继承

继承（Inheritance）

- 继承是一个关联多种类的技术
- 一个常见场景：两个类很相似，只有部分专有行为不同
  - 把两个类的所有内容都写一遍？
- 更好的做法：一个特定的类可以将这些共同点作为一个通用类，额外附加其特定专用的行为即可

Recall: Don't repeat yourself

继承（Inheritance）

```
class <Name>(<Base Class>):
      <suite>
```

- 创建一个类，名字为<Name>， 其是类<Base Class>的子类（subclass），类<Base Class> 也被称为该类的父类、基类或超类（Superclass）。
- 新创建的类继承了（即含有）<Base Class>的属性。
  - 子类可以在<suite>添加自己的专有属性
  - 子类也可以修改父类的属性 ⟶ 即重写（overriding）

⚠️与重载（overloading）不同

一个例子：动物喂养

```
Panda()
Lion()
Rabbit()
Hawk()
Elephant()
Food()
```

一个例子：动物喂养

```
class Food:

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

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

一个例子：动物喂养

```
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)
```

一个例子：动物喂养

```
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)
```

重复的点

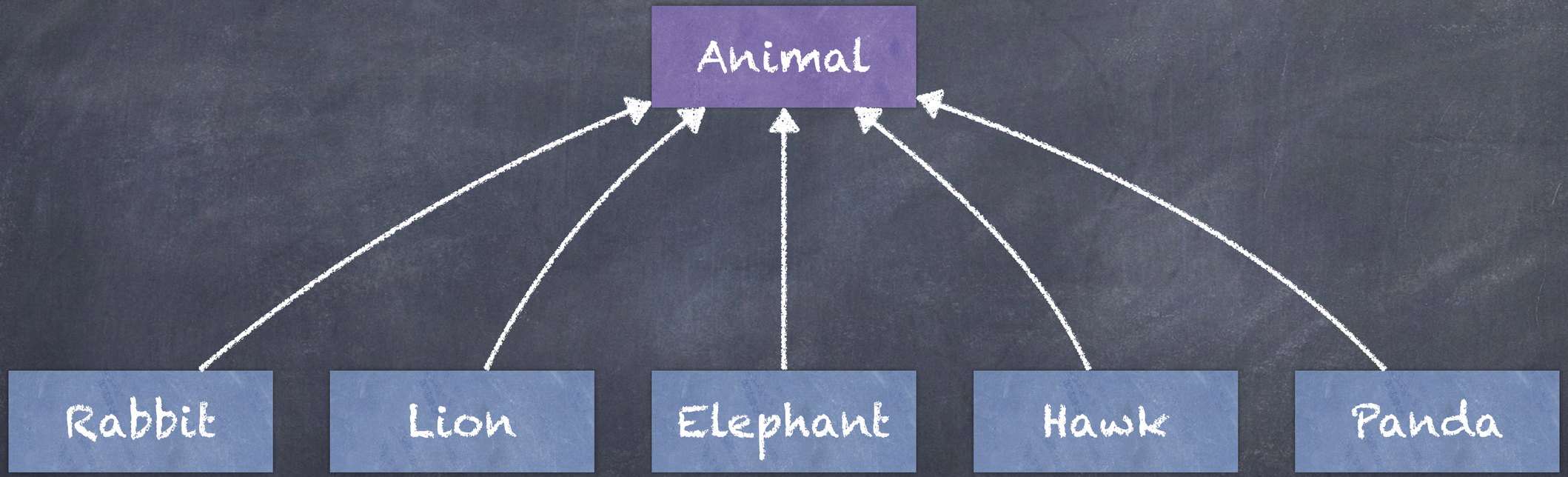
Elephant

```
# Class variables
species_name
scientific_name
calories_needed

# Instance variables
name
age
happiness

# Methods
eat(food)
play()
interact_with(other)
```

Rabbit

```
# Class variables
species_name
scientific_name
calories_needed

# Instance variables
name
age
happiness

# Methods
eat(food)
play()
interact_with(other)
```

设计父类



利用继承实现动物喂养例子

```
class Animal:
    species_name = "Animal"
    scientific_name = "Animalia"
    calories_needed = 100
    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}")
```

利用继承实现动物喂养例子

