Lecture 11: Inheritance, String Representation

July 7, 2022

Richard Roggenkemper

Announcements

- Midterm is next Thursday at 6pm!
  - Midterm Logistics post will be out later today!
- CATS has been released!
  - Partner project if you want
  - Recommendation is to get Phase 1 done as soon as you can then work on Phase 2 after the midterm
- HW 2, Lab 3 are due tonight!

OOP Continued

Looking Up Attributes

<expression>.<name>

How do you evaluate a dot expression?

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Demo

Attribute Assignment

Assignment statements with a dot expression on their left side affect attributes for the object of that dot expression

If the object is an **instance**, then the assignment sets an **instance attribute**

If the object is a **class**, then the assignment sets a **class attribute**

Inheritance

Motivation

We learned about creating classes, but sometimes two classes that we want to define are really similar

Instead of having repeated code, we could create one general class and then define two more specific classes that have their own special behavior, in addition to the generic behavior

This lets us only rewrite the attributes and methods we want to make different instead of rewriting everything

Elephant Class

```
class Elephant:
    scientific_name = “Loxodonta africana”
    calories_needed = 16000

    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(“pawoo”)

    def eat(self, food):
        self.calories_eaten += food.calories
        print(“nom nom nom yummy “ + food.name)
        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
            print(“too full, need nap”)
```

Rabbit Class

```
class 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(“kip kip”)

    def eat(self, food):
        self.calories_eaten += food.calories
        print(“nom nom nom yummy “ + food.name)
        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
            print(“too full, need nap”)
```

Elephant

Class Attributes: scientific_name, calories_needed

Instance Attributes: name, age, happiness, calories_needed

Methods: eat(food), play

Rabbit

Class Attributes: scientific_name, calories_needed

Instance Attributes: name, age, happiness, calories_needed

Methods: eat(food), play

Differences

The methods are slightly different for the classes

- The happiness multiplier is different
- They make different noises

Some attributes are also different

- The scientific name and calories needed are different and dependent on the animal

How do we make this better?

We can use **inheritance** to make a general Animal class and then have the specific animals inherit from that class!

The Animal class can have a generic animal definition while the specific classes will add the specific elements to it

In this example, Animal will be the **superclass** and Elephant, Rabbit, Panda, Dog, etc. will be the **subclasses**

Animal Class

```
class Animal:
    scientific_name = “Animalia”
    calories_needed = 100
    play_multiplier = 1
    noise = “woo”

    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 * play_multiplier
        print(noise)

    def eat(self, food):
        self.calories_eaten += food.calories
        print(“nom nom nom yummy “ + food.name)
        if self.calories_eaten > self.calories_needed:
            self.happiness -= 1
```

Creating a Subclass

You can create the subclass by adding the parent class into the class definition

```
class Panda(Animal)
```

You should only write code in these subclasses that are unique to them

You can redefine class variables, methods, or the constructor which is called overriding

