Inheritance + Composition
Motivation
Building "Animal Conserving"

A game where we take care of cute furry/ferocious animals:
What should be the classes?
What should be the classes?

Panda()
Lion()
Rabbit()
Vulture()
Elephant()
Food()
A Food class

Let's start simple:

class Food:

    def __init__(self, name, type, calories):
        self.name = name
        self.type = type
        self.calories = calories

How would we use that class?
A Food class

Let's start simple:

class Food:
    def __init__(self, name, type, calories):
        self.name = name
        self.type = type
        self.calories = calories

How would we use that class?

broccoli = Food("Broccoli Rabe", "veggies", 20)
bone_marrow = Food("Bone Marrow", "meat", 100)
An Elephant class

```python
class Elephant:
    species_name = "African Savanna Elephant"
    scientific_name = "Loxodonta africana"
    calories_needed = 8000

    def __init__(self, name, age=0):
        self.name = name
        self.age = age
        self.calories_eaten = 0
        self.happiness = 0

    def play(self, num_hours):
        self.happiness += (num_hours * 4)
        print("WHEEE PLAY TIME!")

    def eat(self, food):
        self.calories_eaten += food.calories
        print(f"Om nom nom yummy {food.name}"

        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
            print("Ugh so full")

    def interact_with(self, animal2):
        self.happiness += 1
        print(f"Yay happy fun time with {animal2.name}"

How would we use that class?
```
An Elephant class

class Elephant:
    species_name = "African Savanna Elephant"
    scientific_name = "Loxodonta africana"
    calories_needed = 8000

    def __init__(self, name, age=0):
        self.name = name
        self.age = age
        self.calories_eaten = 0
        self.happiness = 0

    def play(self, num_hours):
        self.happiness += (num_hours * 4)
        print("WHEEE PLAY TIME!")

    def eat(self, food):
        self.calories_eaten += food.calories
        print(f"Om nom nom yummy {food.name}")
        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
            print("Ugh so full")

    def interact_with(self, animal2):
        self.happiness += 1
        print(f"Yay happy fun time with {animal2.name}")

el1 = Elephant("Willaby", 5)
el2 = Elephant("Wallaby", 3)
el1.play(2)
el1.interact_with(el2)
A Rabbit class

class Rabbit:
    species_name = "European rabbit"
    scientific_name = "Oryctolagus cuniculus"
    calories_needed = 200

    def __init__(self, name, age=0):
        self.name = name
        self.age = age
        self.calories_eaten = 0
        self.happiness = 0

    def play(self, num_hours):
        self.happiness += (num_hours * 10)
        print("WHEEE PLAY TIME!")

    def eat(self, food):
        self.calories_eaten += food.calories
        print(f"Om nom nom yummy {food.name}"
        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
            print("Ugh so full")

    def interact_with(self, animal2):
        self.happiness += 4
        print(f"Yay happy fun time with {animal2.name}")

How would we use that class?
A Rabbit class

```python
class Rabbit:
    species_name = "European rabbit"
    scientific_name = "Oryctolagus cuniculus"
    calories_needed = 200

    def __init__(self, name, age=0):
        self.name = name
        self.age = age
        self.calories_eaten = 0
        self.happiness = 0

    def play(self, num_hours):
        self.happiness += (num_hours * 10)
        print("WHEEE PLAY TIME!")

    def eat(self, food):
        self.calories_eaten += food.calories
        print(f"Om nom nom yummy {food.name}")
        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
            print("Ugh so full")

    def interact_with(self, animal2):
        self.happiness += 4
        print(f"Yay happy fun time with {animal2.name}")

rabbit1 = Rabbit("Mister Wabbit", 3)
rabbit2 = Rabbit("Bugs Bunny", 2)
rabbit1.eat(broccoli)
rabbit2.interact_with(rabbit1)
```

How would we use that class?
Notice similarities?

