

# DISCUSSION 07

---

## OOP, String Representation



Mingxiao Wei

[mingxiaowei@berkeley.edu](mailto:mingxiaowei@berkeley.edu)

Oct 14, 2022

# LOGISTICS

---

- ANTS is released!  
  - Checkpoint 1 due tomorrow 10/14
  - Checkpoint 2 due next Tue 10/18
  - The whole project due next Fri 10/21
  - Submit by next Thu 10/20 for one extra point!
  - There's one extra credit question, worth 2 pts - if you want the extra credit from early submission AND this question, make sure to finish this question by next Thursday as well!
- Reminder - Homework 04 recovery (Ed post [#1757](#))

# OBJECT-ORIENTED PROGRAMMING

---



# OBJECT-ORIENTED PROGRAMMING

---

OOP - a programming paradigm that allows us to treat code as objects, extending the idea of data abstraction.

- **class** - a template for objects
- **instance** - a single object created from a class
- **attributes**
  - **instance variable** - a data attribute of an object, specific to an instance
  - **class variable** - a data attribute of an object, shared by all instances of a class
  - **method** - a bound function that may be called on all instances of a class
  - Use **dot notation** to access attributes - `Class.attribute` or `instance.attribute`

# CLASS/INSTANCE ATTRIBUTES VS. OTHER VARIABLES

- Attributes = class/instance variables + methods
- Class variables - use `ClassName.variable` or `InstanceName.variable` to access
  - The latter only works if the instance does not have an instance variable of the same name as the class variable
- Instance variables - use `InstanceName.variable` to access
- Other variables - arguments to a method, or otherwise initiated elsewhere, but are not class/instance variables

```
class Car:
```

```
    num_wheels = 4
```

```
    gas_level = 30
```

```
    def __init__(self, model):
```

```
        self.model = model
```

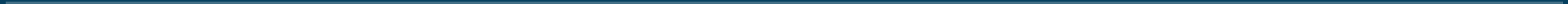
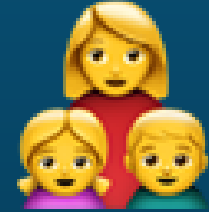
```
        self.num_wheels = Car.num_wheels
```

```
        self.gas = Car.gas_level
```

# WORKSHEET Q1, 2

---

# INHERITANCE



# INHERITANCE

---

- To avoid redefining attributes and methods for similar classes, we can write a single **base class** from which the similar classes **inherit**. A subclass *is a* more specific version of the base class.
- `SubClass(BaseClass)`
- By default, a subclass has the same behavior as its base class - unless we override/add additional attributes
- **Overriding attributes**
  - Class variables - reassign
  - Methods - redefine the method with the same function signature (name and arguments) in the subclass, but different function bodies. Use `super()` to call the same method from the base class when necessary (DRY - Don't Repeat Yourself!)



# INHERITANCE

---

- `super().method(args)`
  - Used to call the corresponding `method` from the base class
  - Can only be used inside of a class method
  - `self` is implicitly bound to the instance on which the `method` is called - no need to pass it in
- `Class.method(instance, args)`
  - Used to call the specific `method` from a specific `class` (can be any class - doesn't have to be the base class!)
  - Can be used anywhere
  - Need to explicitly pass in the `instance`

# INHERITANCE

---

```
class Pet:
    def __init__(self, name, owner):
        self.is_alive = True      # It's alive!!!
        self.name = name
        self.owner = owner

    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")

    def talk(self):
        print(self.name)

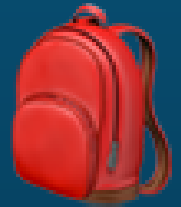
# A dog is a pet!
class Dog(Pet):
    def talk(self):
        super().talk()
        print('This Dog says woof!')
```

# WORKSHEET Q3, 4

---

# CLASS METHODS

---



# DECORATORS

---

- A decorator is a function that takes in a function and returns another function
- The `@decorator` syntax is a syntax sugar

```
def f(arg):  
    ...  
f = classmethod(f)  
# Above and below are equivalent ways of  
# using the classmethod decorator  
@classmethod  
def f(arg):  
    ...
```

# CLASS METHODS

---

- Instead of receiving the instance as the first argument ( `self` ), the method will receive the class itself ( `cls` )
- Commonly used to create "factory methods": methods that **construct and return a new instance of the class**.
- Use the `@classmethod` decorator to turn a method into a class method

```
class Dog(Pet):  
    # With the previously defined methods not written out  
    @classmethod  
    def robo_factory(cls, owner):  
        return cls("RoboDog", owner)
```

With `Dog.robo_factory(owner_name)`, we can create a Dog instance with the name "RoboDog" whose owner has the name `owner_name`, without having to call the Dog constructor with the dog name "RoboDog" every time (`Dog("Robodog", owner_name)`)

# WORKSHEET Q5

---

# REPRESENTATION

---





# REPRESENTATION

---

- To produce a string representation of an object in, we use `str()` or `repr()`
- `str(obj)`
  - Returns `obj.__str__()`
  - Used to describe the object in a human-readable form
  - Called when an object gets printed - `print(obj)` is essentially `print(str(obj))`
    - If `__str__` is not defined for `obj`, use `repr(obj)` instead
- `repr(obj)`
  - Returns `obj.__repr__()`
  - Used to describe the object in a computer-readable form
  - Evaluating an object in the interpreter is essentially `print(repr(obj))`
  - By convention, this should return a string that, when evaluated, returns an object with the same value

# INHERITANCE

```
class Rational:
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __str__(self):
        return f"{self.numer}/{self.denom}"
    def __repr__(self):
        return f"Rational({self.numer}, {self.denom})"
```

```
>>> a = Rational(1, 2)
```

```
>>> str(a)
```

```
'1/2'
```

```
>>> repr(a)
```

```
'Rational(1, 2)'
```

```
>>> print(a)
```

```
1/2
```

```
>>> a
```

```
Rational(1, 2)
```

# WORKSHEET Q6

---

# ATTENDANCE! 🤠

---

[go.cs61a.org/mingxiao-att](https://go.cs61a.org/mingxiao-att)

- The attendance form and slides are both linked on our [section website!](#)
- Once again, please do remember to fill out the form by midnight today!!