最终大作业：

- 完成1个python程序，题目不限，内容不限， 要求：
  - ✓ py文件数不少于4个，并且每个py文件需要有单元测试代码。
  - ✓ 程序运行的主入口： main.py
  - ✓ 调用到的模块数： >= 5个
  - ✓ 总代码数： >= 300行。

- 最终提交的内容：
  - ✓ 含有所有程序和数据的文件夹。
  - ✓ 利用pyinstaller，将python软件打包成可以独立运行的exe。
  - ✓ 一个doc/ ppt， 介绍程序的设计思路、实现的功能， 以及使用到的python知识点。
  - ✓ 一个演示视频， 介绍典型运行效果，老师同学现场评价。

周5下午现场演示　　最佳现场展示奖　　整体1/2/3等奖

Python面向对象编程

-基本概念

本节重点：

- 为什么要引入“类”
- 类的基本概念
- 类的基本使用
- 类的静态变量
- 对象方法、类方法、静态方法

类的引出

类的引出

人的建模：

- 姓名、年龄、身高、位置
- 说话， 走路

方式1：散装式，全局变量的处理的， 只能处理单个对象

```
#
# 定义了一堆数据
# name = 'wang'
# age  = 28
# height = 1.76
# pos = [0.0, 0.0]
#
```

```
# # 定义了两个方法
#
# def talk(info):
#     print(f"{name} age:{age} pos:{pos} say: {info}")
#     return None
#
# def run(dx, dy):
#     pos[0] += dy
#     pos[1] += dy
#
#
# talk("你好啊")
# run(1.0, 2.0)
# talk("我到这里了哈")
```

```
# talk("你好啊")
# run(1.0, 2.0)
# talk("我到这里了哈")
```

类的引出

人的建模：

- 姓名、年龄、身高、位置
- 说话， 走路

方法2， 通过1个字典的方式， 来打包数据，支持对多个对象的处理。。

```
#
# p1 = {'name':'wang', 'age': 28, 'height': 1.76, 'pos':[0.0,0.0]}
# p2 = {'name':'li', 'age': 18, 'height': 1.72, 'pos':[10.0,10.0]}
#
```

```
#
# def talk(p, info):
#     print(f"{p['name']} age:{p['age']} pos:{p['pos']} say: {info}")
#     return None
#
# def run(p, dx, dy):
#     p['pos'][0] += dx
#     p['pos'][1] += dy
```

```
# talk(p1, "你好啊")
# talk(p2, "你好啊")
# run(p1, 1.0, 2.0)
# run(p2, 1.0, 2.0)
# talk(p1, "我到这里了哈")
# talk(p2, "我到这里了哈")
```

类的引出

人的建模：

- 姓名、年龄、身高、位置
- 说话， 走路

方法3， 引入1个函数， 生成字典， 后面的方法不变

```
#
# def create_peson(name, age, height, pos):
#     return {'name':name, 'age': age, 'height': height, 'pos':pos}
#
# p1 = create_peson('wang', 28, 1.76, [0,0])
# p2 = create_peson('li', 18, 1.70, [10,10])
#
```

```
#
# def talk(p, info):
#     print(f"{p['name']} age:{p['age']} pos:{p['pos']} say: {info}")
#     return None
#
# def run(p, dx, dy):
#     p['pos'][0] += dx
#     p['pos'][1] += dy
```

```
# talk(p1, "你好啊")
# talk(p2, "你好啊")
# run(p1, 1.0, 2.0)
# run(p2, 1.0, 2.0)
# talk(p1, "我到这里了哈")
# talk(p2, "我到这里了哈")
```

类的引出

人的建模：

- 姓名、年龄、身高、位置
- 说话， 走路

方法4，引出人类， 是 数据和方法的集合或打包
有 4个属性，2个方法
属性： name, age, height, pos
方法： talk(), run()

```
# 类的定义
# class Person():
#     # def create_peson(name, age, height, pos):
#     #     return {'name': name, 'age': age, 'height': height, 'pos': pos}
#     def __init__(self, name, age, height, pos):
#         self.name = name
#         self.age = age
#         self.height = height
#         self.pos = pos
#
#     def talk(self, info):
#         print(f"{self.name} age:{self.age} pos:{self.pos} say: {info}")
#
#     def run(self, dx, dy):
#         self.pos[0] += dx
#         self.pos[1] += dy
```

```
# 类的实例化
# p1 = Person('wang', 28, 1.76, [0,0])
# p2 = Person('li', 18, 1.70, [10,10])
```

```
# 针对实例的增删改查， 以及实例之间的交互
# # 通过类来调用实例的方法
# Person.talk(p1, "你好啊")
# Person.talk(p2, "我很好，你也好啊")
# Person.run(p1, 1.0, 2.0)
# Person.run(p2, 2.0, 2.0)
# Person.talk(p1, "我到这里了哈")
# Person.talk(p2, "我到这里了哈")
#
```

```
#
# # 进一步简洁，直接通过实例来调用实例的方法
# p1.talk("你好啊")
# # 等价于: Person.talk(p1, "你好啊")
#
```

编程范式

编程是 程序 员 用特定的语法+数据结构+算法 组成的代码来告诉计算机如何执行任务的过程 。

一个程序是程序员为了得到一个任务结果而编写的一组指令的集合，实现一个任务的方式有很多种不同的方式， 对这些不同的编程方式的特点进行归纳总结得出来的编程方式类别，即为编程范式。 不同的编程范式本质上代表对各种类型的任务采取的不同的解决问题的思路， 大多数语言只支持一种编程范式，当然也有些语言可以同时支持多种编程范式。 两种最重要的编程范式分别是面向过程编程和面向对象编程。