New Elephant and Rabbit

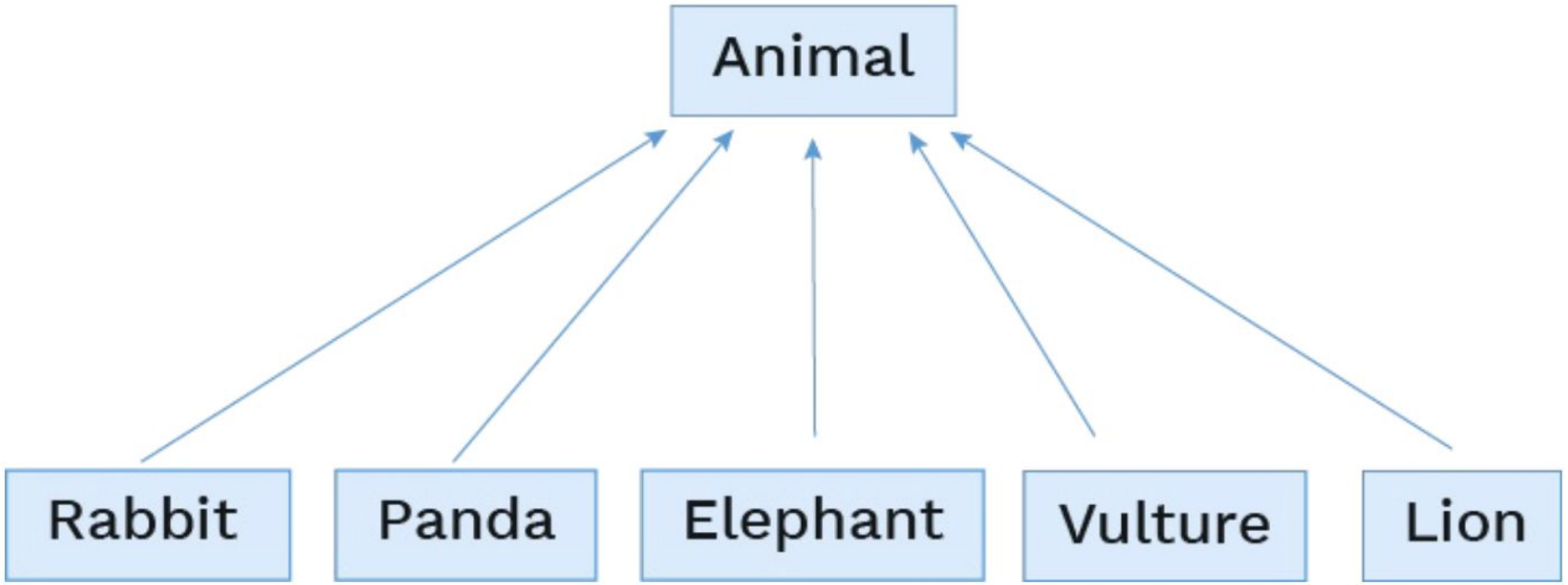
```
class Elephant(Animal):
    scientific_name = “Loxodonta africana”
    calories_needed = 16000
    play_multiplier = 4
    noise = “pawoo”

class Rabbit(Animal):
    scientific_name = “Oryctolagus cuniculus”
    calories_needed = 200
    play_multiplier = 10
    noise = “kip kip”
    hops = True
```