<table>
<thead>
<tr>
<th>Elephant</th>
<th>Rabbit</th>
</tr>
</thead>
<tbody>
<tr>
<td># Class variables</td>
<td># Class variables</td>
</tr>
<tr>
<td>species_name</td>
<td>species_name</td>
</tr>
<tr>
<td>scientific_name</td>
<td>scientific_name</td>
</tr>
<tr>
<td>calories_needed</td>
<td>calories_needed</td>
</tr>
<tr>
<td># Instance variables</td>
<td># Instance variables</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>age</td>
<td>age</td>
</tr>
<tr>
<td>happiness</td>
<td>happiness</td>
</tr>
<tr>
<td># Methods</td>
<td># Methods</td>
</tr>
<tr>
<td>eat(food)</td>
<td>eat(food)</td>
</tr>
<tr>
<td>play()</td>
<td>play()</td>
</tr>
<tr>
<td>interact_with(other)</td>
<td>interact_with(other)</td>
</tr>
</tbody>
</table>

**Elephant** and **Rabbit** are both animals, so they have similar attributes. Instead of repeating code, we can *inherit* the code.
Base classes and subclasses

When multiple classes share similar attributes, you can reduce redundant code by defining a base class and then subclasses can inherit from the base class.

Tip: The base class is also known as the superclass.
The base class

The base class contains method headers common to the subclasses, and code that is used by multiple subclasses.

class Animal:
    species_name = "Animal"
    scientific_name = "Animalia"
    play_multiplier = 2
    interact_increment = 1

    def __init__(self, name, age=0):
        self.name = name
        self.age = age
        self.calories_eaten = 0
        self.happiness = 0

    def play(self, num_hours):
        self.happiness += (num_hours * self.play_multiplier)
        print("WHEEE PLAY TIME!")

    def eat(self, food):
        self.calories_eaten += food.calories
        print(f"Om nom nom yummy {food.name}")
        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
            print("Ugh so full")

    def interact_with(self, animal2):
        self.happiness += self.interact_increment
        print(f"Yay happy fun time with {animal2.name}")
The subclasses

To declare a subclass, put parentheses after the class name and specify the base class in the parentheses:

```python
class Panda(Animal):
```

Then the subclasses only need the code that's unique to them. They can redefine any aspect: class variables, method definitions, or constructor. A redefinition is called **overriding**.

The simplest subclass overrides nothing:

```python
class AmorphousBlob(Animal):
    pass
```
Overriding class variables

Subclasses can override existing class variables and assign new class variables:

class Rabbit(Animal):
    species_name = "European rabbit"
    scientific_name = "Oryctolagus cuniculus"
    calories_needed = 200
    play_multiplier = 8
    interact_increment = 4
    num_in_litter = 12

class Elephant(Animal):
    species_name = "African Savanna Elephant"
    scientific_name = "Loxodonta africana"
    calories_needed = 8000
    play_multiplier = 4
    interact_increment = 2
    num_tusks = 2
Overriding methods

If a subclass overrides a method, Python will use that definition instead of the superclass definition.

```python
class Panda(Animal):
    species_name = "Giant Panda"
    scientific_name = "Ailuropoda melanoleuca"
    calories_needed = 6000

    def interact_with(self, other):
        print(f"I'm a Panda, I'm solitary, go away {other.name}!")
```

How would we call that method?
Overriding methods

If a subclass overrides a method, Python will use that definition instead of the superclass definition.

```python
class Panda(Animal):
    species_name = "Giant Panda"
    scientific_name = "Ailuropoda melanoleuca"
    calories_needed = 6000

    def interact_with(self, other):
        print(f"I'm a Panda, I'm solitary, go away {other.name}!")
```

How would we call that method?

```python
panda1 = Panda("Pandeybear", 6)
panda2 = Panda("Spot", 3)
panda1.interact_with(panda2)
```
Using methods from the base class

To refer to a superclass method, we can use `super()`:

```python
class Lion(Animal):
    species_name = "Lion"
    scientific_name = "Panthera"
    calories_needed = 3000

    def eat(self, food):
        if food.type == "meat":
            super().eat(food)
```