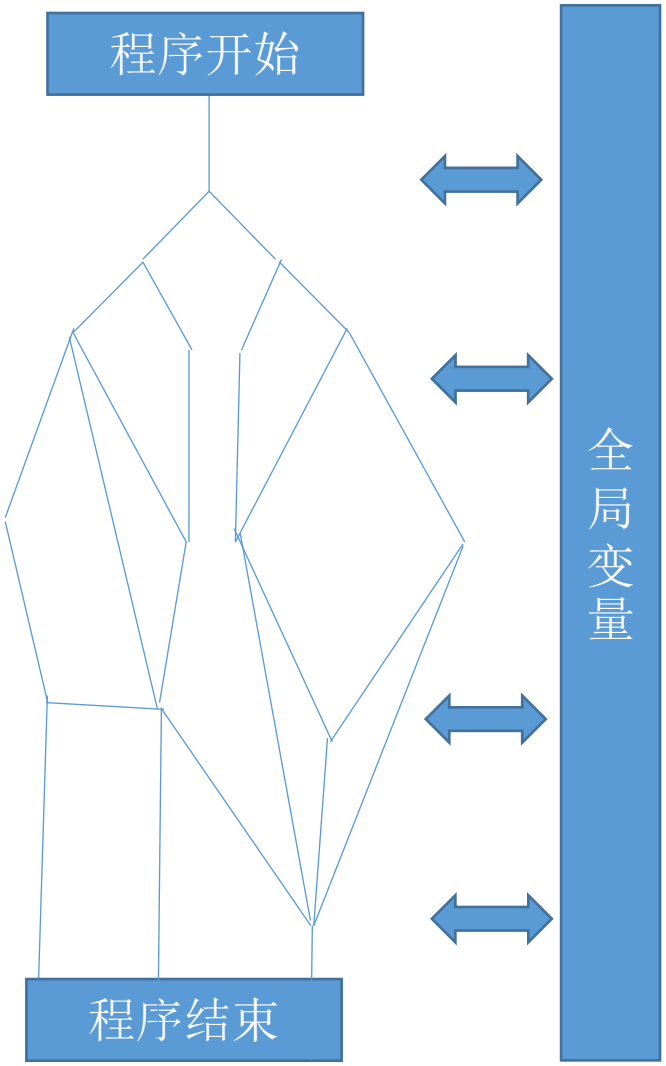
面向过程编程(Procedural Programming)

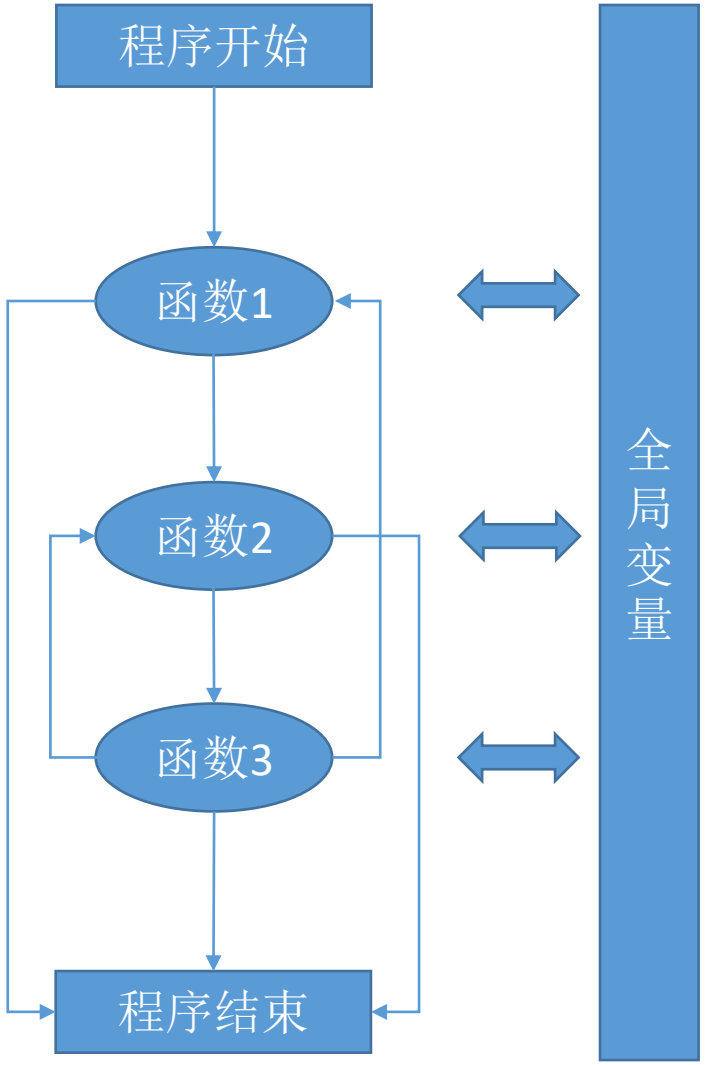
- 适用于处理简单的一次性任务，一个大问题分解成很多个小问题或子过程，这些子过程再执行的过程再继续分解直到小问题足够简单到可以在一个小步骤范围内解决。
- 典型代表：工厂流水线。
- 变量与数据分离。
- 个人视角，想做某件事件，每一步都要通过程序定义出来，写死了，在这个程序里，你只被设定做一件事件，新增功能，或修改原有功能，将导致整个程序的逻辑都得更改，用面向过程的方式写代码，那你care的就是整个事情的执行过程。

