be/jQiUOV15IRI?si=hOojLe_A6soqWEVM

Notes

- `self` means the instance you are calling on/currently using

Video

https://youtu.be/H2SQrZK2nvM?si=oGeI3CL63Y1Sc8IX

Notes

- Overriding occurs when the method signature is the same in the superclass and the child class. Overloading occurs when two or more methods in the same class have the same name but different parameters.
- With inheritance, you want one very general class to be the very top class
  - Then build functionality with the child classes

Video

https://youtu.be/39m3rstTN8w?si=PNVcC_7N2VeQZLu1

Notes

- You can override dunder methods, such as: `__add__`, `__sub__`, `__mul__` and `__div__`

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

# Example usage:
v1 = Vector(1, 2)
v2 = Vector(3, 4)

# Addition
result_add = v1 + v2
print("Addition:", result_add)

# Subtraction
result_sub = v1 - v2
print("Subtraction:", result_sub)

# Multiplication
result_mul = v1 * 2
print("Multiplication:", result_mul)
```

- You can also override `__str__` to change how an object is printed

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        self.x *= scalar
        self.y *= scalar
        return self
```

```
    def __str__(self):
        return "Vector({}, {})".format(self.x, self.y)

# Example usage:
v1 = Vector(1, 2)
v2 = Vector(3, 4)

# Addition
result_add = v1 + v2
print("Addition:", result_add)

# Subtraction
result_sub = v1 - v2
print("Subtraction:", result_sub)

# Multiplication
v1 *= 2
print("Multiplication:", v1)

# String representation
print("String representation:", v1)
```

- There is also: `__gt__` (greater than), `__ge__` (greater than or equal to), `__lt__` (less than), `__le__` (less than or equal to), `__eq__` (equal to)
- You could also override `__len__` which is useful for classes that represents a shape that has a particular way of calculating its length

---

Video

https://youtu.be/MpuOuZKWUWw?si=Ic_vOh4KqdFZavQ-

Notes

- An instance variable is one that is prefixed with `self.` and is specific to an instance of the class
- A class variable is one that is NOT prefixed with `self.` and is shared among all instances of a class

- In the code below, every time an instance of `Dog` is created, that new instance will be appended to the class variable `dogs` which stores a list of all created `Dog` objects

```
class Dog:
    dogs = []

    def __init__(self, name):
        self.name = name
        self.dogs.append(self)

tim = Dog("Tim")
jim = Dog("Jim")
print(Dog.dogs)
```

- Decorators are things that you put above your methods to indicate they are a special type of method

```
class Dog:
    dogs = []

    def __init__(self, name):
        self.name = name
        self.dogs.append(self)

    @classmethod
    def num_dogs(cls):
        return len(cls.dogs)

    @staticmethod
    def bark(n):
        for _ in range(n):
            print("Bark!")

tim = Dog("Tim")
jim = Dog("Jim")
print(Dog.dogs)
```

- `@classmethod` is for “class methods” which are methods that can be called on the name of the class
  - `cls` means the name of the class
  - You would be able to run `print(Dog.num_dogs())`

    - You can still also run `tim.num_dogs())` if you wanted to use an instance of the class instead

- `@staticmethods` is for “static methods” which are methods within a class that have no access to anything else in the class (no self keyword or cls keyword)
    - Useful for when you want a method to act as a function, but be organised within a class
        - This is particularly useful when you are creating ‘import modules’ such as a `Math` class

```
class Dog:
    dogs = []

    def __init__(self, name):
        self.name = name
        self.dogs.append(self)

    @classmethod
    def num_dogs(cls):
        return len(cls.dogs)

    @staticmethod
    def bark(n):
        for _ in range(n):
            print("Bark!")

Dog.bark(5)
```

Video

https://youtu.be/xY__sjI5yVU?si=UZaEKXQU9AuJcWYq

Notes

- A private class is something that can only be accessed from within a certain file or directory and a private method is something that can only be called from within the

class. A public class or method is something that can be accessed anywhere.

- There is actually no real thing in Python as public and private
  - There are just conventions to represent them
- A private class is created by putting a single underscore in front of the class name
  - `class _Dog:`