How would we call that method?
Using methods from the base class

To refer to a superclass method, we can use `super()`:

class Lion(Animal):
    species_name = "Lion"
    scientific_name = "Panthera"
    calories_needed = 3000

    def eat(self, food):
        if food.type == "meat":
            super().eat(food)

How would we call that method?

bones = Food("Bones", "meat")
mufasa = Lion("Mufasa", 10)
mufasa.eat(bones)
super().attribute refers to the definition of attribute in the superclass of the first parameter to the method.

```python
def eat(self, food):
    if food.type == "meat":
        super().eat(food)
```

...is the same as:

```python
def eat(self, food):
    if food.type == "meat":
        Animal.eat(self, food)
```

super() is better style than BaseClassName, though slightly slower.
Overriding __init__

Similarly, we need to explicitly call `super().__init__()` if we want to call the `__init__` functionality of the base class.

```python
class Elephant(Animal):
    species_name = "Elephant"
    scientific_name = "Loxodonta"
    calories_needed = 8000

    def __init__(self, name, age=0):
        super().__init__(name, age)
        if age < 1:
            self.calories_needed = 1000
        elif age < 5:
            self.calories_needed = 3000

elly = Elephant("Ellie", 3)
elly.calories_needed
```
Overriding `__init__`

Similarly, we need to explicitly call `super().__init__()` if we want to call the `__init__` functionality of the base class.

```python
class Elephant(Animal):
    species_name = "Elephant"
    scientific_name = "Loxodonta"
    calories_needed = 8000

    def __init__(self, name, age=0):
        super().__init__(name, age)
        if age < 1:
            self.calories_needed = 1000
        elif age < 5:
            self.calories_needed = 3000

elly = Elephant("Ellie", 3)
elly.calories_needed  # 3000
```
Layers of inheritance
Object base class

Every Python 3 class implicitly extends the `object` class.
Adding layers of inheritance

But we can also add in more levels ourselves.
Adding layers of inheritance

First we define the new classes:

```python
class Herbivore(Animal):
    def eat(self, food):
        if food.type == "meat":
            self.happiness -= 5
        else:
            super().eat(food)

class Carnivore(Animal):
    def eat(self, food):
        if food.type == "meat":
            super().eat(food)
```

Then we change the base classes for the subclasses:

```python
class Rabbit(Herbivore):
class Panda(Herbivore):
class Elephant(Herbivore):
class Vulture(Carnivore):
class Lion(Carnivore):```
Multiple inheritance
Multiple inheritance

A class may inherit from multiple base classes in Python.
The new base classes

First we define the new base classes:

class Predator(Animal):
    def interact_with(self, other):
        if other.type == "meat":
            self.eat(other)
            print("om nom nom, I'm a predator")
        else:
            super().interact_with(other)

class Prey(Animal):
    type = "meat"
    calories = 200
Inheriting from multiple base classes

Then we inherit from them by putting both names in the parentheses:

```python
class Rabbit(Prey, Herbivore):
class Lion(Predator, Carnivore):
```

Python can find the attributes in any of the base classes:

```python
>>> r = Rabbit("Peter", 4)
>>> r.play()
>>> r.type
>>> r.eat(Food("carrot", "veggies"))
>>> l = Lion("Scar", 12)
>>> l.eat(Food("zazu", "meat"))
>>> l.encounter(r)
```
Inheriting from multiple base classes

Then we inherit from them by putting both names in the parentheses:

```python
class Rabbit(Prey, Herbivore):
class Lion(Predator, Carnivore):
```

Python can find the attributes in any of the base classes:

```python
>>> r = Rabbit("Peter", 4)  # Animal __init__
>>> r.play()  # Animal method
>>> r.type  # Prey class variable
>>> r.eat(Food("carrot", "veggies"))  # Herbivore method
>>> l = Lion("Scar", 12)  # Animal __init__
>>> l.eat(Food("zazu", "meat"))  # Carnivore method
>>> l.encounter(r)  # Predator method
```
Refresher: Identity
Checking identity

\texttt{exp0 \textbf{is} expl} evaluates to \texttt{True} if both \texttt{exp0} and \texttt{expl} evaluate to the same object.

\begin{Verbatim}
mufasa = Lion("Mufasa", 15)
nala = Lion("Nala", 16)

mufasa \textbf{is} mufasa
mufasa \textbf{is} Nala
mufasa \textbf{is not} Nala
nala \textbf{is not} None
\end{Verbatim}
Checking identity

**exp0** is **exp1**

evaluates to **True** if both **exp0** and **exp1** evaluate to the same object

mufasa = Lion("Mufasa", 15)
nala = Lion("Nala", 16)

mufasa is mufasa    # True
mufasa is Nala      # False
mufasa is not Nala  # True
nala is not None    # True
Composition
Composition

An object can contain references to objects of other classes.

What examples of composition are in an animal conservatory?

- An animal has a mate.
- An animal has a mother.
- An animal has children.
- A conservatory has animals.
Referencing other instances

An instance variable can refer to another instance:

class Animal:
    
    def mate_with(self, other):
        if other is not self and other.species_name == self.species_name:
            self.mate = other
            other.mate = self

How would we call that method?
Referencing other instances

An instance variable can refer to another instance:

class Animal:
    
def mate_with(self, other):
        if other is not self and other.species_name == self.species_name:
            self.mate = other
            other.mate = self

How would we call that method?

mr_wabbit = Rabbit("Mister Wabbit", 3)
jane_doe = Rabbit("Jane Doe", 2)
mr_wabbit.mate_with(jane_doe)
Referencing a list of instances

An instance variable can also refer to a list of instances:

class Rabbit(Animal):

    def reproduce_like_rabbits(self):
        if self.mate is None:
            print("oh no! better go on ZoOkCupid")
        return
        self.babies = []
        for _ in range(0, self.num_in_litter):
            self.babies.append(Rabbit("bunny", 0))

How would we call that function?
Referencing a list of instances

An instance variable can also refer to a list of instances:

class Rabbit(Animal):
    
    def reproduce_like_rabbits(self):
        if self.mate is None:
            print("oh no! better go on ZoOkCupid")
            return
        self.babies = []
        for _ in range(0, self.num_in_litter):
            self.babies.append(Rabbit("bunny", 0))

mr_wabbit = Rabbit("Mister Wabbit", 3)
jane_doe = Rabbit("Jane Doe", 2)
mr_wabbit.mate_with(jane_doe)
jane_doe.reproduce_like_rabbits()
Relying on a common interface

If all instances implement a method with the same function signature, a program can rely on that method across instances of different subclasses.

```python
def partytime(animals):
    """Assuming ANIMALS is a list of Animals, cause each to interact with all the others exactly once.""
    for i in range(len(animals)):
        for j in range(i + 1, len(animals)):
            animals[i].interact_with(animals[j])
```

How would we call that function?
Relying on a common interface

If all instances implement a method with the same function signature, a program can rely on that method across instances of different subclasses.

def partytime(animals):
    """Assuming ANIMALS is a list of Animals, cause each
to interact with all the others exactly once.""
    for i in range(len(animals)):
        for j in range(i + 1, len(animals)):
            animals[i].interact_with(animals[j])

How would we call that function?

jane_doe = Rabbit("Jane Doe", 2)
scar = Lion("Scar", 12)
elly = Elephant("Elly", 5)
pandy = Panda("PandeyBear", 4)
partytime([jane_doe, scar, elly, pandy])
Composition vs. Inheritance

Inheritance is best for representing "is-a" relationships

- Rabbit is a specific type of Animal
- So, Rabbit inherits from Animal

Composition is best for representing "has-a" relationships

- A conservatory has a collection of animals it cares for
- So, a conservatory has a list of animals as an instance variable
Quiz
What would Python print?

class Parent:
    def f(s):
        print("Parent.f")

    def g(s):
        s.f()

class Child(Parent):
    def f(me):
        print("Child.f")

a_child = Child()
a_child.g()