第7课　面向对象编程
Object-Oriented Programming

OOP

面向对象编程——Object Oriented Programming，简称OOP

一种重要的程序设计思想

•OOP把**对象作为程序的基本单元**，一个对象包含了数据和操作数据的函数。

•**面向过程的程序设计**把计算机程序视为一系列的命令集合，即一组函数的顺序执行。

•面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

•在Python中，所有数据类型都可以视为对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

面向过程

- 假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个字典表示：

```
std1 = { 'name': 'Michael', 'score': 'A' }
std2 = { 'name': 'Bob', 'score': 'C' }
```

- 打印学生的成绩：通过函数实现

```
def print_score(std):
        print('%s: %s' % (std['name'], std['score']))


print_score(std1) #Michael: A

print_score(std2) #Bob: C
```

面向对象

- 思考的不是程序的执行流程，而是对象（即学生）
- 学生们都有什么共同的属性（property）？
  - 字典里的姓名（name）和分数（score）都是学生的属性
- 打印学生的成绩：
  - 先创建学生对象
  - 再让对象打印成绩
  - 注：打印成绩可以让对象自行操作

```
class Student():
        def __init__(self, name, score):
                self.name = name
                self.score = score

        def print_score(self):
                print('%s: %s' % (self.name, self.score))
```

面向对象

类的名字

```
class Student:
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

定义新的类的关键字

类的属性

类的方法（函数）

类的内部实现

注意：

- 类名字后的冒号：
- 缩进！

使用：实例化对象

```
michael = Student('Michael','A')
bob = Student('Bob','C')
```

面向对象

```
class Student:
        def __init__(self, name, score):
```

使用：实例化对象

```
michael = Student('Michael','A')
bob = Student('Bob','C')
```

```
x = MyClass(args)
```

我们之前也用过：

```
float('3.5')      # => 3.5
str(8)            # => '8'
list('god')       # => ['g', 'o', 'd', ...]
set()             # => empty set
```

面向对象

```
class Student:
        def __init__(self, name, score):
                self.name = name
                self.score = score

        def print_score(self):
                print('%s: %s' % (self.name, self.score))
```

```
michael = Student('Michael','A')
bob = Student('Bob','C')
```

#类的方法：调用对象对应的关联函数

```
michael.print_score()#Michael: A
bob.print_score()     #Bob: C
```

#类的属性

```
print(michael.name) #Michael
print(michael.score) #A

print(bob.name)   #Bob
print(bob.score) #C
```

对象名.xxx

面向对象

```
class Student:
        def __init__(self, name, score):
                self.name = name
                self.score = score

        def print_score(self):
                print('%s: %s' % (self.name, self.score))
```

```
michael = Student('Michael','A')
bob = Student('Bob','C')

#类的方法
isinstance(michael,Student)  #True
isinstance(bob,Student)      #True
isinstance(Student,object)   #True
```

- 类（class）是指Student这个概念
- 实例（Instance）是一个个具体的Student，Bob，Michael

面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象

```
class Student:
        def __init__(self, name, score):
                self.name = name
                self.score = score
```

- 类相当于模版：在创建实例的时候，可以把我们认为模版必须有的属性填写进去。
- 定义一个构造函数__init__，在创建实例的时候就把name，score等属性绑上去。
- 注意：__init__前后分别有两个下划线！！
- __init__的第一个参数永远是self，表示创建的实例本身：因为self指向创建的实例本身，__init__把各种属性绑定到self，即绑定到实例。

面向对象

```
class Student:
        def __init__(self, name, score):
                self.name = name
                self.score = score

        def print_score(self):
                print('%s: %s' % (self.name, self.score))
```

- 类的**所有方法**都必须**至少**有一个名为self的参数，并且必须是方法的第一个参数，self参数代表将来要创建的对象本身。
- 在类的方法中访问实例属性时需要以self为前缀。
- 在外部通过对象名调用对象方法时**并不需要**传递这个参数。

面向对象

```
class Student:
        def __init__(aabb, name, score):
                aabb.name = name
                aabb.score = score

        def print_score(aabb):
                print('%s: %s' % (aabb.name, aabb.score))
```

- 注意：
  - self只是一个习惯
  - 实际名字是可以变化的，不一定是self
  - 建议编写代码时仍以self作为方法的第一个参数名字

实例属性与类属性

- 实例属性：
  - 一般在构造函数__init__中定义；定义和使用时必须以self作为前缀
  - 属于实例，只能通过对象名访问
- 类属性：
  - 类中所有方法之前定义的数据成员
  - 通过对象名或类名访问

```
class Student:
        university = "Shenzhen University"
        def __init__(self, name, score):
                self.name = name
                self.score = score
```

实例属性与类属性

```
class Student:
        university = "Shenzhen University"
        def __init__(self, name, score):
                self.name = name
                self.score = score

    print(michael.name,michael.score,
          michael.university,Student.university)
    #Michael A Shenzhen University Shenzhen University


    print(bob.name,bob.score,
          bob.university,Student.university)
    #Bob C Shenzhen University Shenzhen University

    print(Student.name)
```

```
Traceback (most recent call last):
  File "test.py", line 18, in <module>
    print(Student.name)
AttributeError: type object 'Student' has no attribute 'name'
```

实例属性与类属性

```
class Student:
        university = "Shenzhen University"
        def __init__(self, name, score):
                self.name = name
                self.score = score
```

```
michael = Student('Michael','A')
bob = Student('Bob','C')
```

#修改类属性

```
Student.university = 'SZU'
print(michael.university) #SZU
print(bob.university)     #SZU
```

#增加类属性

```
Student.major = 'CS'
print(michael.major)     #CS
print(bob.major)         #CS
```

#修改实例属性

```
michael.score = 'B'
bob.name = 'BOB'
```

#增加实例属性

```
michael.id = 1001
print(michael.id) #1001
print(bob.id)     #AttributeError
```

- Python可以动态地为类和对象增加成员，动态类型特点的一种重要体现。

实例属性与类属性

```
class Student:
        university = "Shenzhen University"
        def __init__(self, name, score):
                self.name = name
                self.score = score

michael = Student('Michael','A')
bob = Student('Bob','C')

#动态增加成员方法
import types
michael.change_score = types.MethodType(change_score,michael)

michael.change_score('B')
print(michael.score)    #B

bob.change_score('D')
print(bob.score)        #AttributeError

del michael.change_score  #删除
```

应用：管理复杂的用户分类：不同用户组具有不同的行为和权限，并且可能会经常改变。

私有变量

```
class Student:
        university = "Shenzhen University"
        def __init__(self, name, score):
                self.name = name
                self.score = score
```

```
#如果不想内部属性在外部被修改应该怎么办?
michael.score = 'B'
bob.name = 'BOB'
```

- 可以把属性的名称前加上两个下划线__
- 实例的变量名如果以__开头，代表一个私有变量（private）
- 只有内部可以访问，外部不能访问！
- 一般通过调用对象的公有成员方法来访问

```
self.__name = name
self.__score = score
```

私有变量

```
class Student:
        def __init__(self, name, score):
                self.__name = name
                self.__score = score

        def show_score(self):
                print('%s' % self.__score)


michael = Student('Michael','A')
bob = Student('Bob','C')

print(michael.__score)
AttributeError: 'Student' object has no attribute '__score'

print(bob.__name)
AttributeError: 'Student' object has no attribute '__name'

michael.show_score()
A
```

确保外部代码不能随意修改对象内部的状态!

私有变量

```
class Student:
        def __init__(self, name, score):
                self.__name = name
                self.__score = score

        def show_score(self):    #显示
                return self.__score

        def change_score(self,score): #修改
                self.__score = score


michael = Student('Michael','A')
print(michael.show_score())      #A

michael.change_score('B')
print(michael.show_score())      #B
```

私有变量

- 另一种声音：Python中不存在严格意义上的私有成员。
- 例如：不能直接访问__name是因为：
  - Python解释器对外把__name变量改成了_Student__name
  - 可以通过_Student__name来访问__name变量