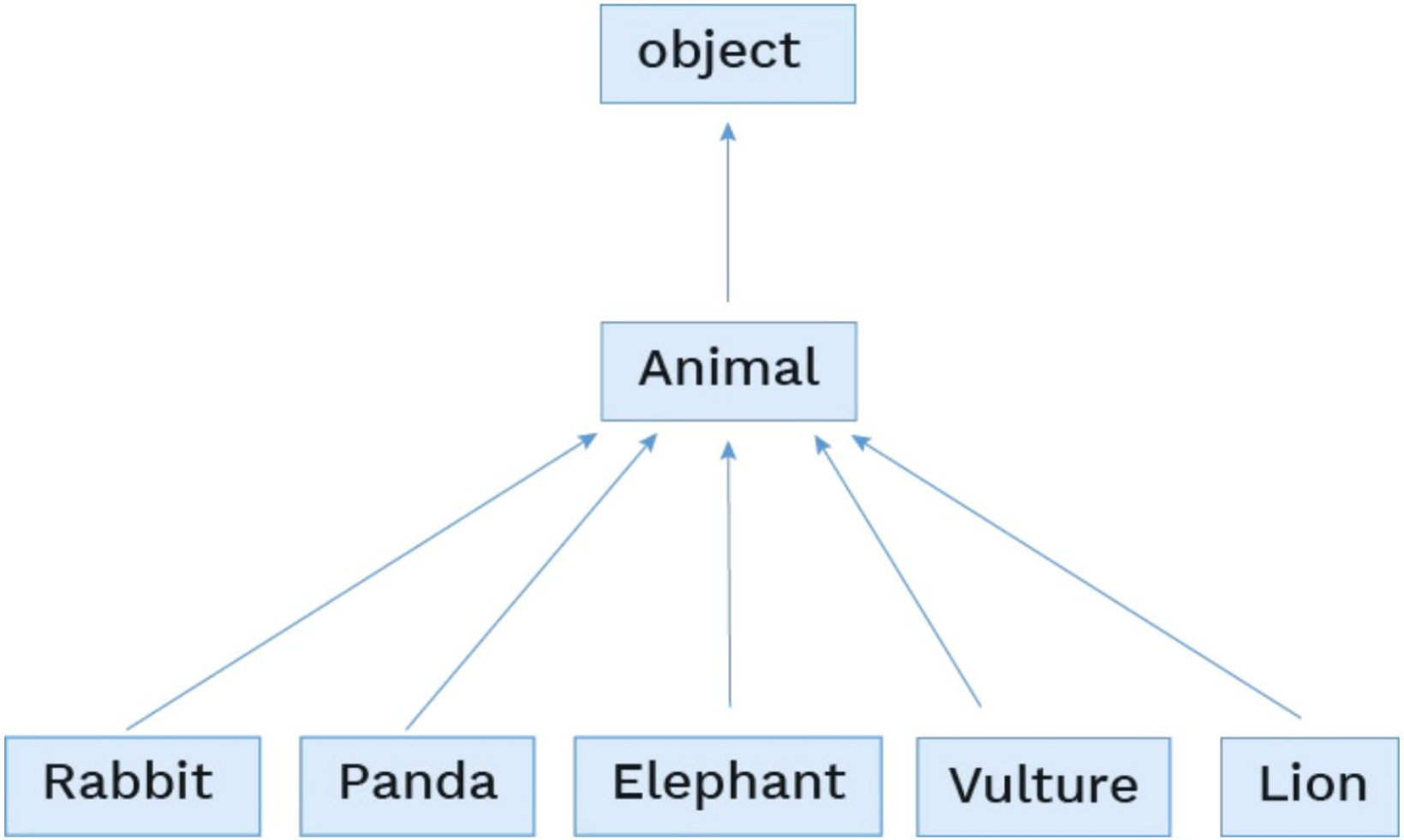
Overriding Methods

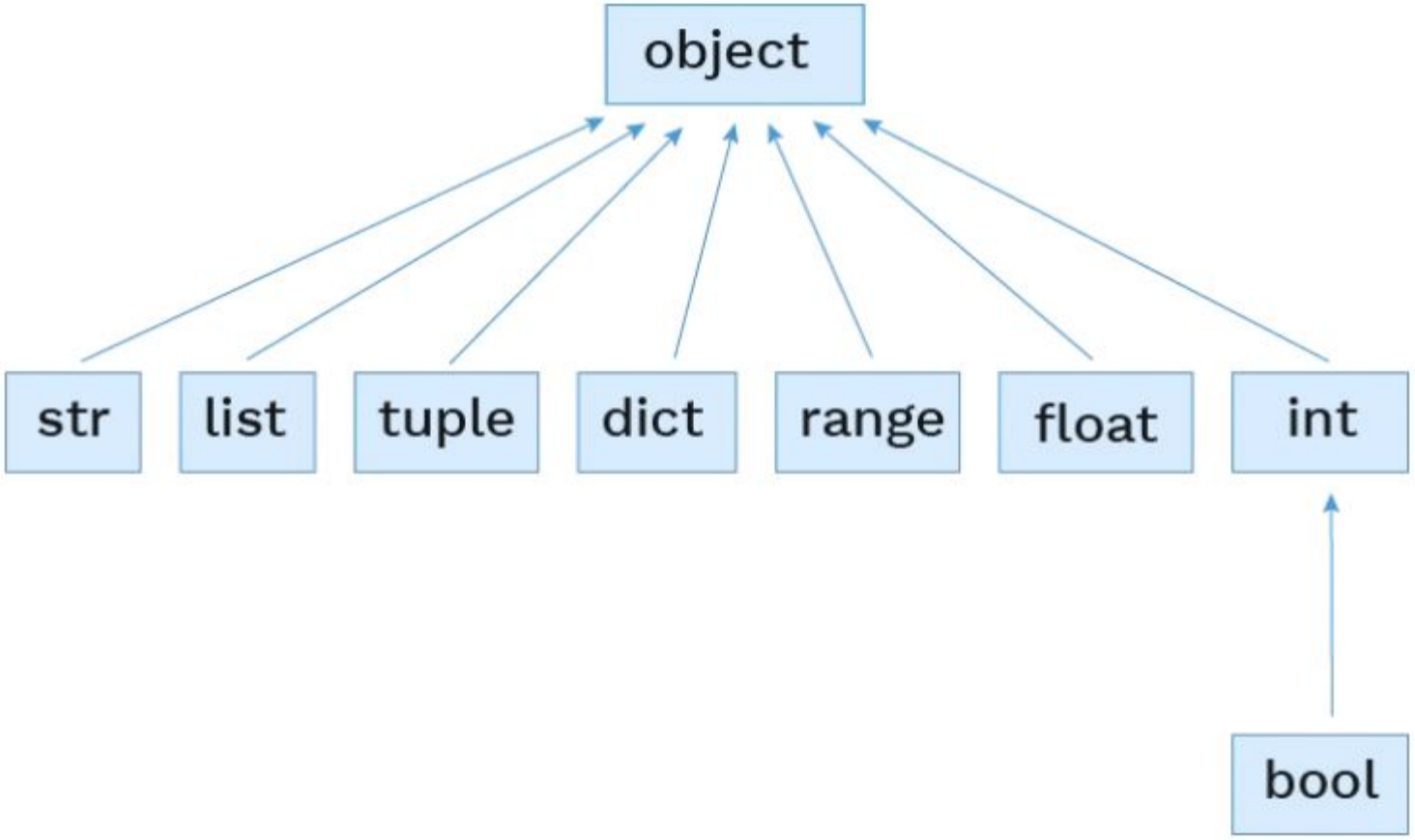
```
class Panda(Animal):
    scientific_name = “Ailuropoda melanoleuca”
    calories_needed = 6000
    play_multiplier = 5
    noise = “neeeeeh”

    def eat(self):
        self.calories_eaten += 150
        print(“nom nom bamboo is the best nom nom”)
        if self.calories_eaten > self.calories_needed:
            self.calories_eaten += 150
            self.happiness += 1
            print(“more bamboo, i love bamboo”)
```