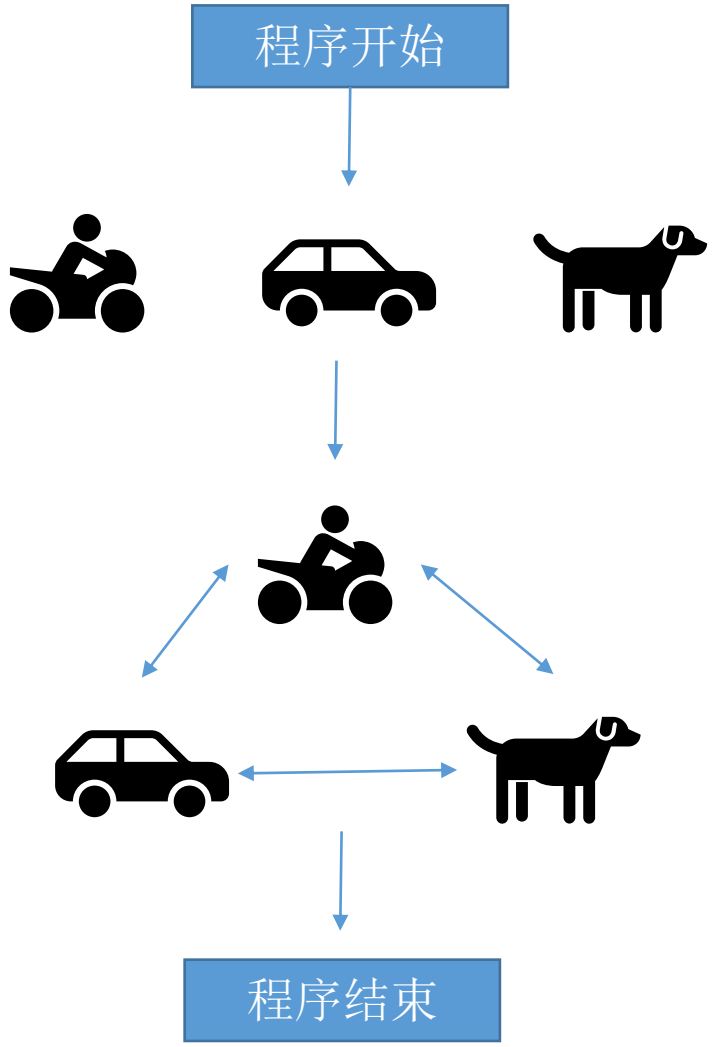
面向对象编程(Object Oriented Programing)

- 处理需要不断迭代、维护或交互的复杂任务。先把世界按物种、样貌、有无生命等各种维度分类，然后给每类东西建模型，再让其在不脱离你模型定义的框架下，自我繁衍、交互、发展。
- 典型代表：大型网游。
- 变量与方法结合。
- 上帝视角，你现在要创世纪，把这么多人、动物、山河造出来，先造模子（定义类），再一个个复制（类实例化）模子定义了人这个物种所具备的所有特征（属性、行为）。

编程范式







面向过程与面向对象的区别

面向过程	面向对象
数据与功能是分离。 会大量使用全局变量。 ■ 命名冲突问题。 ■ 并发冲突问题。	将数据与专门操作该数据的方法整合到一起。
简单易懂，适用于处理简单逻辑	各种封装继承，需要通盘阅读，支持处理复杂逻辑
可扩展性弱	可扩展性强

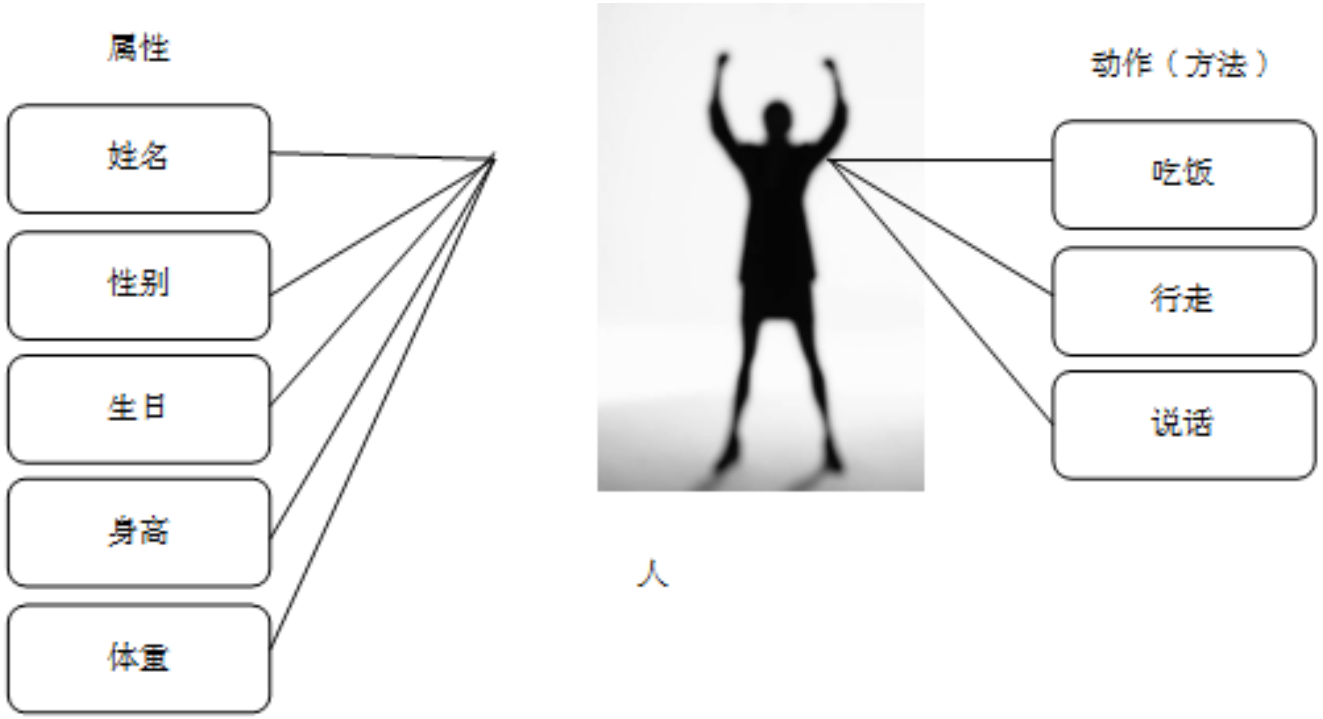
类的概念

面向对象编程

名词	解释
类	对一类拥有相同属性的对象的抽象、蓝图、原型、模板。在类中定义了这些对象的都具备的属性（variable）、共同的方法(method)
属性	把属于同一类对象的特征用程序来描述，叫做属性，以人类为例，年龄、身高、性别、姓名等都叫做属性，一个类中，可以有多个属性。
方法	人类还能做好多事情，比如说话、走路、吃饭等，相比较于属性是名词，说话、走路是动词，这些动词用程序来描述就叫做方法。
实例(对象)	一个对象即是一个类的实例化后实例，一个类必须经过实例化后方可在程序中调用，一个类可以实例化多个对象，每个对象亦可以有不同的属性，就像人类是指所有人，每个人是指具体的对象，人与人之前有共性，亦有不同
实例化	把一个类转变为一个对象的过程就叫实例化