```
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
```

重写（Overriding）

- 重写变量：子类可以重新定义已在父类中的变量

```
class Animal:
    species_name = "Animal"
    calories_needed = 100
    scientific_name = "Animalia"
    play_multiplier = 2
    interact_increment = 1
```

Overriding variables

Overriding 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
```

Specific variables

重写

- 重写方法：子类可以重新定义已在父类中的方法

```
class Animal:
    species_name = "Animal"
    calories_needed = 100
    scientific_name = "Animalia"
    play_multiplier = 2
    interact_increment = 1

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

Overriding methods

```
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}!")

    def push_down_tree(self):
        print("I will push down the tree!")
```

Specific methods

```
panda1 = Panda("Pandeybear", 6)
panda2 = Panda("Spot", 3)
panda1.interact_with(panda2)
```

使用基类中的方法

- 可以使用super()方法来调用

```
class Animal:
    species_name = "Animal"
    calories_needed = 100
    scientific_name = "Animalia"
    play_multiplier = 2
    interact_increment = 1

    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”)

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

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

Invoke

```
bones = Food("Bones", "meat", 100)
mufasa = Lion("Mufasa", 10)
mufasa.eat(bones)
```

使用基类中的方法

- Super().method(…) 调用了父类中的方法method(…)，并将调用Super()处所在方法的第一个参数（一般是self）传入该方法method(…)作为第一个参数的实参。

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

is the same as:

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

Super() is better style than BaseClassName, though slightly slower.

重写 __init__

- 同样，如果想要调用基类的__init__方法，需要显式地调用super().__init__()

```
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
```

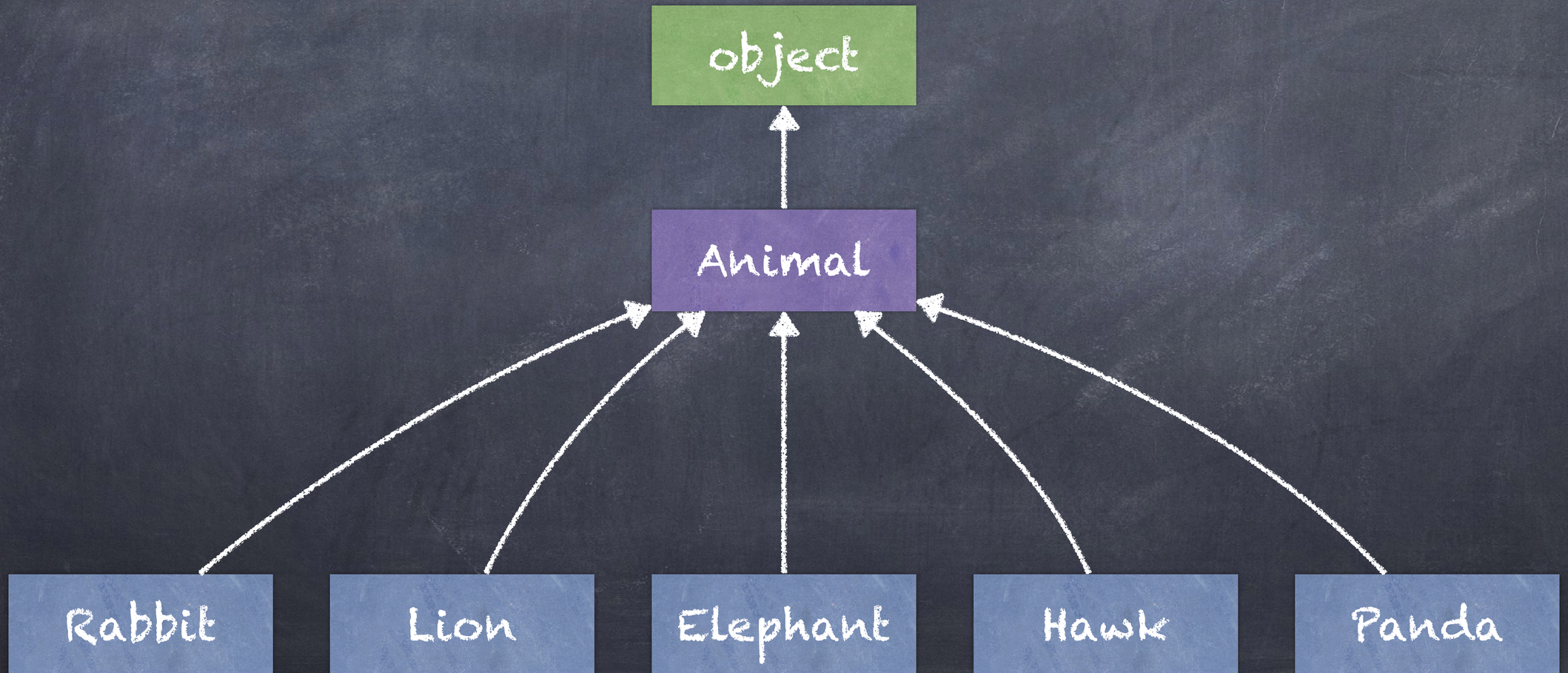
What would this display?

