Inheritance
Class outline:

- Motivation
- Inheritance
- Multiple Inheritance
- Identity
- Class methods
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:

```python
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

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

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)
Elephant and Rabbit are both animals, so they have similar attributes. Instead of repeating code, we can inherit the code.
Inheritance
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}"

12
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:

```python
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
```
class LearnableContent:
    """A base class for specific kinds of learnable content. All kinds have title and author attributes, but each kind may have additional attributes. """
    license = "Creative Commons"

    def __init__(self, title, author):
        self.title = title
        self.author = author

# Create a Video subclass with
# license of "YouTube Standard License"

# Create an Article subclass with
# license of "CC-BY-NC-SA"

# Create a new Video instance with a title of "DNA" and an author of "Megan"
# Create a new Article instance with a title of "Water phases" and an author of
Exercise: LearnableContent (solution)

class LearnableContent:
    """A base class for specific kinds of learnable content.  
    All kinds have title and author attributes,  
    but each kind may have additional attributes.  
    """
    license = "Creative Commons"

    def __init__(self, title, author):
        self.title = title
        self.author = author

# Create a Video subclass with license of "YouTube Standard License"
class Video(LearnableContent):
    license = "YouTube Standard License"

# Create an Article subclass with license of "CC-BY-NC-SA"
class Article(LearnableContent):
    license = "CC-BY-NC-SA"

# Create a new Video instance with a title of "DNA" and an author of "Megan"
dna_video = Video("DNA", "Megan")

# Create a new Article instance with a title of "Water phases" and an author of "Lauren"
water_article = Article("Water phases", "Lauren")
Overriding methods

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

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.

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?

panda1 = Panda("Pandeybear", 6)
panda2 = Panda("Spot", 3)
panda1.interact_with(panda2)
class Clothing:
    ""
    >>> blue_shirt = Clothing("shirt", "blue")
    >>> blue_shirt.category
    'shirt'
    >>> blue_shirt.color
    'blue'
    >>> blue_shirt.is_clean
    True
    >>> blue_shirt.wear()
    >>> blue_shirt.is_clean
    False
    >>> blue_shirt.clean()
    >>> blue_shirt.is_clean
    True
    ""

def __init__(self, category, color):
    self.category = category
    self.color = color
    self.is_clean = True

def wear(self):
    self.is_clean = False

def clean(self):
    self.is_clean = True

class KidsClothing(Clothing):
    """
>>> onesie = KidsClothing("onesie", "polka dots")
>>> onesie.wear()
False
>>> onesie.is_clean
False
>>> onesie.clean()
>>> onesie.is_clean
False

>>> dress = KidsClothing("dress", "rainbow")
>>> dress.clean()
>>> dress.is_clean
True
>>> dress.wear()
>>> dress.is_clean
False
>>> dress.clean()
>>> dress.is_clean
False

"""

# Override the clean() method
# so that kids clothing always stays dirty!
Exercise: Clothing (solution)

class Clothing:
    """
    >>> blue_shirt = Clothing("shirt", "blue")
    >>> blue_shirt.category
    'shirt'
    >>> blue_shirt.color
    'blue'
    >>> blue_shirt.is_clean
    True
    >>> blue_shirt.wear()
    >>> blue_shirt.is_clean
    False
    >>> blue_shirt.clean()
    >>> blue_shirt.is_clean
    True
    """

def __init__(self, category, color):
    self.category = category
    self.color = color
    self.is_clean = True

def wear(self):
    self.is_clean = False

def clean(self):
    self.is_clean = True

class KidsClothing(Clothing):
    """
```python
>>> onesie = KidsClothing("onesie", "polka dots")
>>> onesie.wear()
>>> onesie.is_clean
False
>>> onesie.clean()
>>> onesie.is_clean
False
>>> dress = KidsClothing("dress", "rainbow")
>>> dress.clean()
>>> dress.is_clean
True
>>> dress.wear()
>>> dress.is_clean
False
>>> dress.clean()
>>> dress.is_clean
False

# Override the clean() method
# so that kids clothing always stays dirty!
def clean(self):
    self.is_clean = self.is_clean
```
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?
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)
More on `super()`

`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.

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

What would this display?

```python
elly = Elephant("Ellie", 3)
elly.calories_needed  # 3000
```
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}")

class Cat(Animal):
    """
```python
>>> adult = Cat("Winston", 12)
>>> adult.name
'Winston'
>>> adult.age
12
>>> adult.play_multiplier
3

>>> kitty = Cat("Kurty", 0.5)
>>> kitty.name
'Kurty'
>>> kitty.age
0.5
>>> kitty.play_multiplier
6

"""

species_name = "Domestic cat"
scientific_name = "Felis silvestris catus"
calories_needed = 200
play_multiplier = 3

def __init__(self, name, age):
    
    # Call the super class to set name and age
    # If age is less than 1, set play multiplier to 6
```

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}")

class Cat(Animal):
    ...
'adult' = Cat("Winston", 12)
>>> adult.name
'Winston'
>>> adult.age
12
>>> adult.play_multiplier
3
>>> kitty = Cat("Kurty", 0.5)
>>> kitty.name
'Kurty'
>>> kitty.age
0.5
>>> kitty.play_multiplier
6

""

species_name = "Domestic cat"
scientific_name = "Felis silvestris catus"
calories_needed = 200
play_multiplier = 3

def __init__(self, name, age):
    super().__init__(name, age)
    if self.age < 1:
        self.play_multiplier = 6
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:

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:

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:

```python
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)
>>> r.play()
>>> r.type
>>> r.eat(Food("carrot", "veggies"))
>>> l = Lion("Scar", 12)
>>> l.eat(Food("zazu", "meat"))
>>> l.encounter(r)
```
Identity
Checking identity

**exp0** is **exp1**

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

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

mufasa is mufasa
mufasa is Nala
mufasa is not Nala
nala is not None
```
Checking identity

`exp0 is exp1` evaluates to `True` if both `exp0` and `exp1` evaluate to the same object.

```python
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
```
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()

Find out in PythonTutor
Class Methods
The @classmethod decorator

By default, a function definition inside a class is a bound method that receives an instance of that class.

To instead make a function that receives the class itself, use the @classmethod decorator.

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

    @classmethod
def rabbit_twins(cls, name1, name2):
        rabbit1 = cls(name1)
        rabbit2 = cls(name2)
        rabbit1.interact_with(rabbit2)
        return [rabbit1, rabbit2]

twinsies = Rabbit.rabbit_twins("Fluffy", "Hoppy")