```
class Student:
        def __init__(self, name, score):
        self.__name = name
                self.__score = score


 michael = Student('Michael','A')

 print(michael.__name)    # error
 print(michael.__score)   # error
 print(michael._Student__name)   #Michael
 print(michael._Student__score) #A
```

不建议：不同版本的Python解释器可能会把__name改成不同的变量名。

面向对象

- 进一步理解：

```
class Student:
        def __init__(self, name, score):
                self.__name = name
                self.__score = score

        def show_score(self):                    #显示
                return self.__score

        def change_score(self,score):  #修改
                self.__score = score
```

```
michael=Student('Michael','A')
michael.__score ='B'
print(michael.__score)   #B，修改成功了？
```

面向对象

- 进一步理解：

```
class Student:
        def __init__(self, name, score):
                self.__name = name
                self.__score = score

        def show_score(self):                    #显示
                return self.__score

        def change_score(self,score): #修改
                self.__score = score


michael=Student('Michael','A')
michael.__score ='B'
print(michael.__score)          #B，修改成功了？
print(michael.show_score())     #A
```

- 表面上，外部代码“成功”修改了__score变量
- 实际上，这个__score变量和class内部的__ score变量不是一个变量！
- 内部的__ score变量被Python解释器自动改成了_Student__ score
- 外部代码给michael新增了一个__ score变量

面向对象

- 在Python中，以下划线开头的变量名和方法名有特殊的含义，尤其是在类的定义中。
- 用下划线作为变量名和方法名前缀和后缀来表示类的特殊成员：
  - ✓ __xxx__：系统定义的特殊成员、特殊变量、可以直接访问的
  - ✓ __xxx：私有成员，只有类对象自己能访问，但在对象外部可以通过“对象名._类名__xxx”这样的特殊方式来访问
  - ✓ _xxx：受保护成员，不能用'from module import *'导入

类属性:可变类型

注意:

```
class Dog:
        motion = []          #属于可变类型的列表
        def __init__(self, name):
                self.name = name

        def add_motion(self, motion):
                self.motion.append(motion)

d1 = Dog('Fido')
d2 = Dog('Buddy')
d1.add_motion('run')
d2.add_motion('sit')

print(d1.motion)                    # ['run', 'sit']
```

解决办法

```
def __init__(self, name):
        self.name = name
        self.tricks = []
```

公有方法、私有方法

- 公有方法、私有方法都属于对象。私有方法的名字以两个下划线“__”开始。
- 每个对象都有自己的公有方法和私有方法，可以访问属于类和对象的成员；
- 公有方法通过对象名直接调用；私有方法不能通过对象名直接调用，只能在属于对象的方法中通过self调用

```
class Student:
        def __init__(self, name, score):
                self.__name = name
                self.__score = score

        def __change_score(self,score):
                self.__score = score


michael = Student('Michael','A')
michael.__change_score('B')
#AttributeError: 'Student' object has no attribute '__change_score'
```

公有方法、私有方法

- 如果真的想要在外部调用__change_score()

```
class Student:
        def __init__(self, name, score):
                self.__name = name
                self.__score = score

        def show_score(self):
                return self.__score
        def __change_score(self,score):
                self.__score = score


michael = Student('Michael','A')
michael._Student__change_score('B')
print(michael.show_score())          #B
```

换一个例子：宠物

- 编写一个宠物的类

```
class Pet:
    def __init__(self, name, age=0):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age
```

```
mypet1 = Pet('Ben')
print(mypet1.get_name()) #Ben
print(mypet1.get_age())  #0
```

```
mypet2 = Pet(age=2,name='Bob')
print(mypet2.get_name()) #Bob
print(mypet2.get_age())  #2
```

需要复习上周的函数参数的用法

换一个例子：宠物

- 编写一个宠物的类

```
class Pet:
    def __init__(self, name, age=0):
        self. __name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def __str__(self):
        return "This pet's name is " + str(self.__name)
```

内置函数__str__定义当我们打印Pet实例时要发生的操作。在这里我们重写它来打印宠物的名字。

```
mypet1 = Pet('Ben')
print(mypet1)
```

This pet's name is Ben

特殊方法

- Python类有大量的特殊方法，其中比较常见的是构造函数和析构函数
  - ✓ 类的构造函数(constructor) 是__init__()，一般用来为数据成员设置初值或进行其他必要的初始化工作，在创建对象时被自动调用和执行。如果用户没有设计构造函数，Python将提供一个默认的构造函数用来进行必要的初始化工作。
  - ✓ 类的析构函数(destructor) 是__del__()，一般用来释放对象占用的资源，在Python删除对象和收回对象空间时被自动调用和执行。如果用户没有编写析构函数，Python将提供一个默认的析构函数进行必要的清理工作。
- 除此之外，可以通过重写特殊方法来实现运算符重载。

特殊方法：例子

- 一个列表中每个元素都加上一个相同数字？
- **面向过程：**

```
lst = [1,2,3,5]
[(item + 2) for item in lst]
```

- **面向对象：**lst是一个对象，2是一个对象，对象与对象的操作！
- `[1,2,3,5] + 2` ???

```
>>> [1,2,3,5] + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

- `[1,2,3,5] + [2]*4` ???

```
>>> [1,2,3,5] + [2]*4
[1, 2, 3, 5, 2, 2, 2, 2]
```

特殊方法：例子

- 面向对象：

  重新编写类 + 重写特殊方法来实现运算符重载（override）

```
class mylist:
        def __init__(self,lst=[]):
                self.__size = len(lst)
                self.__list = lst

        def add_item(self,item):
                self.__list.append(item)

        def __add__(self, n):
                temp = mylist()
                for item in self.__list:
                        temp.add_item(item+n)
                return temp. __list
```

特殊方法：例子

```
class mylist:
        def __init__(self,lst=[]):
                self.__size = len(lst)
                self.__list = lst

        def add_item(self,item):
                self.__list.append(item)

        def __add__(self, n):
                temp = mylist()
                for item in self.__list:
                        temp.add_item(item+n)
                return temp. __list
```

重写内置函数__add__
实现运算符重载

这样写好不好？
既然是面向对象编程，最好是返回整个mylist对象！

```
lst = mylist([1,2,3,5])
print(lst+2)   #[3, 4, 5, 7]
```

return temp
如果是返回整个mylist对象，这里会输出
<__main__.mylist object at 0x101870c10>

特殊方法：例子

```
class mylist:
        def __init__(self,lst=[]):
                self.__size = len(lst)
                self.__list = lst

        def add_item(self,item):
                self.__list.append(item)

        def __add__(self, n):
                temp = mylist()
                for item in self.__list:
                        temp.add_item(item+n)
                return temp

        def __str__(self):
                return str(self.__list)


lst = mylist([1,2,3,5])
print(lst+2)   #[3, 4, 5, 7]
```

内置函数__str__定义我们想要的输出，处理的是mylist对象，输出mylist对象的属性

特殊方法

更多特殊方法以及它们的用法

https://diveintopython3.net/special-method-names.html

https://docs.python.org/3.8/reference/datamodel.html

特殊方法

练习

```
class Point:
        def __init__(self, x=0, y=0):
                self.x = x
                self.y = y

        def __add__(self, other):
                return Point(self.x + other.x, self.y + other.y)
        def __str__(self):
                return "Point({0}, {1})".format(self.x, self.y)



        A = Point(3, 4)
        B = Point(-1, 2)
        str(A)              #? ? ?
        print(A + B)        #? ? ?
```

特殊方法

练习

```
class Point:
        def __init__(self, x=0, y=0):
                self.x = x
                self.y = y

        def __add__(self, other):
                return Point(self.x + other.x, self.y + other.y)
        def __str__(self):
                return "Point({0}, {1})".format(self.x, self.y)



        A = Point(3, 4)
        B = Point(-1, 2)
        str(A)          # => Point(3, 4)
        print(A + B)    # => Point(2, 6)
```