查找属性

- 基类的属性并不是拷贝进子类中去的！
- 要查找类中的某个名字：
  1. 如果该名字就是该类的某个属性，那么直接返回该属性的值
  2. 否则，在基类中的查找该名字

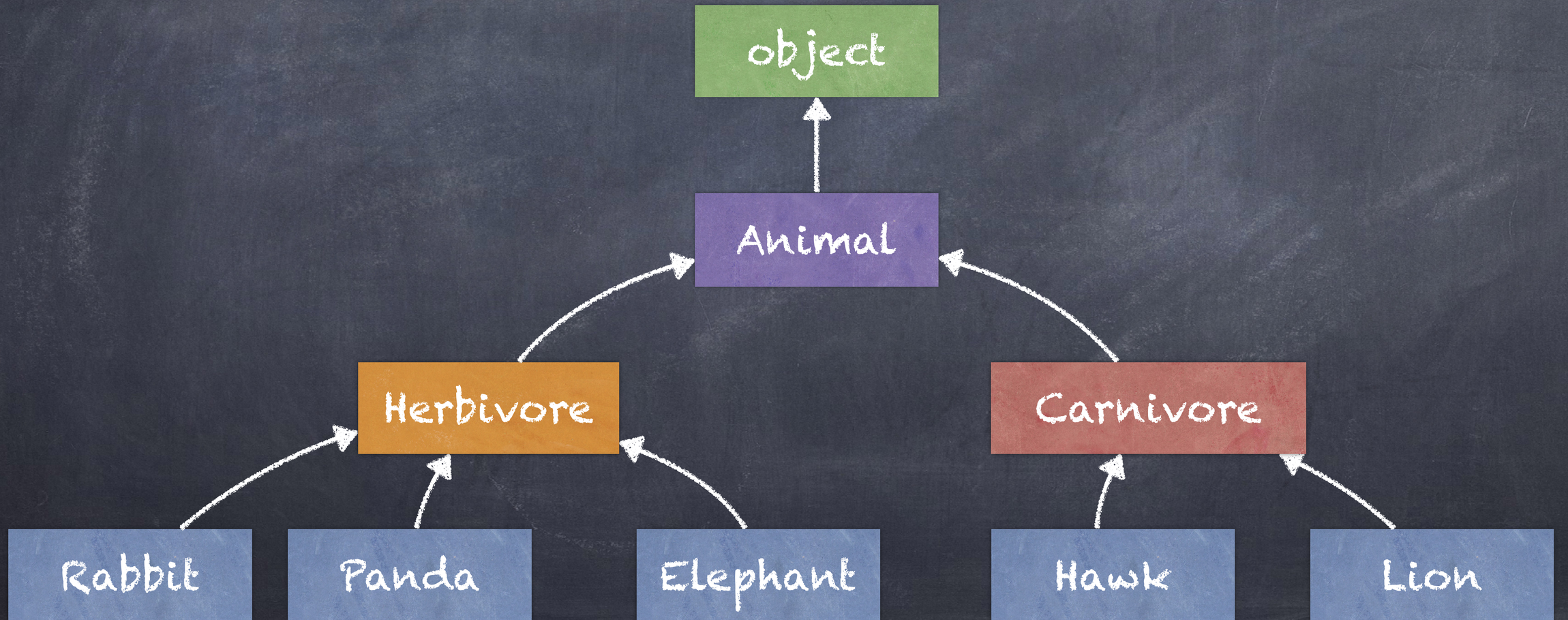
继承的层次性

- 每一个Python 3的类都隐式地继承了object类（都是object类的子类）



继承的层次性

- 当然，还可以增加更多继承层次



继承的层次性

- 当然，还可以增加更多继承层次

