Object Oriented Programming

Object Oriented Programming

The Three Principles of OOP

1. **Encapsulation/Abstraction**

   - Message passing, control complexity

2. **Inheritance**

   - Code reuse and hierarchical relationships

3. **Polymorphism**

   - Objects can take on many forms

Encapsulation

Repeat after me:

"I will not violate abstraction barriers" (or DeNero will set your code on fire)

Basic principles

- Keep fields (data) hidden from users
- Only allow "behavior" to modify data
- Objects communicate via message passing

Encapsulation

Consider the following example:

- Driving (or just accelerating) a car
    - Data: fuel total, engine status, motor rpm, car weight, wheel angle etc.
    - Behavior: press gas pedal, press brake, turn steering wheel
    - Pedals are a convenient interface for controlling speed. Imagine the driver manually feeding the fuel line while driving!

Encapsulation

```
class Car:
    """ Description for a basic driving car. """
    def __init__(self, ...):
        """ Set whatever initial state here. """
        ...
    def steer(self, turn_deg):
        """ Steer car in a direction. """
        ...
    def step(self, which_ped, step_force):
        """ Step on a pedal with certain force. """
        ...
    def check_dash(self, attr):
        """ Checks a part of the dashboard and returns it. """
    ...
    Other methods here that the user should not worry about.
    Ex: def burn_fuel(self):
    ...
```

Abstraction Barriers

Which lines have abstraction barriers? Assume the functions all work properly and you are the car user.

```
19 def drive_then_stop(my_car):
20     my_car.check_engine()
21     my_car.steer(30)
22     my_car.burn_fuel(my_car.fuel/10)
23     my_car.step("accel", 20)
24     while my_car.check_dash("speed") <= 40:
25         print(my_car.speed)
26     my_car.brake();
```

Abstraction Barriers

Which lines have abstraction barriers? Assume the functions all work properly and you are the car user.

```
19 def drive_then_stop(my_car):
20     my_car.check_engine()
21     my_car.steer(30)
22     my_car.burn_fuel(my_car.fuel/10)
23     my_car.step("accel", 20)
24     while my_car.check_dash("speed") <= 40:
25         print(my_car.speed)
26     my_car.brake();
```

Encapsulation

Doesn’t look hard now, but encapsulation will become an important design consideration soon.

Be able to identify suspicious lines in classes, especially in code cross out questions!

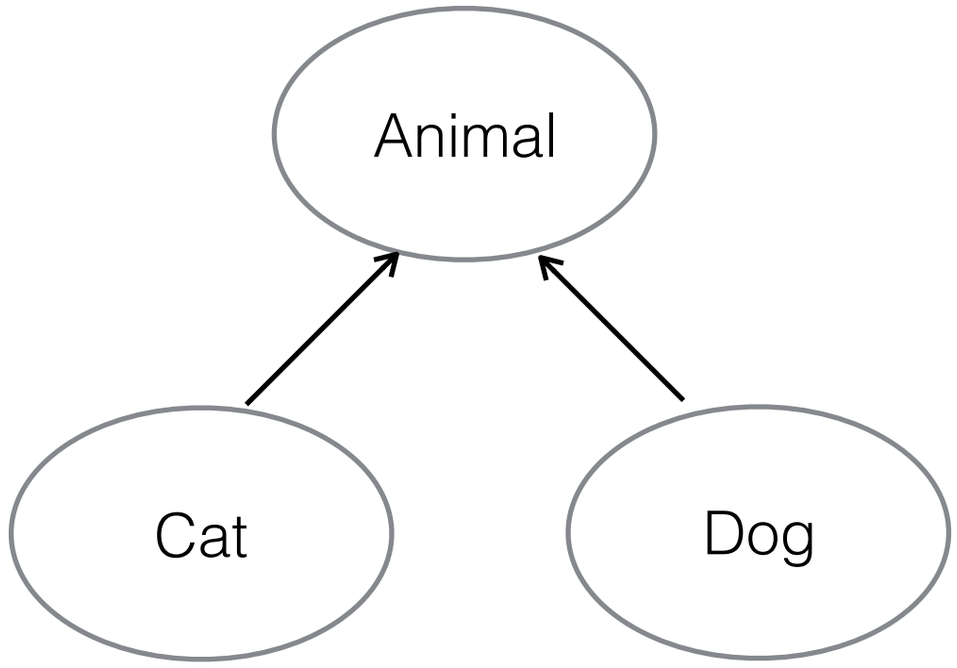
Questions?