继承 inheritance

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为**子类或派生类**（Subclass），而被继承的class称为**基类、父类或超类**（Base class、Super class）。

```
class Animal(object):
    def run(self):
        print('Animal is running...')

class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

继承于Animal类

继承于Animal类

pass表示跳过：不需要实现，也可以运行？

对于Dog、 Cat来说，Animal是它的父类，
对于Animal来说，Dog、 Cat是它的子类。

继承 inheritance

```
class Animal(object):
    def run(self):
        print('Animal is running...')

class Dog(Animal):
    pass

class Cat(Animal):
    pass

dog = Dog()
dog.run()    #Animal is running...

cat = Cat()
cat.run()    #Animal is running...
```

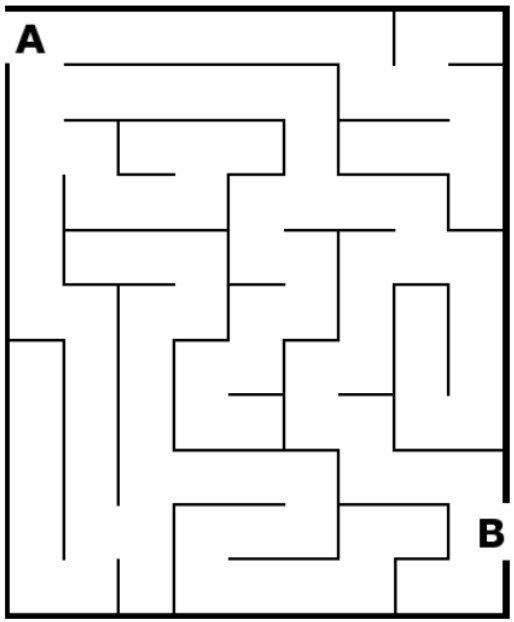
子类获得了父类的全部功能。由于Animial实现了run()方法，因此，Dog和Cat作为它的子类，自动拥有了run()方法

■继承是为代码复用和设计复用而设计的，是面向对象程序设计的重要特性之一。设计一个新类时，如果可以继承一个已有的设计良好的类然后进行二次开发，无疑会大幅度减少开发工作量。

继承 inheritance

摘自： **Information and Software Engineering Practice,** CUHK (C#, OOB)

Example: Maze game



继承 inheritance

Example: Maze game

抽象出对象

- **The game**
  - **A maze *has-a* number of rooms**
  - **A room has four sides (North, East, South, West)**
  - **A player enters a side**
    - **If the side is a door, it leads to another room**
    - **If the side is a wall, the player is hurt**
  - **The player wins if he/she can leave the maze**

继承 inheritance

找出对象之间的关系

The objects

- `MazeGame, Maze, Room, Door, Wall, Side`

- **Abstraction**
  - **Treat objects uniformly**
  - **Group `Room, Door` and `Wall` by base class `MapSite`**
  - **All share the base class virtual function**
    - `Enter()`

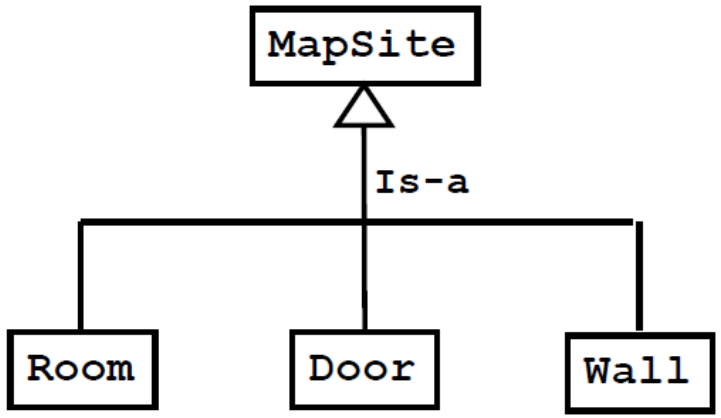
继承 inheritance

Is-a relation

- **A `Room` *is-a* kind of `MapSite`**
- **A `Door` *is-a* kind of `MapSite`**
- **A `Wall` *is-a* kind of `MapSite`**

对象之间的关系
is-a:继承

- **Is-a suggests *class inheritance***



继承 inheritance

设计模式 （Design Patterns），通常被有经验的**面向对象的软件开发人员**所采用，是软件开发人员在软件开发过程中面临的一般问题的解决方案。

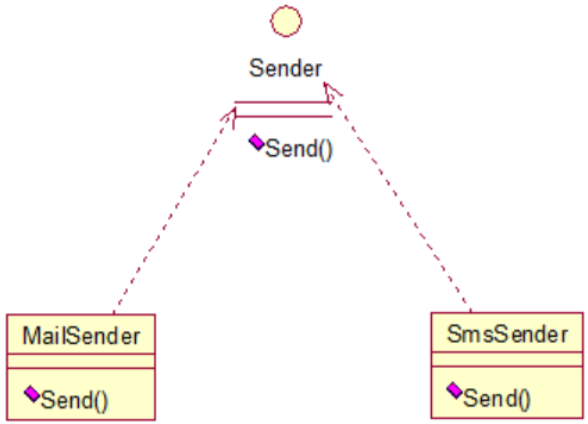
工厂方法模式（**Factory Method**）

Factory Method

- **Define an interface (the virtual functions) for creating an object, but let *subclasses* decide how to instantiate.**

  Factory Method lets a class *defer instantiation to subclasses*.

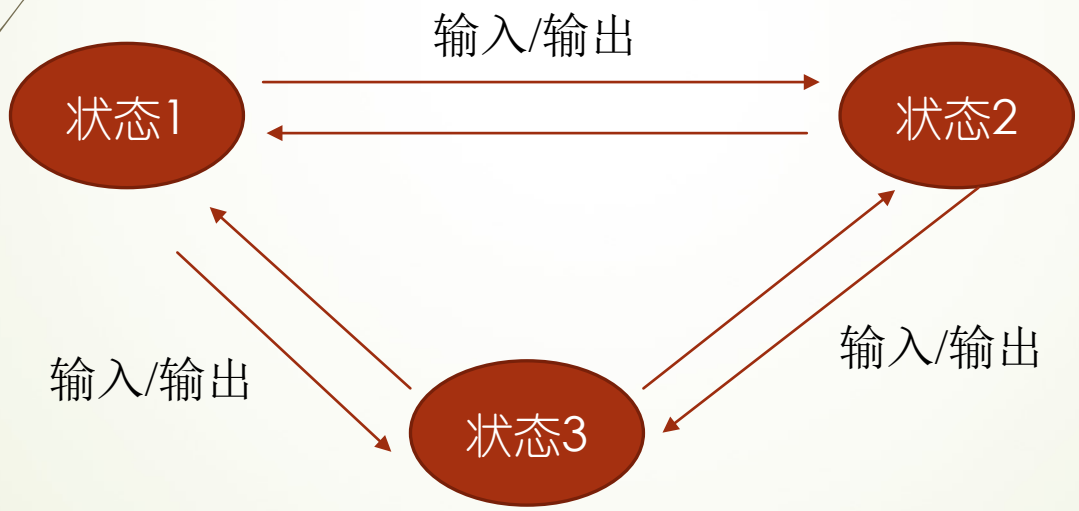
- **Also known as virtual constructor**



继承 inheritance

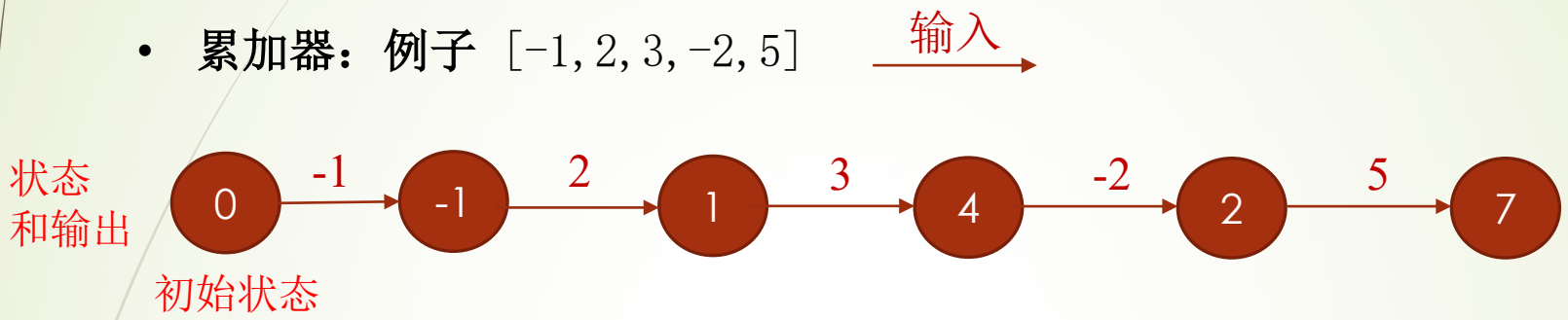
摘自： **Machine Learning,** MIT (Python, OOB)

状态机（强化学习，马尔科夫决策过程）



继承 inheritance

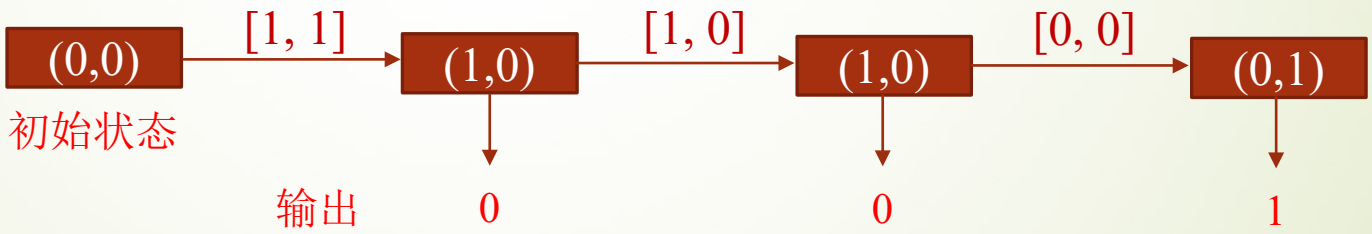
- **累加器：例子** [-1, 2, 3, -2, 5]



- **二进制相加** 定义状态：（进位，结果）

例子