面向对象编程-基本理念

面向对象编程是Python采用的基本编程思想，它可以将属性和代码集成在一起，定义为类，从而使程序设计更加简单、规范、有条理。本章将介绍如何在Python中使用类和对象。



在日常生活中，要描述一个事务，既要说明它的属性，也要说明它所能进行的操作。

面向对象编程-类的声明

在Python中，可以使用class关键字来声明一个类，其基本语法如下：

```
class 类名:
        成员函数
```

同样，Python使用缩进标识类的定义代码。

定义一个类Person

```
class Person:
        def SayHello(self):
                print("Hello!")
```

- 在成员函数SayHello()中有一个参数self。这也是类的成员函数（方法）与普通函数的主要区别。
- 类的成员函数必须有一个参数self，而且位于参数列表的开头。self就代表类的实例（对象）自身，可以使用self引用类的属性和成员函数。

在类Person中，定义了一个成员函数SayHello，用于输出字符串"Hello!"

面向对象编程-定义对象

- 对象是类的实例。只有定义了具体的对象，才能使用类。
- Python创建对象的方法如下：

  对象名 = 类名()
- 例如，下面的代码定义了一个类Person的对象p：p = Person()

  p实际相当于一个变量，可以使用它来访问类的成员变量和成员函数。

```
class Person:
        def SayHello(self):
                print("Hello!")
p = Person()
p.SayHello()
```

程序定义了类Person的一个对象p，然后使用它来调用类Person的成员函数SayHello()，运行结果如下：

```
Hello!
```

在类定义中，可以定义成员变量并同时对其赋初始值。

【例4-3】定义一个类MyString，定义成员变量str，并同时对其赋初始值。

```
class MyString:
        str = "MyString"
#在类的成员函数中使用self引用成员变量
        def output(self):
                print(self.str)
s = MyString()
s. output()
```

面向对象编程-成员变量

Python使用下划线作为变量前缀和后缀来指定特殊变量，规则如下：

- __xxx__表示系统定义名字。
- __xxx表示类中的私有变量名。

类的成员变量可以分为两种情况，一种是公有变量，一种是私有变量。公有变量可以在类的外部访问，它是类与用户之间交流的接口。用户可以通过公有变量向类中传递数据，也可以通过公有变量获取类中的数据。在类的外部无法访问私有变量，从而保证类的设计思想和内部结构并不完全对外公开。在Python中除了__xxx格式的成员变量外，其他的成员变量都是公有变量。

面向对象编程-构造函数

构造函数是类的一个特殊函数，它拥有一个固定的名称，即__init__（注意，函数名是以两个下划线开头和两个下划线结束的）。当创建类的对象实例时系统会自动调用构造函数，通过构造函数对类进行初始化操作。

面向对象编程-构造函数

在构造函数中，程序对公有变量str设置了初始值。可以在构造函数中使用参数，通常使用参数来设置成员变量（特别是私有变量）的值，举例：在MyString类中使用构造函数的实例。

```
class MyString:
        def __init__(self):
                self.str = "MyString"
        def output(self):
                print(self.str)
s = MyString()
s.output()
```

再例：在类UserInfo中使用带参数的构造函数。

```
class UserInfo:
        def __init__(self,name,pwd):
                self.username = name #公有变量
                self._pwd = pwd  #私有变量
        def output(self):
                print("用户：" + self.username + "\n密码：" + self._pwd)
u = UserInfo("admin","123456")
u.output()  # <-> 等价于 Userinfo.output(u)
```

面向对象编程-析构函数

- Python析构函数有一个固定的名称，即__del__。通常在析构函数中释放类所占用的资源。
- 使用del语句可以删除一个对象。释放它所占用的资源。
- 在实例对象被回收时将调用析构函数。

使用析构函数的一个实例。

```
class MyString:
        def __init__(self): #构造函数
                self.str = "MyString"
        def __del__(self):  #析构函数
                print("byebye~")
        def output(self):
                print(self.str)
s = MyString()
s.output()
del s  #删除对象
```

类的定义

构造方法

实例化时给实例一些初始化参数，或执行一些其它的初始化工作，只要一实例化，就会自动执行。

普通方法

定义类的一些正常功能，比如人这个类， 可以说话、走路、吃饭等，每个方法其实想当于一个功能或动作

析构方法

实例在内存中被删除时，会自动执行这个方法。

注意：绑定到对象的方法的这种自动传值的特征，决定了在类中定义的函数都要默认写一个参数self，self可以是任意名字，但是约定俗成地写出self。

```
class Person(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def talk(self):
        print("Hello, my name is %s, I'm %s years old!" %
(self.name, self.age))

    def __del__(self):
        print("running del method, this person must be
died.")


p = Person("person1", 22)
p.talk()

del p

print('--end program--')
```

课堂练习（1） 人狗大战

你现在是一家游戏公司的开发人员，现在需要你开发一款叫做<人狗大战>的游戏：

- 至少需要2个角色，一个是人， 一个是狗。
- 不同的角色有不同的属性。
- 人和狗都有不同的技能，比如人拿棍打狗， 狗可以咬人。

怎么描述这种不同的角色和他们的功能呢？

		人	狗
属性	姓名 name	amao	agou
	生命值 life_val	1000	100
	攻击能力 attack_val	4	10
方法	攻击	attack	bite