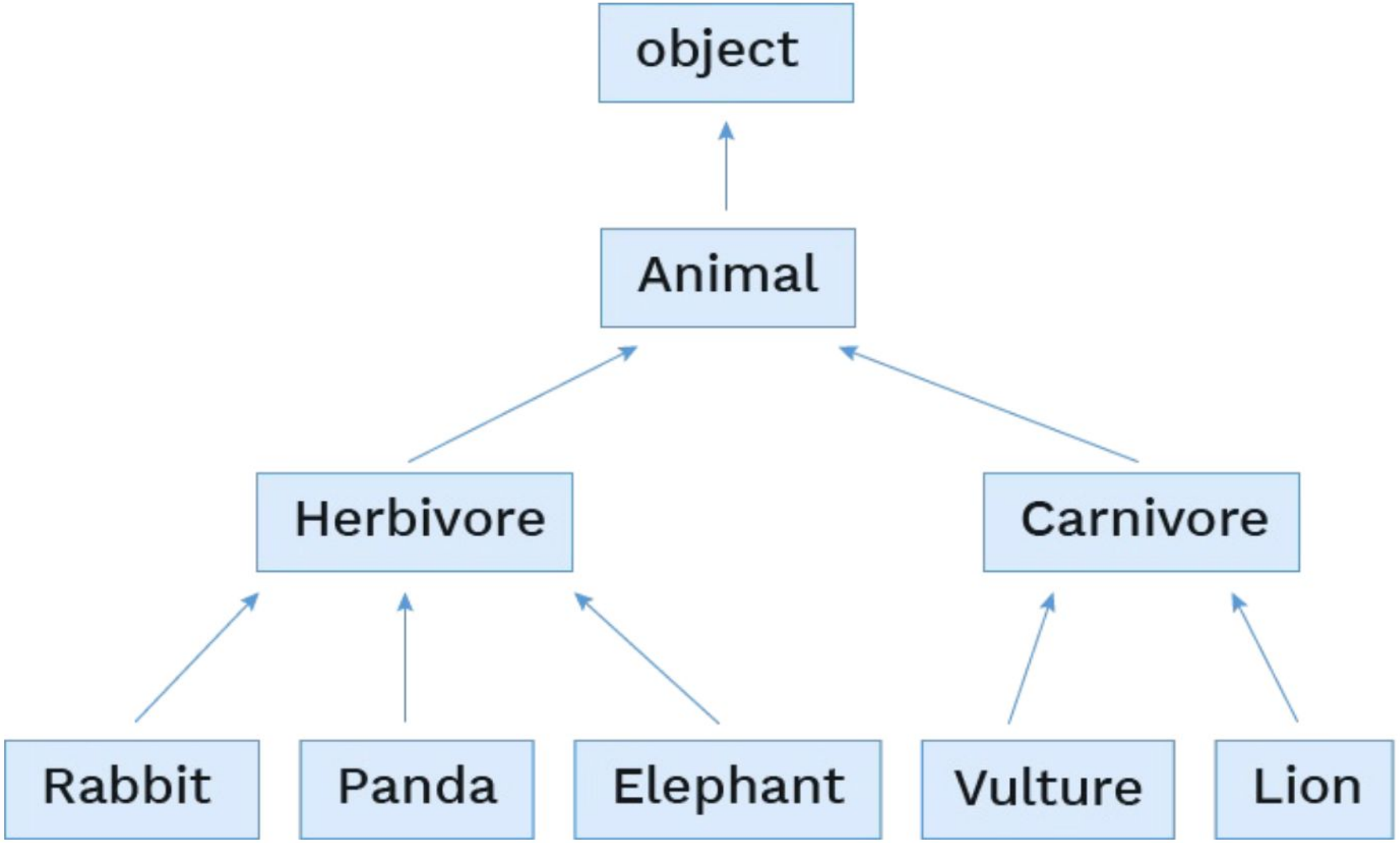
Using Methods from the Superclass

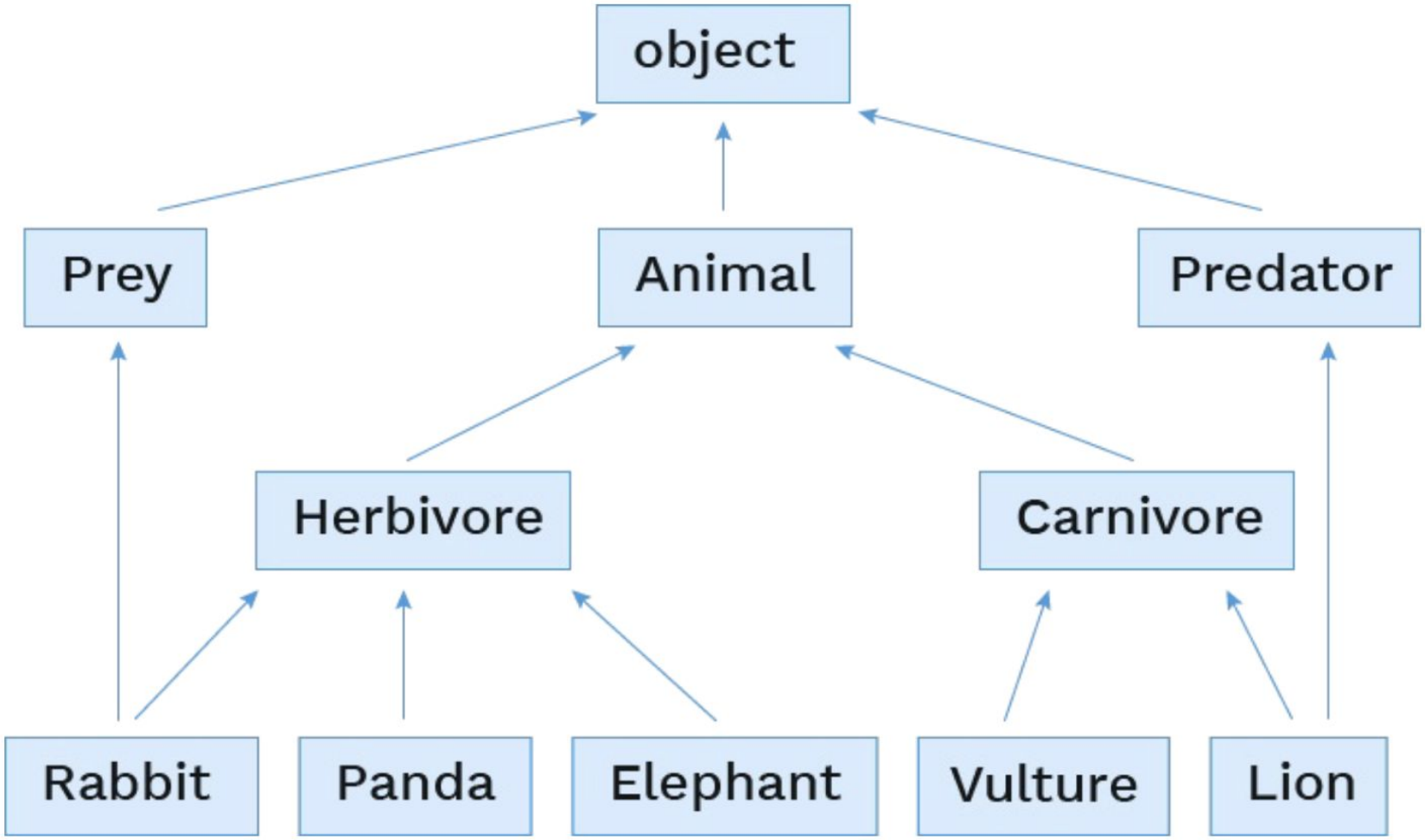
```
class Lion(Animal):

    scientific_name = “Panthera”
    calories_needed = 3000
    play_multiplier = 12
    noise = “rooooaaar”

    def eat(self):
        if food.type == “meat” :
            super().eat(food)          # Animal.eat(self, food)
        else:
            print(“no thank you”)
```