```
  0 1 1
+ 0 0 1
-------
  1 0 0
```



- **相同点：转换（输入→状态转换→输出→输入→状态转换→输出…）**
- **不同点：状态的形式，具体的状态转换函数，输出函数**

继承 inheritance

父类 ← 继承 ← 子类

```
class SM:
    start_state = None  # default start state

    def transition_fn(self, s, x):
        '''s:       the current state
           x:       the given input
           returns: the next state'''
        raise NotImplementedError

    def output_fn(self, s):
        '''s:       the current state
           returns: the corresponding output'''
        raise NotImplementedError

    def transduce(self, input_seq):
        … {use transition_fn and output_fn}
```

```
class Accumulator(SM):
    start_state = 0

    def transition_fn(self, s, x):
        …

    def output_fn(self, s):
        …
```

```
class Binary_Addition(SM):
    start_state = (0,0)  #(carry,digit)

    def transition_fn(self, s, x):
        …

    def output_fn(self, s):
        …
```

- **Programming to an Interface, not an Implementation**
  - **Interface is just like a socket**
  - **Implementation can be changed easily in future**

好处？稍后揭晓！

继承 inheritance

我们回到简单的例子：

```
class Animal(object):
    def run(self):
        print('Animal is running...')

class Dog(Animal):
    pass

class Cat(Animal):
    pass

dog = Dog()
dog.run()    #Animal is running...

cat = Cat()
cat.run()    #Animal is running...
```

Dog和Cat都输出Animal，显然是不那么合理的==

```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

class Cat(Animal):
    def run(self):
        print('Cat is running...')

dog = Dog()
dog.run() #Dog is running...

cat = Cat()
cat.run() #Cat is running...
```

当子类和父类都存在相同的run()方法时，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()。

多态（Polymorphism）
按字面的意思就是“多种状态”。

继承 inheritance

当然，我们也可以对子类增加一些父类没有的方法

```
class Dog(Animal):
     def run(self):
        print('Dog is running...' )

     def eat(self):
        print('Dogs like meat...')

class Cat(Animal):
     def run(self):
        print('Cat is running...' )

     def eat(self):
        print('Cats like fish...')

dog = Dog()
dog.eat()    #Dogs like meat...

cat = Cat()
cat.eat()    #Cats like fish...
```

继承 inheritance

- issubclass(*class*, *classinfo*)，如果class是classinfo的子类，返回True

```
class Animal(object):
     pass
class Dog(Animal):
     pass
class Cat(Animal):
     pass

issubclass(Animal, Animal)    #True
issubclass(Dog, Animal)       #True
issubclass(Cat, Animal)       #True
issubclass(Animal, object)    #True
issubclass(Dog, object)       #True
issubclass(int, Animal)       #False
issubclass(list, Animal)      #False
```

继承 inheritance

- 注意：以下代码能否正确运行？

```
class Father:
        def __init__(self, iterable=[1,2,3]):
                self.items_list = []
                self.update(iterable)

        def update(self, iterable):
                for item in iterable:
                        self.items_list.append(item)

class Son(Father):
        def update(self, keys, values):
                for item in zip(keys, values):
                        self.items_list.append(item)

a=Father()
b=Son()
```

继承 inheritance

```
class Father:
        def __init__(self, iterable=[1,2,3]):
                self.items_list = []
                self.update(iterable)

        def update(self, iterable):
                for item in iterable:
                        self.items_list.append(item)

class Son(Father):
        def update(self, keys, values):
                for item in zip(keys, values):
                        self.items_list.append(item)

a=Father()
b=Son()
```

```
Traceback (most recent call last):
 line 15, in <module>
    b=Son()
 line 4, in __init__
    self.update(iterable)
TypeError: update() missing 1 required positional argument: 'values
```

？？？我们不是重写了update吗？？？

继承 inheritance

```
class Father:
        def __init__(self, iterable=[1,2,3]):
                self.items_list = []
                self.update(iterable)

        def update(self, iterable):
                for item in iterable:
                        self.items_list.append(item)

class Son(Father):

        def update(self, keys, values):
                for item in zip(keys, values):
                        self.items_list.append(item)

a=Father()
b=Son()
```

但我们没有重写__init__!!!

__init__ 调用 update (public): Python 找到的是Son的update.

继承 inheritance

练习

```
class A(object):
        def __init__(self):
                self.__private()
                self.public()
        def __private(self):
                print('__private() method in A')
        def public(self):
                print('public() method in A')


class B(A):
        def __private(self):
                print('__private() method in B')
        def public(self):
                print('public() method in B')
b=B() #输出什么?
```

继承 inheritance

练习

```
class A(object):
        def __init__(self):
                self.__private()
                self.public()
        def __private(self):
                print('__private() method in A')
        def public(self):
                print('public() method in A')


class B(A):
        def __private(self):
                print('__private() method in B')
        def public(self):
                print('public() method in B')
b=B()
```

```
__private() method in A
public() method in B
```

多态 polymorphism

- 所谓多态（polymorphism），是指父类的同一个方法在不同子类对象中具有不同的表现和行为。
- 子类继承了父类行为和属性之后，还会增加某些特定的行为和属性，同时还可能会对继承来的某些行为进行一定的改变。

小测试:

```
a = Animal()
d = Dog()
c = Cat()

#返回?
isinstance(a, Animal)
isinstance(a, Dog)
isinstance(a, Cat)

isinstance(d, Animal)
isinstance(d, Dog)
isinstance(a, Cat)

isinstance(c, Animal)
isinstance(c, Dog)
isinstance(c, Cat)
```

多态 polymorphism

小测试:

```
a = Animal()
d = Dog()
c = Cat()

#返回?
isinstance(a, Animal)  #True
isinstance(a, Dog)     #False
isinstance(a, Cat)     #False

isinstance(d, Animal)  #True
isinstance(d, Dog)     #True
isinstance(a, Cat)     #False

isinstance(c, Animal)  #True
isinstance(c, Dog)     #False
isinstance(c, Cat)     #True
```

- Dog、Cat本来就是Animal的一种！
- 在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行。

多态 polymorphism

多态的好处

```
class Animal(object):
    def run(self):
        print('Animal is running...')
    def run_twice(self):
        self.run()
        self.run()

class Dog(Animal):
    def run(self):
        print('Dog is running...')

class Cat(Animal):
    def run(self):
        print('Cat is running...')