课堂练习（1） 人狗大战

1、先生成Person对象（与c++程序对比）：

```python
class Person(object):
    def __init__(self, name, life_value, attack_value):
        self.name = name
        self.life_value = life_value
        self.attack_value = attack_value
    def talk(self):
        print("Hello, my name is %s, My life value is %s and
attack value is %s!"
              % (self.name, self.life_value, self.attack_value))
    def attack(self, dog):
        dog.life_value -= self.attack_value
        print("Person %s attack Dog %s remain life: %s" %
(self.name, dog.name, dog.life_value))
```

12行

```
class Person(SuperClass) {
public:
        Person(PCSTR name, float attack_value, float life_value) {
                strcpy(m_szName, name);
                m_fAttackVal = attack_value;
                m_fLifeVal = life_value;
        }
        void Talk() {
                printf("hello my name is %s attack is %f life is %f\n",
                m_szName, m_fAttackVal, m_fLifeVal);
        }
        void Attack(Dog & dog) {
                dog.m_fLifeVal -= m_fAttackVal
                printf("%s attack %s remain %f\n",
                m_szName, dog.m_szName, dog.m_fLifeVal);
        }
private:
        char m_szName[64];
        float m_fAttackVal;
        float m_fLifeVal
};
```

21行

课堂练习（1） 人狗大战

1、实例化对象

```
class Person(object):
    def __init__(self, name, life_value, attack_value):
        self.name = name
        self.life_value = life_value
        self.attack_value = attack_value
    def talk(self):
        print("Hello, my name is %s, My life value is %s and
attack value is %s!"
              % (self.name, self.life_value, self.attack_value))
    def attack(self, dog):
        dog.life_value -= self.attack_value
        print("Person %s attack Dog %s remain life: %s" %
(self.name, dog.name, dog.life_value))
p = Person("person1", 1000, 100)
p.talk()
```

说明：

- 定义类头：class 类名(父类)
- 类方法： def method():
- __init__()是默认构造函数，类实例化过程中自动执行。
- 定义时所有方法第一个参数必须为self，pycharm会自动补齐。
- 访问类属性：self.property = …
- 与c++不同，不需要显示定义成员变量。
- p.talk() == Peason.talk(p)，解释器帮你做了。
- 不同的实例，拥有不同的内存空间，相互独立。

课堂练习（1） 人狗大战

2、实例化Dog对象

```
class Dog(object):

    def __init__(self, name, life_value, attack_value):
        self.name = name
        self.life_value = life_value
        self.attack_value = attack_value

    def bite(self, person):
        person.life_value -= self.attack_value
        print("Dog %s bite Person %s remain life: %s" %
(self.name, person.name, person.life_value))
```

3、人狗大站

```
p = Person("person1", 1000, 100)
print(p.name, p.life_value, p.attack_value)
p.talk()  # 注意这里调用并未传递参数

dog = Dog("dog1", 400, 300)
print(dog.name, dog.life_value, dog.attack_value)

p.attack(dog)
dog.bite(p)
```

课堂练习（1）

1.1 Person对象：

- 属性： name, weight
- 方法：
  - run(), 跑一次，体重减0.5公斤。
  - eat(), 跑一次，体重加1.0公斤。
- 小明体重75公斤，跑10次，吃5次后，体重是多少？

1.2 士兵与枪

- Soldier:
  - 属性： name, gun
  - 方法： fire()、add()
- Gun:
  - 属性： name, bullets（子弹数量）
  - 方法： add(), shoot()
- 士兵许三多有一把AK47, 填装10颗子弹后，开火5次，问最终子弹数量。

课堂练习（1）

1.3　房屋与家具

- House:
  - ◆ 属性： 名称（name），户型(style), 总面积(area)、家具列表(items)
  - ◆ 方法： 添加家具：addItem(item)，打印房屋信息, print(), 输出户型、总面积、剩余面积、家具列表。
- HouseItem:
  - ◆ 属性： name, 面积（area）
  - ◆ 方法：打印，print(), 输出家具名称、面积。
- 新房屋10平米，没有任何家具，添加床(bed，4平米)、柜子(chest, 2平米)， 餐桌(table， 1.5平米)后，打印房屋信息。

1.4 人(Person)狗(Dog)大战：

- 定义人、狗两个类。
- 实例化p1,d1两个对象。
- 回合制游戏，每1回合， p1和 d1对打一次，每一次掉血量是1个随机数，上限是其攻击力。
- 循环对打，一直到某一方死亡。
- 宣布游戏结果。

课堂练习（1）

1.5 选课系统

- 编写一个课程类(Course)
  - 包含属性：名称(name, string)，教师列表（teachers, list），学生列表（students，list)，成绩（scores，dict）
  - 主要方法：添加/删除学生， add_student/drop_student，添加/删除教师， add_teacher/drop_teacher，添加/查找某位同学的分数：set_score/get_score
- 编写一个学生类(Student)，
  - 包含属性：姓名(name , string)，性别(sex, int) ，学号(student_id , int)，已选修课程(courses，列表)
  - 主要方法： 选修/退出某课程 select_course/ drop_course，查看自己某个课程的成绩, get_score
- 编写一个教师类(Teacher)：
  - 包含属性：姓名(name , string)，性别(sex , int) ，拥有的课程(courses，列表)
  - 主要方法：选修/退出某课程 select_course/ drop_course，给某个课程和某个课程打成绩, set_score。
- 完成如下操作：
  - 定义10个学生，3个教师、5门课程，某一名学生选择某一门课程，某一名教师选择某一门课程，教师打分，学生查询分数。