Animal
Rabbit
Panda
Elephant
Vulture
Lion


object
Animal
Rabbit
Panda
Elephant
Vulture
Lion


object
str
list
tuple
dict
range
float
int
bool


object
Animal
Herbivore
Carnivore
Rabbit
Panda
Elephant
Vulture
Lion


object
Prey
Animal
Predator
Herbivore
Carnivore
Rabbit
Panda
Elephant
Vulture
Lion

Demo

Looking up Attribute Names with Inheritance

The base class attributes are **not** copied into the subclasses

To look up a name with inheritance

1. If it names an attribute in the class, return the attribute value
2. Otherwise, look up the name in the super class, if it exists

Using Inheritance

With inheritance, we want to avoid repeated ourselves if possible, we can just use the existing implementation instead

Attributes that have been overridden are still accessible via the subclass

Look up attributes on instances when possible

Inheritance vs Composition

OOP is best when we think about the following metaphor

Inheritance is best for representing is-a relationships

- A panda is an animal so Panda inherits from Animal

Composition is best for representing has-a relationships

- A zoo has a collection of animals so the zoo has a list of animals as an attribute

Let’s Design some Objects

I want to be the very best, like no one ever was

To catch them is my real test, to train them is my cause

I will travel across the land, searching far and wide

Teach Pokemon to understand, the power that’s inside

Design Pokemon

- Let’s have a generic Pokemon class
  - Pokemon should have a name, trainer, level, HP (health points), and the attack tackle
  - They should be able to attack other Pokemon, receive damage, say their name
- We also want to have types (electric, water, fire, etc.)
  - Each class should have a specialized attack
- You can have multiple of each Pokemon
- A trainer should be able to have a team of Pokemon
  - There should be a way to add, remove Pokemon to this team with there being a limit of 6 Pokemon in a team
  - The trainer should have a name and age

Break

String Representation

String Representation

By default, objects in Python are ugly and unreadable

We have the ability to set our own representation for these objects to make them easier for the computer or humans to read

While there are best practices for this representation, we technically have the ability to do whatever we want with it

Demo

__str__

The **__str__** method returns a human readable string representation of an object

For example, instead of having fractions evaluate to the decimal representation, I want to keep it in fraction form

The **__str__** method is used in Python when you call **print()**, **str()**, **f-strings**, and more

Demo

__repr__

The **__repr__** method returns a computer readable string representation of an object

This usually means returning a string that would evaluate to an instance of the object with the same values

This method is used when **repr()** is called and when displaying an instance in an interactive Python session

If **str(obj)** is called but no __str__ exists, then we use the __repr__

Using String Representation

When you print an object, you want to look at the __str__ first, followed by the __repr__ if not found. This continues up the parent classes until you are unable to anymore.

In the end, you end up calling **print(str(obj))**

When evaluating the object in the terminal, you look for the __repr__, look at parents until you are unable to anymore.

When evaluating, you call **print(repr(obj))**

Summary

- **Inheritance** lets us inherit attributes and methods from parent classes which is useful when trying to cut down on repeated code
- Inheritance is best for **is-a relationships**
- Objects has a set of **lookup rules** for methods and attributes like we saw with environment diagrams
- **String Representation** gives us the ability to create more readable objects