Dog().run_twice()
Cat().run_twice()
```

新的类

```
class Pig(Animal):
    def run(self):
        print('Pig is running...')

class Goat(Animal):
    def run(self):
        print('Goat is running...' )

class Mouse(Animal):
    def run(self):
        print('Mouse is running...')

class Cow(Animal):
    def run(self):
        print('Cow is running...')

Pig().run_twice()
Goat().run_twice()
Mouse().run_twice()
Cow().run_twice()
```

- 新增一个Animal的子类，不必对run_twice()做任何修改
- 只要是Animal，就可以调用run_twice()
- run_twice()里面的run()由具体的子类决定

多态 polymorphism

父类 ← 继承 ← 子类

```
class SM:
    start_state = None  # default start state

    def transition_fn(self, s, x):
        '''s:       the current state
           x:       the given input
           returns: the next state'''
        raise NotImplementedError

    def output_fn(self, s):
        '''s:       the current state
           returns: the corresponding output'''
        raise NotImplementedError

    def transduce(self, input_seq):
        … {use transition_fn and output_fn}
```

```
class Accumulator(SM):
    start_state = 0

    def transition_fn(self, s, x):
        …

    def output_fn(self, s):
        …
```

```
class Binary_Addition(SM):
    start_state = (0,0)   #(carry,digit)

    def transition_fn(self, s, x):
        …

    def output_fn(self, s):
        …
```

- Accumulator和Binary_Addition都可以调用transduce；
- **编程思想：开闭原则**
  - 对拓展开放：允许新增状态机（SM）的子类
  - 对修改封闭：不需要依赖父类状态机（SM）的方法（ transduce ）修改

多态 polymorphism

练习

```
class Animal(object):
    def show(self):
        print('I am an animal.')
class Cat(Animal):
    def show(self):
        print('I am a cat.')
class Dog(Animal):
    def show(self):
        print('I am a dog.')
class Tiger(Animal):
    def show(self):
        print('I am a tiger.')
class Test(Animal):
    pass
```

```
x = [item() for item in
    (Animal, Cat, Dog, Tiger, Test)]
for item in x:
    item.show()
```

输出什么？

多态 polymorphism

练习

```
class Animal(object):
    def show(self):
        print('I am an animal.')
class Cat(Animal):
    def show(self):
        print('I am a cat.')
class Dog(Animal):
    def show(self):
        print('I am a dog.')
class Tiger(Animal):
    def show(self):
        print('I am a tiger.')
class Test(Animal):
    pass
```

```
x = [item() for item in
    (Animal, Cat, Dog, Tiger, Test)]
for item in x:
    item.show()
```

```
I am an animal.
I am a cat.
I am a dog.
I am a tiger.
I am an animal.
```

多重继承

动物园

```
class Animal(object):
        pass

class Mammal(Animal):
        pass

class Bird(Animal):
        pass
```

每增加一种分类标准，子类将大量增加

```
class Animal(object):
        pass

class FlyMammal(Animal):
        pass

class FlyBird(Animal):
        pass

class GroundBird(Animal):
        pass

class GroundMammal(Animal):
        pass
```

多重继承

动物园

```
class Animal(object):
    pass

class Mammal(Animal):
    pass

class Bird(Animal):
    pass
```

```
class fly(object):
    pass

class Ground(object):
    pass
```

一个子类继承多个父类

```
class Dog(Mammal, Ground):
    pass

class Bat(Mammal, fly):
    pass

class Parrot(Bird, fly):
    pass

class Ostrich(Bird, Ground):
    pass
```

多重继承

```
class Animal(object):
    pass

class Mammal(Animal):
    pass

class Bird(Animal):
    pass
```

```
class fly(object):
    pass

class Ground(object):
    pass
```

```
class pet(object):
    pass
```

一个子类继承多个父类

```
class Dog(Mammal, Ground, Pet):
    pass

class Bat(Mammal, fly):
    pass

class Parrot(Bird, fly , Pet):
    pass

class Ostrich(Bird, Ground):
    pass
```

多重继承

```
class Canadian:
        def __init__(self, first_name, last_name):
                self.first_name = first_name
                self.last_name = last_name
        def print_name(self):
                print("I'm {} {}, Canadian!".format(self.first_name,self.last_name))

class American:
        def __init__(self, first_name, last_name):
                self.first_name = first_name
                self.last_name = last_name
        def print_name(self):
                print("I'm {} {}, American!".format(self.first_name, self.last_name))

class BritishColumbian(Canadian, American):
        pass


michael = BritishColumbian("Michael", "Cooper")

michael.print_name()#输出什么？
```

多重继承

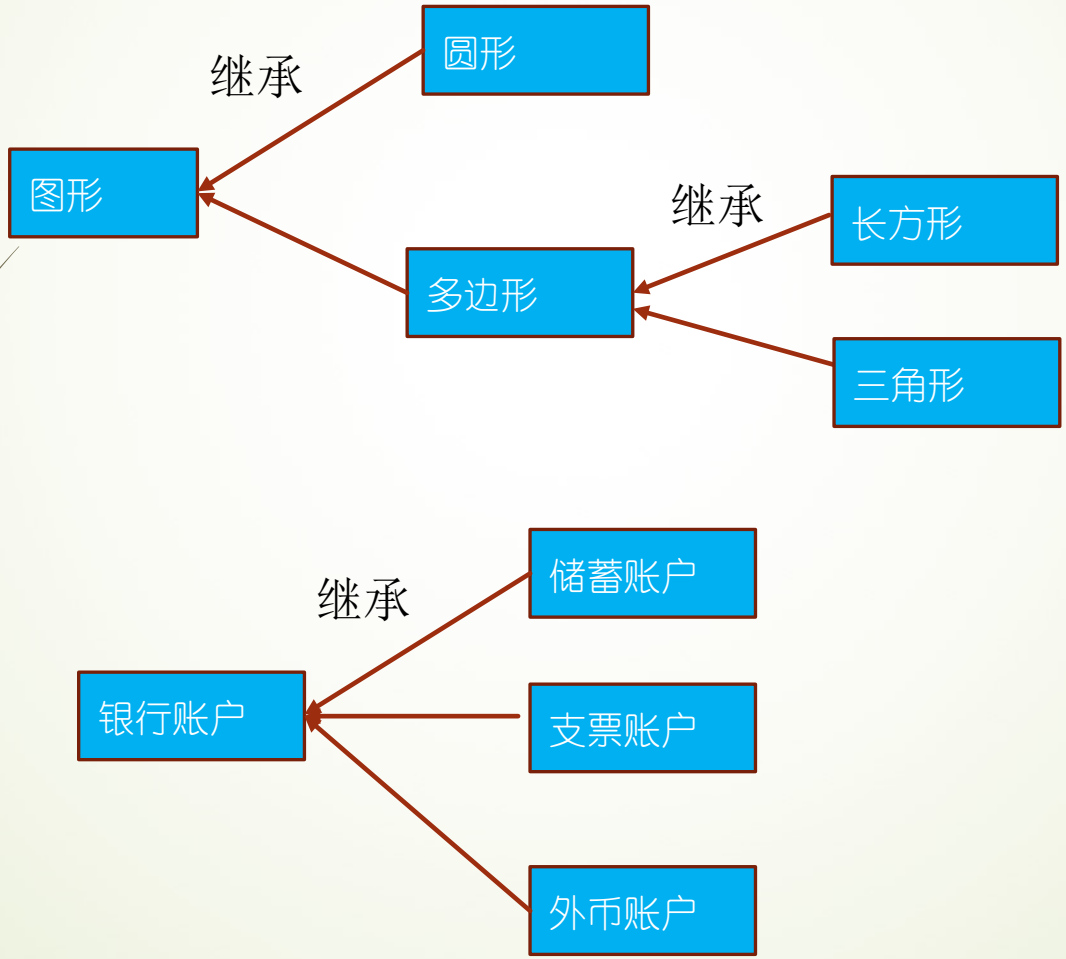
```
class Canadian:
        def __init__(self, first_name, last_name):
                self.first_name = first_name
                self.last_name = last_name
        def print_name(self):
                print("I'm {} {}, Canadian!".format(self.first_name,self.last_name))