本节重点：

- 继承
- 多态
- 封装

面向对象编程-基本概念

在面向对象程序设计中，将事务的属性和方法都包含在类中，而对象则是类的一个实例。如果将人定义为类的话，那么某个具体的人就是一个对象。不同的对象拥有不同的属性值。

（1）类（class）：具有相同或相似性质的对象的抽象就是类。因此，对象的抽象是类，类的具体化就是对象。例如，如果人类是一个类，则一个具体的人就是一个对象。

（2）对象（Object）：面向对象程序设计思想可以将一组数据和与这组数据有关操作组装在一起，形成一个实体，这个实体就是对象。

（3）封装：将数据和操作捆绑在一起，定义一个新类的过程就是封装。

（4）属性：也称为成员变量，把属于同一类对象的特征用程序来描述，叫做属性，以人类为例，年龄、身高、性别、姓名等都叫做属性，一个类中，可以有多个属性。

（4）方法：也称为成员函数，是指对象上的操作，作为类声明的一部分来定义。方法定义了对一个对象可以执行的操作。

（5）构造函数：一种成员函数，用来在创建对象时初始化对象。构造函数一般与它所属的类完全同名。

（6）析构函数：析构函数与构造函数相反，当对象脱离其作用域时（例如对象所在的函数已调用完毕），系统自动执行析构函数。析构函数往往用来做“清理善后”的工作。

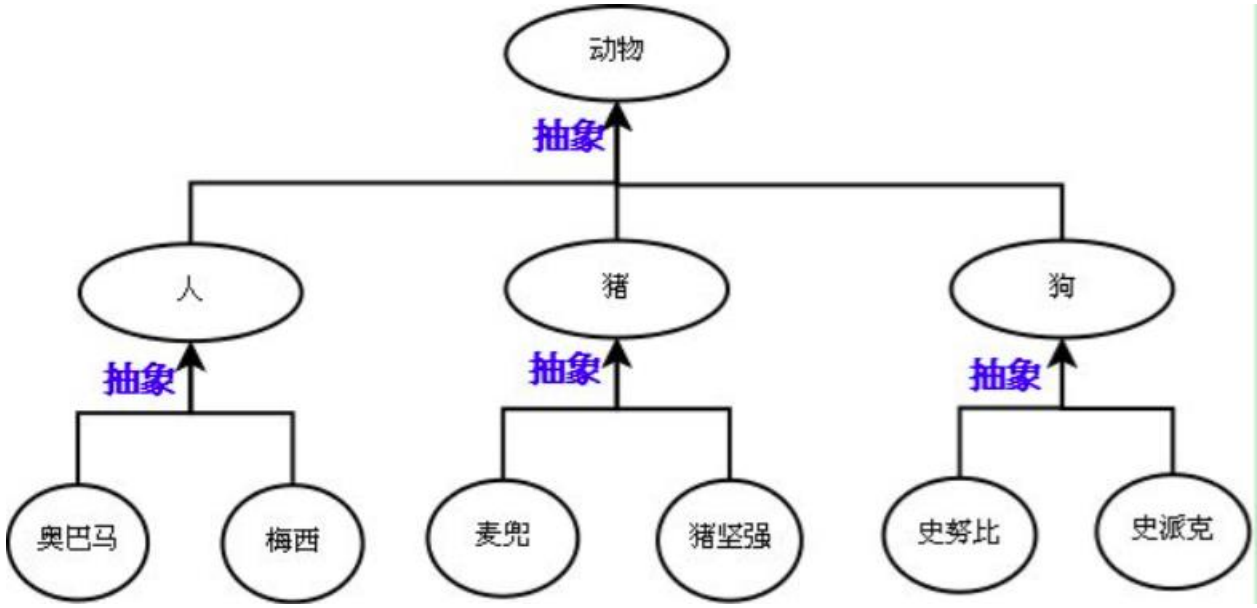
面向对象3大特性之一：继承、派生与组合

继承指的是类与类之间的关系，是一种什么“是”什么的关系，继承的功能之一就是用来解决代码重用问题。继承是一种创建新类的方式，在python中，新建的类可以继承一个或多个父类，父类又可以成为基类或超类，新建的类称为派生类或子类。

```
class ParentClass1: #定义父类
    pass

class ParentClass2: #定义父类
    pass

class SubClass1(ParentClass1):
#单继承，基类是ParentClass1，派生类是SubClass
    pass

class SubClass2(ParentClass1,ParentClass2):
#python支持多继承，用逗号分隔开多个继承的类
    pass
```