```
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)
```

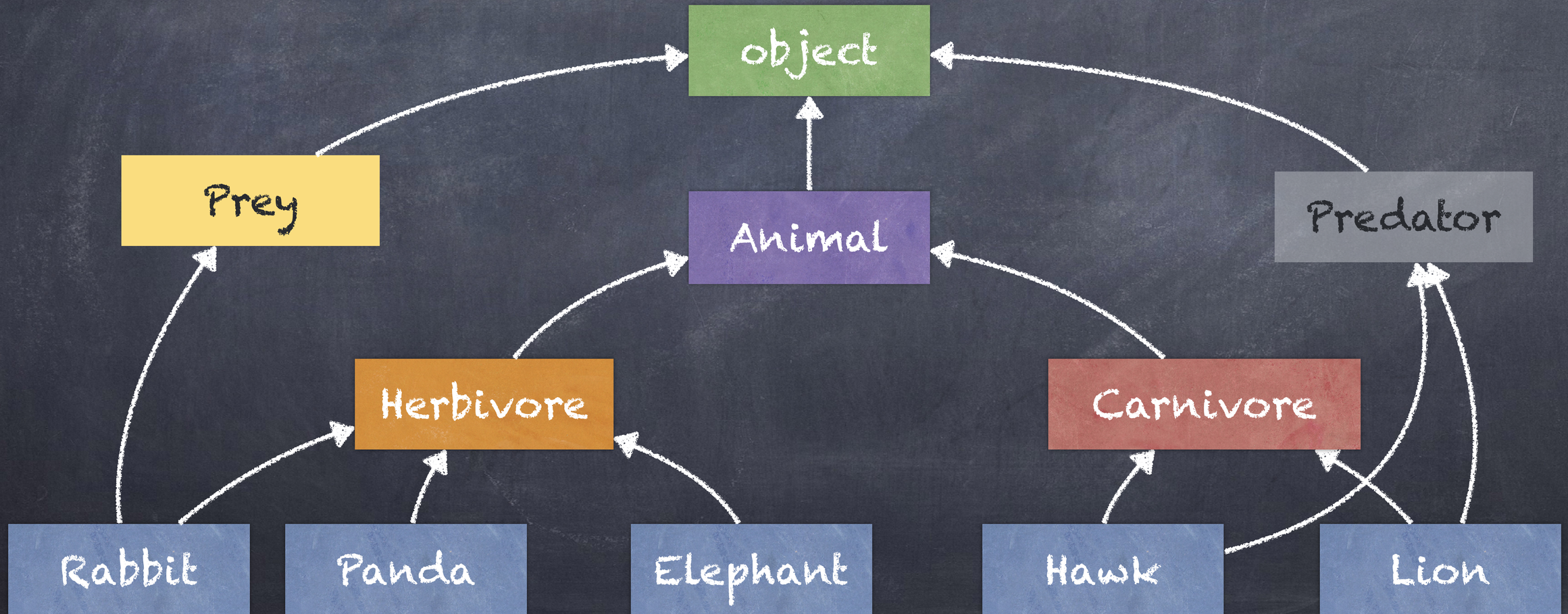
```
class Rabbit(Herbivore):
class Panda(Herbivore):
class Elephant(Herbivore):

class Hawk(Carnivore):
class Lion(Carnivore):
```