class American:
        def __init__(self, first_name, last_name):
                self.first_name = first_name
                self.last_name = last_name
        def print_name(self):
                print("I'm {} {}, American!".format(self.first_name, self.last_name))

class BritishColumbian(Canadian, American):
        pass


michael = BritishColumbian("Michael", "Cooper")

michael.print_name()#I'm Michael Cooper,  Canadian!
```

Python 先寻找 Canadian类，再寻找American类。

面向对象设计案例

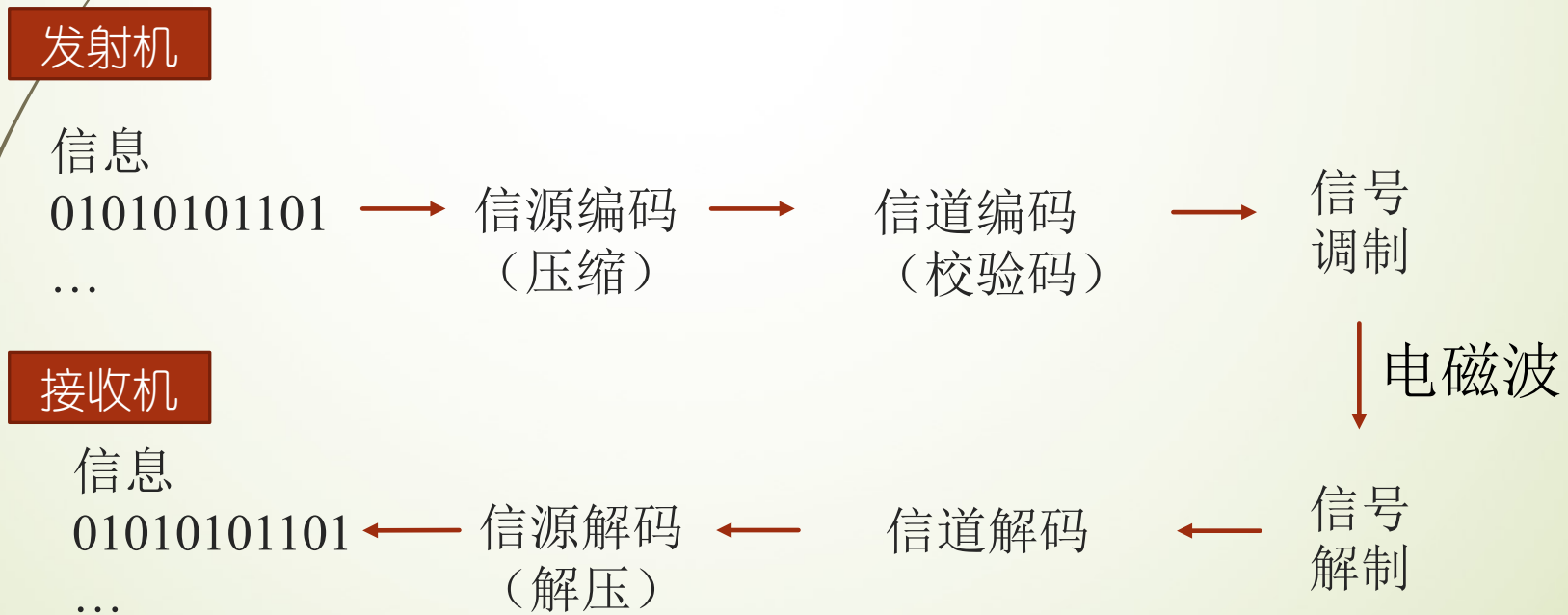
```
class Stack:
    def __init__(self, size = 10):
        self._content = []              #使用列表存放栈的元素
        self._size = size               #初始栈大小
        self._current = 0               #栈中元素个数初始化为0

    def empty(self):
    def isEmpty(self):
    def isFull(self):