Next is Inheritance.

Inheritance

Also plays heavily into polymorphism:

Simplifies code structure using “is-a” relationships



Cat **is an** Animal and Dog **is an** Animal but Cat is not a Dog

Inheritance

Inherit **fields** and **methods** from your parent class

- Don’t have to use them, can choose to override
- However, “default” behavior is always present

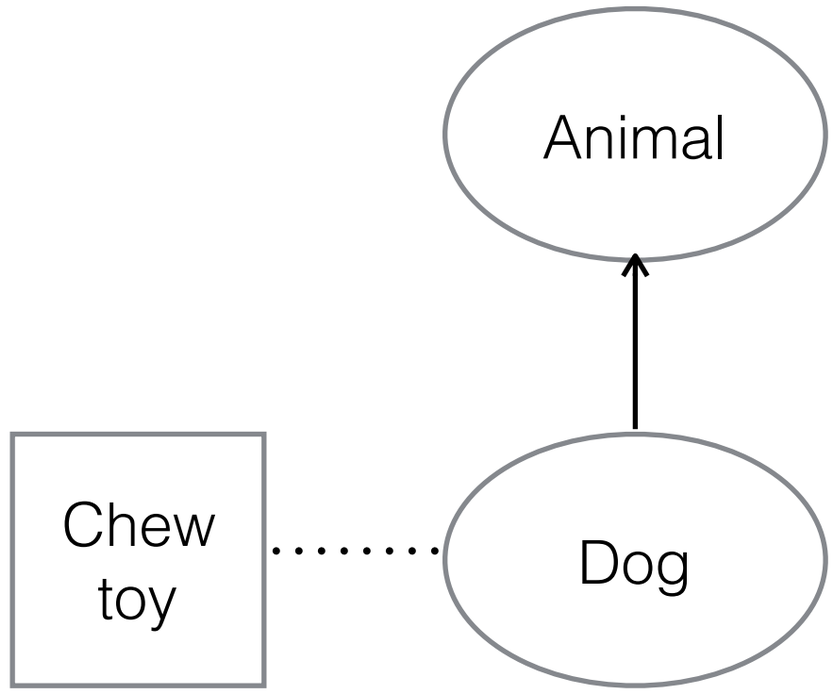
Possibility of **multiple inheritance**

- Can be problematic (what if you inherit two different things with the same name?)
- Not in the “spirit” of OOP

Inheritance

Beware: not everything should be inherited (“is-a”)!

Sometimes, composition or **“has-a”** relationships are better.



Dog **is an** Animal and **has a** chew toy.

Polymorphism

Ability for a class to take on many forms

- For example, pick the right animal sound based upon the type of animal
  - Based upon the **instance**

Also important: class vs. instance data

- Implicit self for bound methods
- Instance data overriding class data

Class vs Instance

Classes have methods; instances have a **bound method** associated with them

Dot expressions used to pass in an instance into "self" (review 2.5 in your textbook if this is confusing)

```
class Fruit:
    …
    def peel(self):
    …
orange.peel()
```

This is implicitly "self"