提示：python的类会默认继承object类，object是所有python类的基类，它提供了一些常见方法的实现。

分析的过程：实例 -》抽象 -》 对象



基于抽象后的结果，通过继承的方式，利用编程语言来表达出抽象的结构。

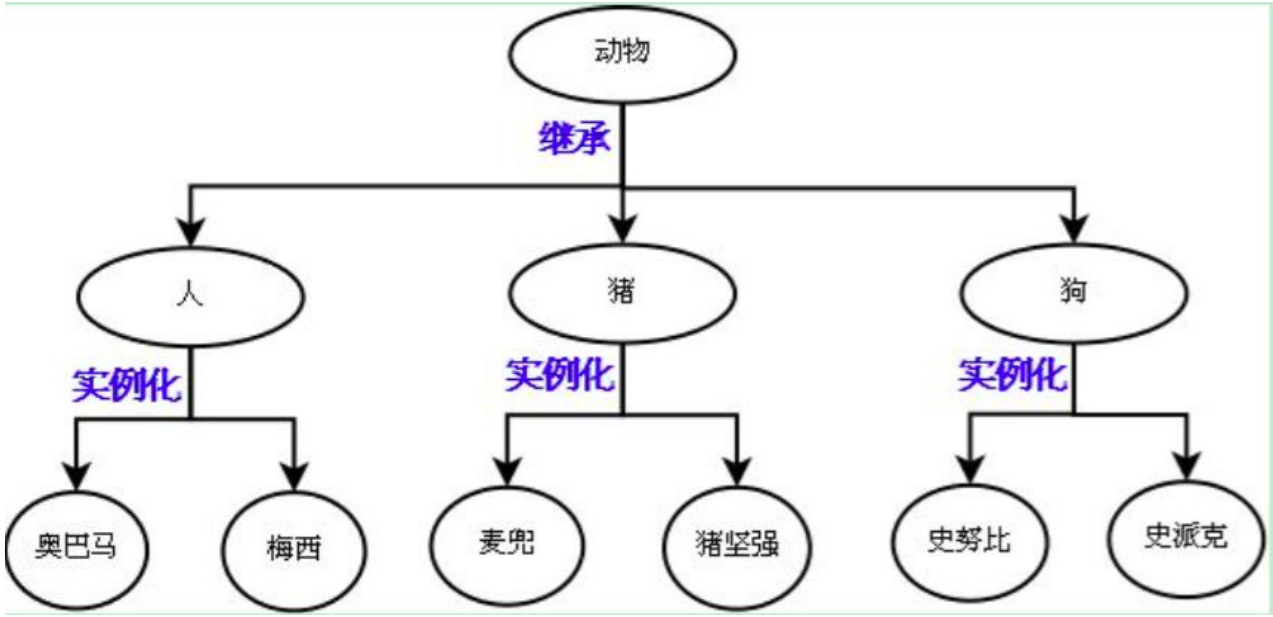
面向对象3大特性之1：继承、派生与组合

继承指的是类与类之间的关系，是一种什么“是”什么的关系，继承的功能之一就是用来解决代码重用问题。继承是一种创建新类的方式，在python中，新建的类可以继承一个或多个父类，父类又可以成为基类或超类，新建的类称为派生类或子类。

```
class ParentClass1: #定义父类
    pass

class ParentClass2: #定义父类
    pass

class SubClass1(ParentClass1):
#单继承，基类是ParentClass1，派生类是SubClass
    pass

class SubClass2(ParentClass1,ParentClass2):
#python支持多继承，用逗号分隔开多个继承的类
    pass
```

提示：python的类会默认继承object类，object是所有python类的基类，它提供了一些常见方法的实现。

实现的过程：对象 -》编程 -》 实例化



基于抽象后的结果，通过继承的方式，利用编程语言来表达出抽象的结构。

面向对象3大特性之1：继承、派生与组合

如果我们定义了一个类A，然后又想新建立另外一个类B，但是类B的大部分内容与类A的相同时，通过继承的方式新建类B，让B继承A，B会‘遗传’A的所有属性(数据属性和函数属性)，实现代码重用

```
class Human:
    def __init__(self, name, attack_value,
life_value):
        self.name = name
        self.attack_value = attack_value
        self.life_value = life_value

    def move_forward(self):
        print('%s move forward' % self.name)
    def move_backward(self):
        print('%s move backward' % self.name)
    def attack(self, enemy):
        enemy.life_value -= self.attack_value
```

```
class Male(Human):
    pass
class Female(Human):
    pass
male = Male('男生', 100, 300)
female = Female('女生', 57, 200)
male.attack(female)
female.attack(male)
```

提示：用已经有的类建立一个新的类，这样就重用了已经有的软件中的一部分设置大部分，大大节省了编程工作量，这就是常说的软件重用，不仅可以重用自己的类，也可以继承别人的，比如标准库，来定制新的数据类型。

面向对象3大特性之1：继承、派生与组合

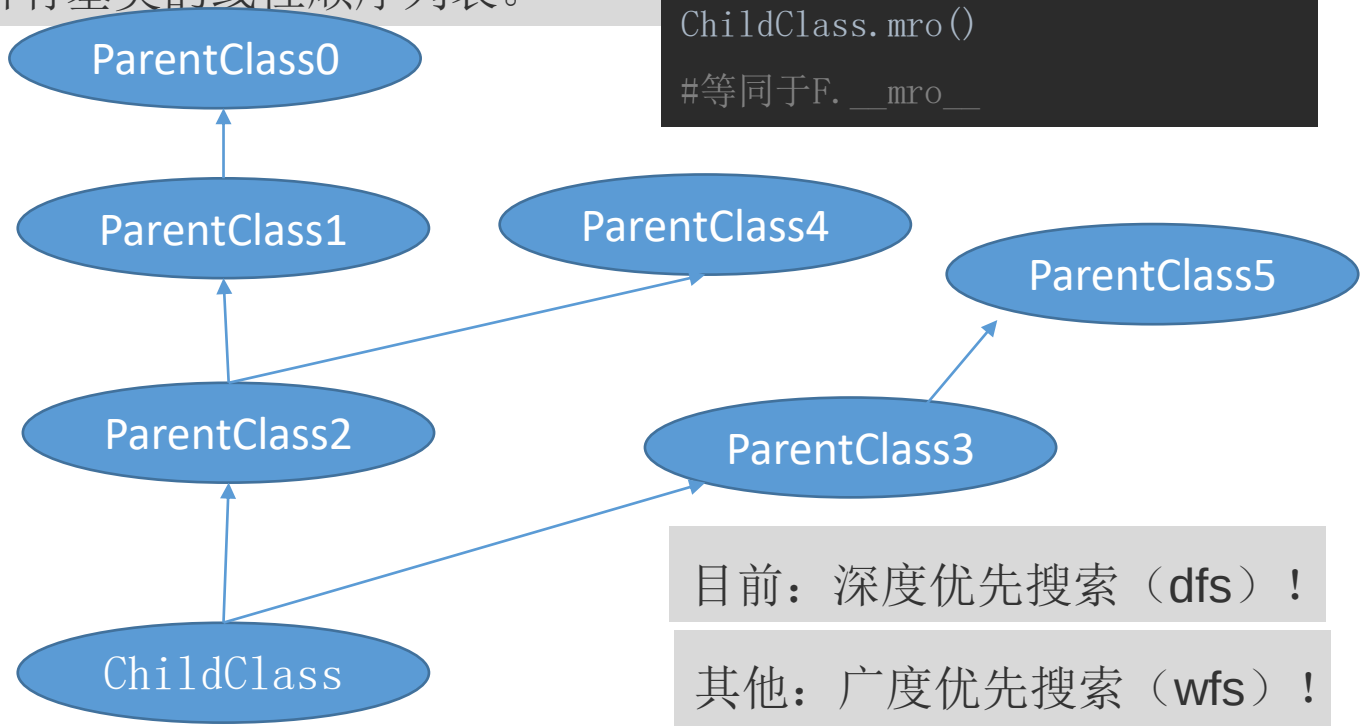
1、子类里是否可以定义新的属性/ 方法？

2、子类里定义属性名称/方法名称能否与父类重复？重复了用哪个？

```python
class ParentClass1:
    def __init__(self):
        pass
    def talk(self):
        print("ParentClass1 talk....")

class ParentClass2:
    def __init__(self):
        pass
    def talk(self):
        print("ParentClass2 talk....")

class ParentClass3(ParentClass1):
    def __init__(self):
        pass
    def talk(self):
        print("ParentClass3 talk....")

class ChildClass(ParentClass3, ParentClass2):
    def __init__(self):
        pass
    def talk(self):
        print("ChildClass talk....")

child = ChildClass()

child.talk()
```