    def setSize(self, size) :
                #如果缩小栈空间，则删除指定大小之后的已有元素

    def push(self, v):
    def pop(self):
    def show(self):
```

面向对象设计案例



面向对象设计案例


发射机
无线信号
数据包
接收机


发射机
信息
01010101101
…
信源编码
（压缩）
信道编码
（校验码）
信号
调制
电磁波
接收机
信息
01010101101
…
信源解码
（解压）
信道解码
信号
解制

面向对象设计案例

抽象出对象

收发机？

Wifi接收机

Wifi发射机

蓝牙接收机

蓝牙发射机

信息

信道编码
华为的极化码

信道编码
高通的LDPC码

信源编码
JPEG

信源编码
PDF

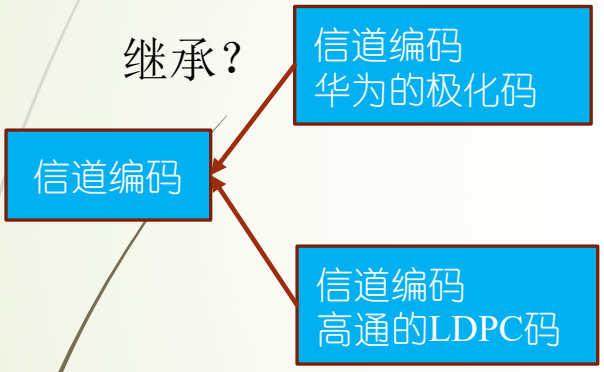
信源编码
MP3

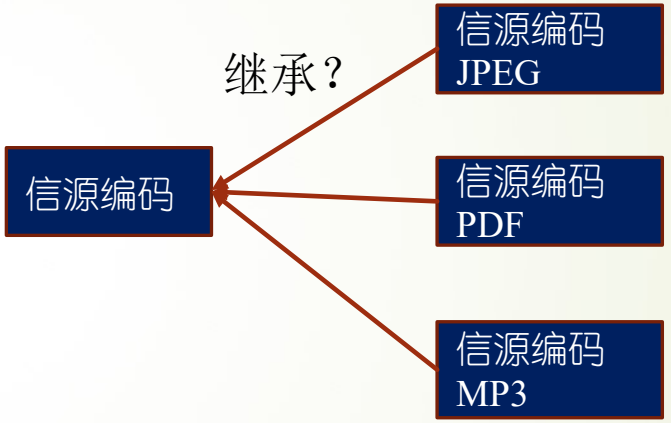
影音大数据包的调制

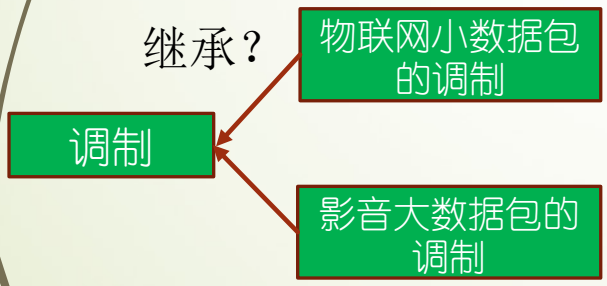
物联网小数据包的调制

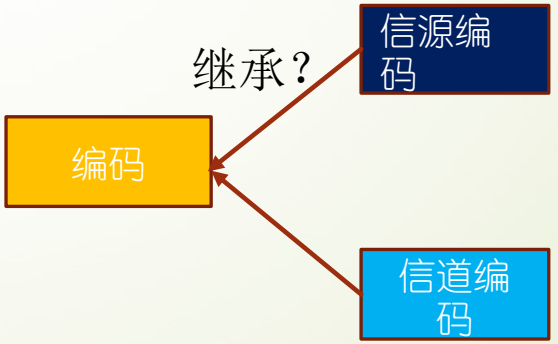
面向对象设计案例

处理好对象之间的关系
***Is-a* relationship？**



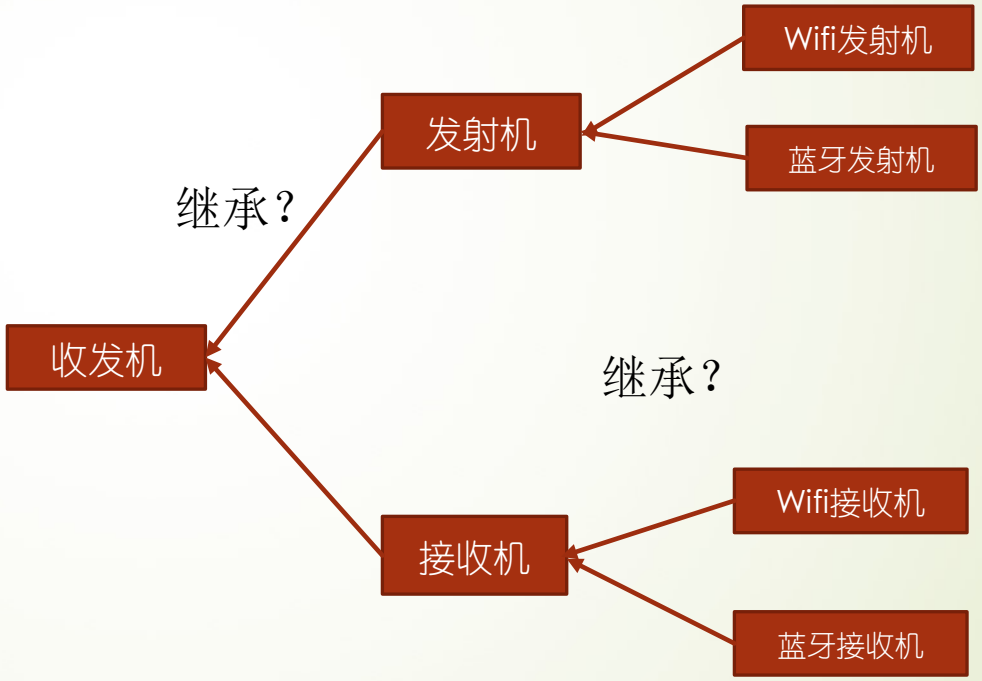






面向对象设计案例

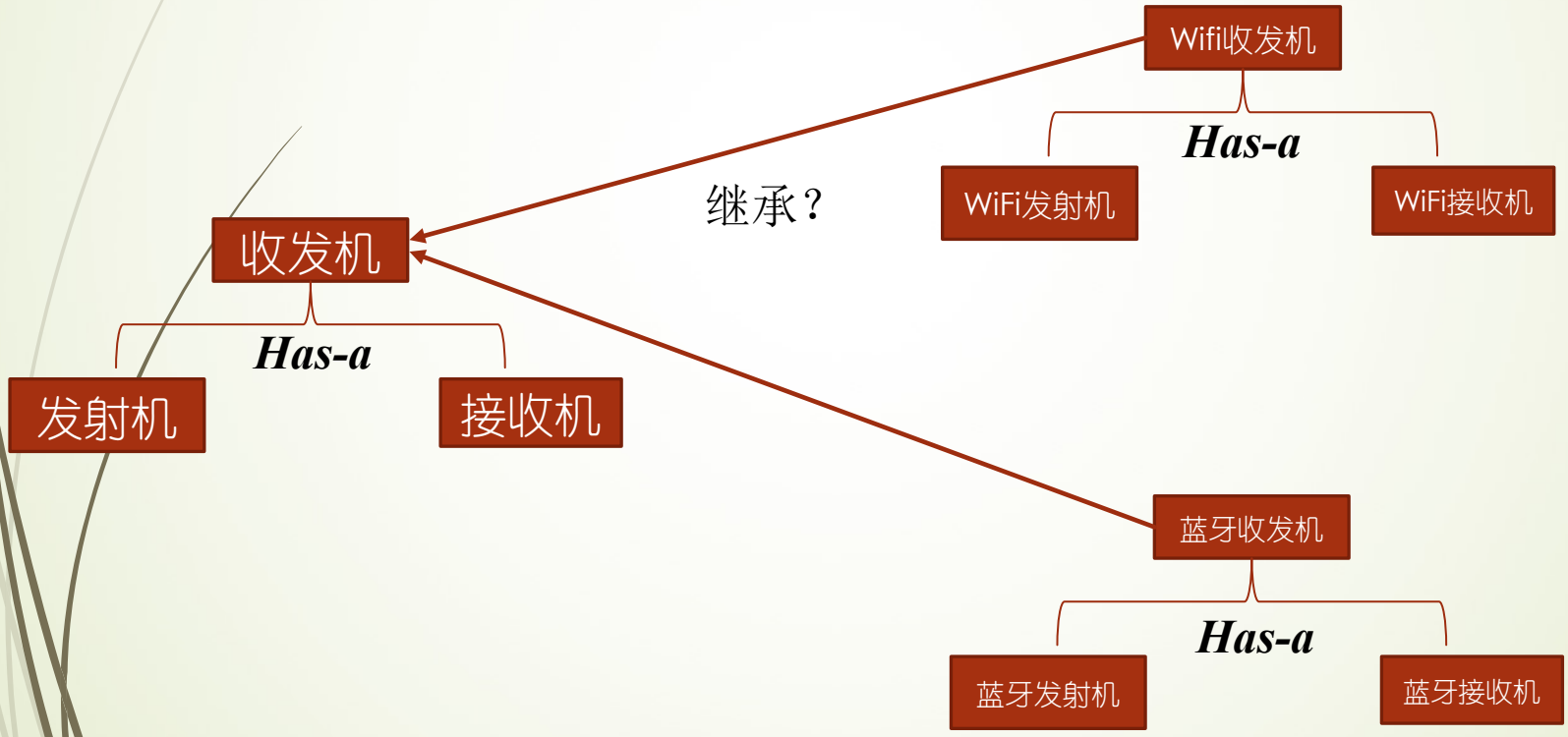
处理好对象之间的关系
***Is-a* relationship？**



发射机和接收机属于收发机的一部分

面向对象设计案例

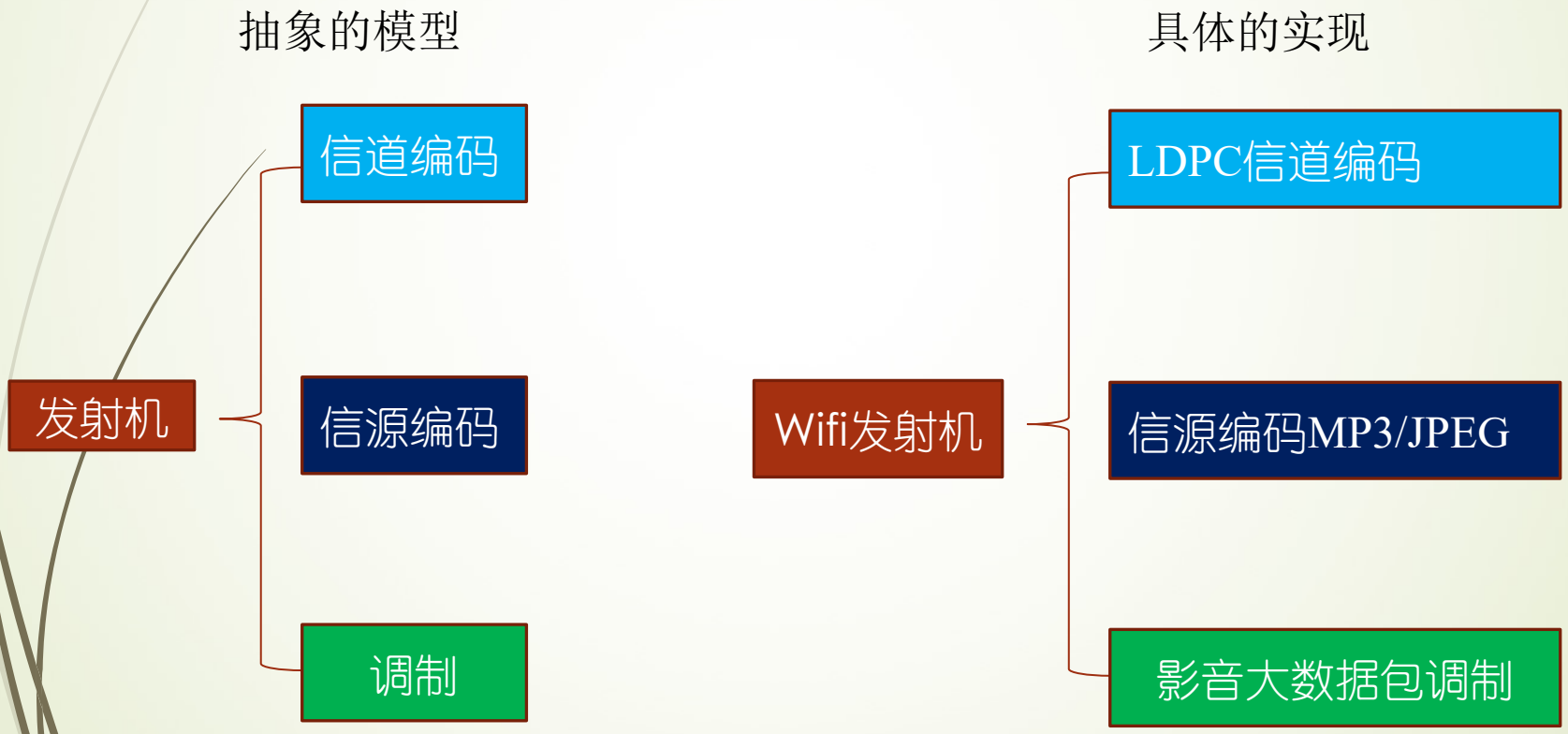
处理好对象之间的关系
***Has-a* relationship？**



- **Programming to an Interface, not an Implementation**