Class vs Instance

Predict the output:

```
1  class Fruit:
2      radioactivity = 0
3
4      def __init__(self, flavor):
5          self.flavor = flavor
6
7      def eat(self):
8          if self.radioactivity > 0:
9              return "delicious " + self.flavor
10         return "ok " + self.flavor
```

```python
class Fruit:
    radioactivity = 0

    def __init__(self, flavor):
        self.flavor = flavor

    def eat(self):
        if self.radioactivity > 0:
            return "delicious " + self.flavor
        return "ok " + self.flavor
```

```
>>> banana = Fruit("potassium")
>>> banana.eat()
>>> Fruit.radioactivity = 100
>>> Fruit.eat()
>>> banana.eat()
>>> banana.radioactivity
>>> Fruit.eat(banana)
>>> banana.radioactivity = 2000
>>> Fruit.radioactivity
>>> banana.radioactivity
>>> banana.eat(banana)
```

```python
class Fruit:
    radioactivity = 0

    def __init__(self, flavor):
        self.flavor = flavor

    def eat(self):
        if self.radioactivity > 0:
            return "delicious " + self.flavor
        return "ok " + self.flavor
```

Which lines (if any) will Error?

```
>>> banana = Fruit("potassium")
>>> banana.eat()
>>> Fruit.radioactivity = 100
>>> Fruit.eat()
>>> banana.eat()
>>> banana.radioactivity
>>> Fruit.eat(banana)
>>> banana.radioactivity = 2000
>>> Fruit.radioactivity
>>> banana.radioactivity
>>> banana.eat(banana)
```

```python
class Fruit:
    radioactivity = 0

    def __init__(self, flavor):
        self.flavor = flavor

    def eat(self):
        if self.radioactivity > 0:
            return "delicious " + self.flavor
        return "ok " + self.flavor
```

Which lines (if any) will Error?

```
>>> banana = Fruit("potassium")
>>> banana.eat()
>>> Fruit.radioactivity = 100
>>> Fruit.eat()
>>> banana.eat()
>>> banana.radioactivity
>>> Fruit.eat(banana)
>>> banana.radioactivity = 2000
>>> Fruit.radioactivity
>>> banana.radioactivity
>>> banana.eat(banana)
```

```
>>> banana = Fruit(“potassium”)
>>> banana.eat()
“ok potassium”
>>> Fruit.radioactivity = 100
>>> banana.eat()
“delicious potassium”
>>> banana.radioactivity
100
>>> Fruit.eat(banana)
“delicious potassium”
>>> banana.radioactivity = 2000
>>> Fruit.radioactivity
100
>>> banana.radioactivity
2000
```

Solution for **executed** lines:

Class vs Instance

The purpose of this exercise was to highlight the differences between **class** and **instance**.

- Instance variables **take precedence** over class variables (instances are more specific than classes)

- However, new instance **defaults** to the class variables unless they are changed in the constructor (common) or somehow modified elsewhere.

Class vs Instance

OOP is not meant to be tricky; focus on the big ideas, rather than edge cases.
What should be put in a class vs. in an instance?

Class	Instance
State shared amongst all of the same type	State or functionality for a more specialized version of a class
Variables you can “rely upon.” Will always be available to check	Not always the best place to put things. Consider making a subclass?
Frequently a bad idea to change class state after initial declaration	Constructors are the primary vehicles for carrying specific instance state

Object Oriented Programming

Test your understanding. Let's say we're designing a boat class. Decide if these attributes should be class or instance based.

```
Class Boat(Vehicle):
```

Variable	Class/Instance
is_waterproof	
max_capacity	
top_speed	
total_boats	

Object Oriented Programming

Test your understanding. Let's say we're designing a boat class. Decide if these attributes should be class or instance based.

```
Class Boat(Vehicle):
```

Variable	Class/Instance
is_waterproof	**Class.** All boats are waterproof
max_capacity	**Instance.** Unique to a specific boat.
top_speed	**Instance.** Unique to a specific boat
total_boats	**Class.** A bit weird/contrived, but all boats should be monitored by the Boat class.