demo

多重继承（Multiple inheritance）

多重继承（Multiple inheritance）

- Python中可以继承多个基类



多重继承（Multiple inheritance）

```
class Predator(Animal):

    def encounter(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


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

多重继承（Multiple inheritance）

```
class Predator(Animal):

    def encounter(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
```

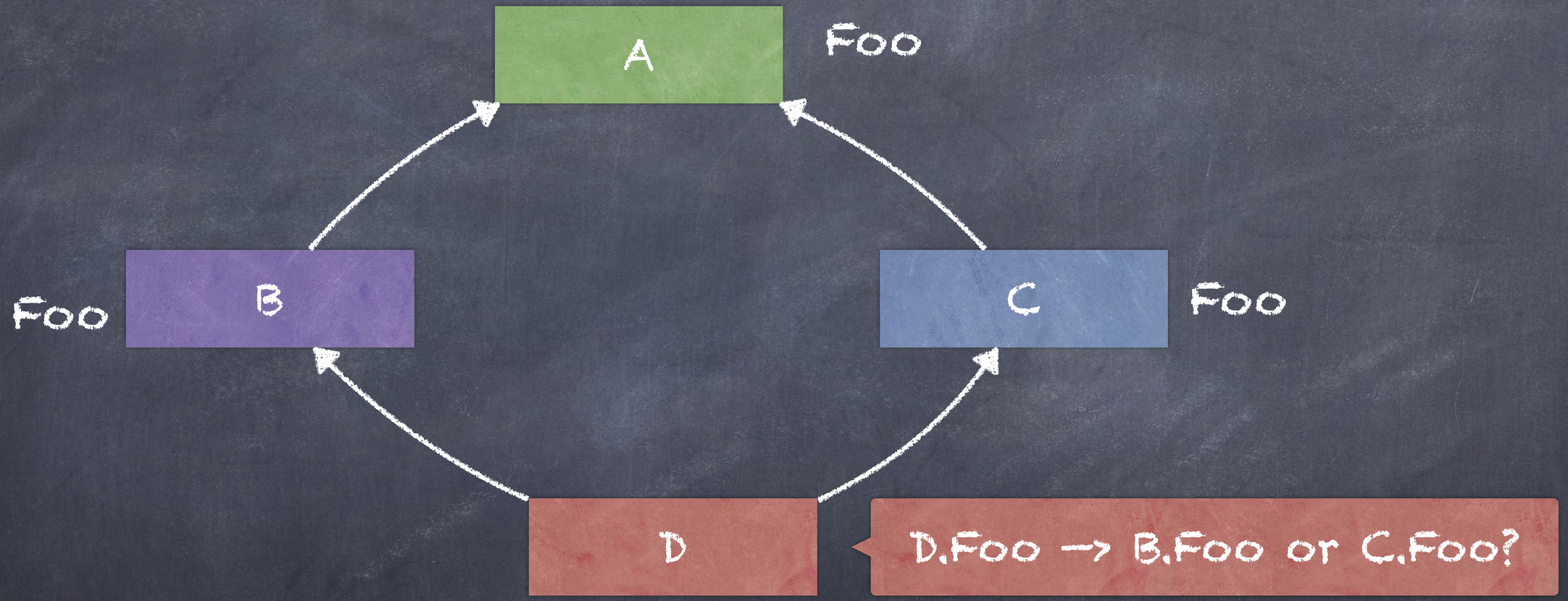
```
class Herbivore(Animal):

    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)
```

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