  - **Interface is just like a socket**
  - **Implementation can be changed easily in future**

面向对象设计案例

处理好对象之间的关系
***Has-a* relationship?**

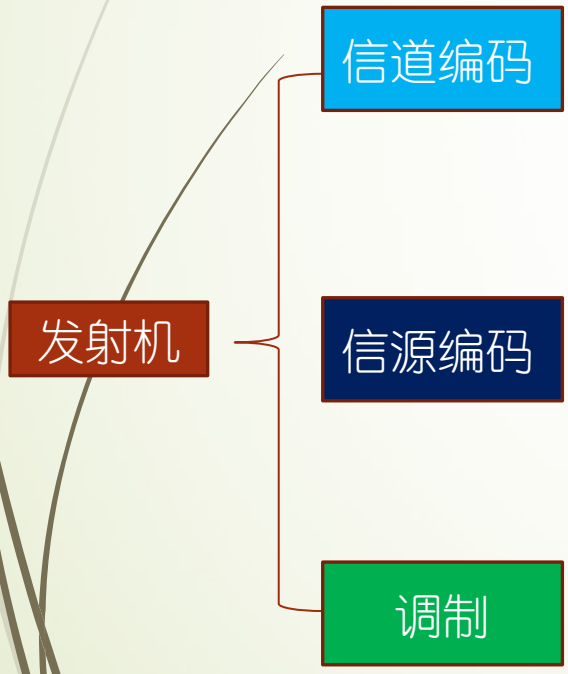


- **Programming to an Interface, not an Implementation**
  - **Interface is just like a socket**
  - **Implementation can be changed easily in future**

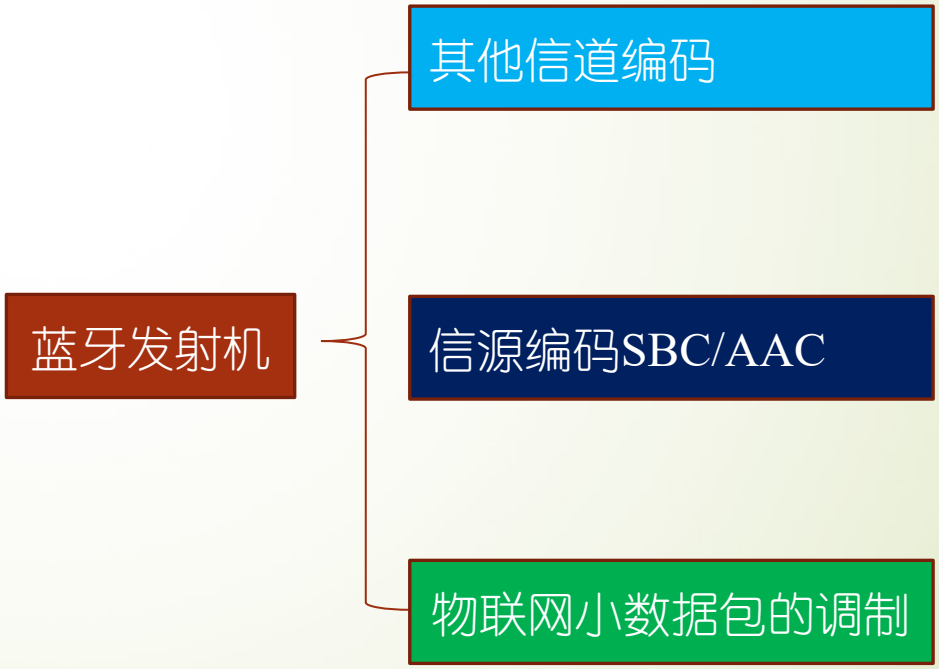
面向对象设计案例

处理好对象之间的关系
***Has-a* relationship?**

抽象的模型



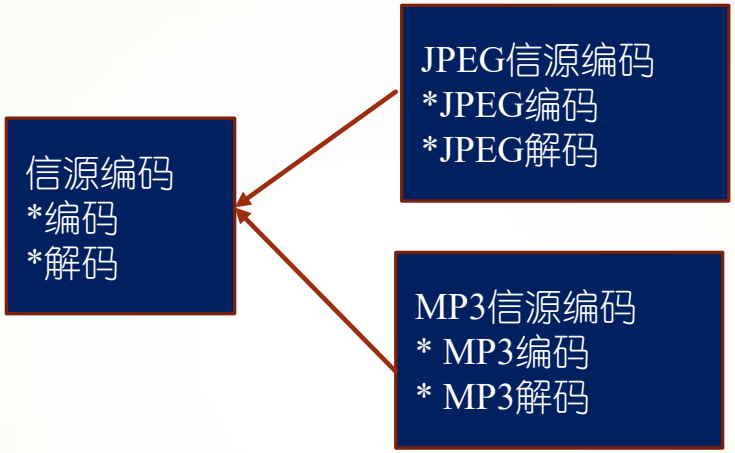
具体的实现



- **Programming to an Interface, not an Implementation**
  - **Interface is just like a socket**
  - **Implementation can be changed easily in future**

面向对象设计案例

对象的功能？



发射机 **func** (信源编码 , 信道编码 , 调制)

Wifi发射机 **func** (信源编码 MP3/JPEG , LDPC信道编码 , 影音大数据包调制)

蓝牙发射机 **func** (信源编码 SBC/AAC , 其他信道编码 , 物联网小数据包的调制)

多态

好好学习面向对象编程

Python

C++

C#

Java

Matlab

…