```
>>> r = Rabbit("Peter", 4)                    # Animal __init__
>>> r.play(5)                                 # Animal method
>>> r.type                                    # Prey class variable
>>> r.eat(Food("carrot", "veggies", 100))     # Herbivore method
>>> l = Lion("Scar", 12)                      # Animal __init__
>>> l.eat(Food("zazu", "meat", 1000))         # Carnivore method
>>> l.encounter(r)                            # Predator method
```

demo

菱形继承问题（Diamond Problem）



Method Resolution Order (MRO): 一般情况下：从左到右，从下到上（C3 linearization 算法）

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A monotonic superclass linearization for Dylan. (OOPSLA '96)

Is-a relationship

- 继承代表了一种包摄（Subsumption）关系：每一个子类都是其基类一种
- 任何子类的对象都是其基类的对象

```
class A:
    pass

class B(A):
    pass

isinstance(A(), A)
isinstance(B(), A)
issubclass(B, A)
```

Quiz:

```
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()
```

What would Python print?

Python tutor

继承的一些缺点

- 打破了封装性：继承强制对子类开发者知道其父类的内部信息
- 继承所带来的代价：需要存储超类的变量、构造子、方法、但可能只有少量的超类方法被用到

Inheritance helps code reuse but NOT for code reuse!

组合

组合（Composition）

- 一个对象可以拥有其他类的对象的引用
- 组合代表了一种“Has-a” relationship (possessive hierarchy)

引用其他实例

- 一个对象可以引用其他对象作为其实例变量之一

```
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

mr_wabbit = Rabbit("Mister Wabbit", 3)
jane_doe = Rabbit("Jane Doe", 2)
mr_wabbit.mate_with(jane_doe)
```

引用一批其他实例

- 一个对象可以引用一批其他对象

```
class Rabbit(Animal):

    def reproduce_like_rabbits(self):
        if self.mate is None:
            print("oh no! better date  someone")
            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()
```

引用其他类的实例

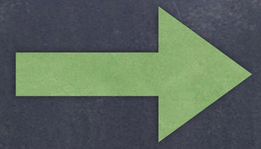
- 一个对象可以引用其他类的对象

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

jane_doe = Rabbit("Jane Doe", 2)
l = Lion("Scar", 12)
panda1 = Panda("Pandeybear", 6)
el1 = Elephant("Willaby", 5)
con = Conservatory([jane_doe, l, panda1, el1])
```

组合的缺点

- 使用组合时，当我们需要处理某些请求时，我们其实需要委托（delegation）某个类型的对象的方法去完成相应的请求。

```
class A:
  def spam(self, x):
    pass
  def foo(self):
    pass
```



```
class B:
  def __init__(self):
    self._a = A()
  def spam(self, x):
    # Delegate to the internal self._a instance
    return self._a.spam(x)
  def foo(self):
    # Delegate to the internal self._a instance
    return self._a.foo()
  def bar(self):
    pass
```

- 写法上没有继承的"直接复用"相应方法方便。

混入(Mixin or Mix-in)

Mixin

- 在python中，Mixin是一个类，其包含一些属性方法，但其本身不是为了实例化对象
- 它的存在是为了让其他类可以拥有这些方法（在python中通过继承来得到，但在其他语言中，不一定要作为父类，如Java接口的默认方法、scala的trait、Ruby的module）
- 每一个Mixin类应该提供一些紧密相关的方法，为了实现"一个"特有的行为
- 代表了"-able" relationship

例子

it is not intended for direction instantiation.

```
class MappingMixin:
    def __getitem__(self, key):
        return self.__dict__.get(key)

    def __setitem__(self, key, value):
        return self.__dict__.set(key, value)
```

A specific behavior

```
class Rabbit(MappingMixin,Animal):
    pass
```

Conventions: 1. Python doesn't define a formal way to define mixin classes, it's a good practice to name mixin classes with the suffix `Mixin`
2. `mixin` classess should be on the left to the based classes

Any questions ?