python到底是如何实现继承的，对于你定义的每一个类，python会计算出一个方法解析顺序(MRO)列表，这个MRO列表就是一个简单的所有基类的线性顺序列表。

```python
ChildClass.mro()
#等同于F.__mro__
```



目前：深度优先搜索（dfs）！

其他：广度优先搜索（wfs）！

面向对象3大特性之1：继承、派生与组合

3、子类里如何调用父类的属性/方法

4、子类初始化时，如何对父类元素进行赋值？(super()方法)

```
class ParentClass1:
    def __init__(self):
        pass
    def talk(self):
        print("ParentClass1 talk....")
class ParentClass2:
    def __init__(self):
        pass
    def talk(self):
        print("ParentClass2 talk....")
class ParentClass3(ParentClass1):
    def __init__(self):
        pass
    def talk(self):
        print("ParentClass3 talk....")
class ChildClass(ParentClass3, ParentClass2):
    def __init__(self):
        pass
    def talk(self):
        print("ChildClass talk....")
```

方式1：指名道姓，即父类名.父类方法()

```
ParentClass3.talk(child)
ParentClass1.talk(child)
ParentClass2.talk(child)
ChildClass.talk(child)
```

方式2：super()

```
class ChildClass(ParentClass3, ParentClass2):
    def __init__(self):
        pass

    def talk(self):
        super().talk()
        super(ParentClass3, self).talk()
        super(ParentClass2, self).talk()
        super(ParentClass1, self).talk()
        print("ChildClass talk....")
```

方式1是跟继承没有关系的，而方式2的super()是依赖于继承的，会按照mro继续往后查找。

面向对象3大特性之1：继承、派生与组合

派生：python为大家提供了标准数据类型，以及丰富的内置方法，其实在很多场景下我们都需要基于标准数据类型来定制我们自己的数据类型，新增/改写方法，这就用到了我们刚学的继承/派生知识（其他的标准类型均可以通过下面的方式进行二次加工）

```
class List(list):
#继承list所有的属性，也可以派生出自己新的，
#比如append和mid
    def append(self, p_object):
        '派生自己的append：加上类型检查'
        if not isinstance(p_object,int):
            raise TypeError('must be int')
        super().append(p_object)
    def mid(self):
        '新增自己的属性'
        index=len(self)//2
        return self[index]
```

```
l = List([1, 2, 3, 4])
print(l)
l.append(5)
print(l)
# l.append('11') #报错

print(l.mid)

# 其余的方法都继承list的
l.insert(0, -123)
print(l)
l.clear()
print(l)
```

```
class List(list):
    def __init__(self, item, tag=False):
        super().__init__(item)
        self.tag = tag

    def append(self, p_object):
        if not isinstance(p_object, str):
            raise TypeError
        super().append(p_object)

    def clear(self):
        if not self.tag:
            raise PermissionError
        super().clear()
```

clear加权限限制

面向对象3大特性之1：继承、派生与组合

用组合的方式建立了类与类之间的关系，它是一种‘有’的关系，比如：教师➔课程➔学生

```
class People:
    def __init__(self,name,age,sex):
        self.name=name
        self.age=age
        self.sex=sex
class Course:
    def __init__(self,name):
        self.name=name
    def tell_info(self):
        print(self.name,self.period)
class Teacher(People):
    def __init__(self,name,age,sex,title):
        People.__init__(self,name,age,sex)
        self.job_title=title
        self.course=[]
class Student(People):
    def __init__(self,name,age,sex):
        People.__init__(self,name,age,sex)
        self.course=[]
```

```
t1 = Teacher('t1', 18, 'male', '电机系教师')
s1 = Student( 's1', 18, 'female')
s2 = Student( 's2', 18, 'female')

python = Course( 'python' ,  '3mons' )
linux = Course( 'python' ,  '3mons' )

# 为老师t1和学生s1添加课程
t1.course.append(python)
s1.course.append(linux)

# 为课程添加学生s1
python.students.append(s1)
python.students.append(s2)
linux.students.append(s1)
# 使用
for obj in t1.course:
    obj.tell_info()
```

面向对象3大特性之1：继承、派生与组合

- 继承是面向对象程序设计思想的重要机制。类可以继承其他类的内容，包括成员变量和成员函数。而从同一个类中继承得到的子类也具有多态性，即相同的函数名在不同子类中有不同的实现。就如同子女会从父母那里继承到人类共有的特性，而子女也具有自己的特性。
- 通过继承机制，用户可以很方便地继承其他类的工作成果。如果有一个设计完成的类A，可以从其派生出一个子类B，类B拥有类A的所有属性和函数，这个过程叫做继承。类A被称为类B的父类。
- 使用面向对象程序设计思想可以通过对类的继承实现应用程序的层次化设计。类的继承关系是树状的，从一个根类中可以派生出多个子类，而子类还可以派生出其他子类，以此类推。每个子类都可以从父类中继承成员变量和成员函数，实际上相当于继承了一套程序设计框架。

可以在定义类时指定其父类。例如，存在一个类A，定义代码如下：

```
class A:
    def __init__(self,property): #构造函数
        self.propertyA = property  #类A的成员变量
    def functionA():                          #类A的成员函数
```

从类A派生一个类B，代码如下：

```
class B(A):
    propertyB   #类B的成员变量
    def functionB():                #类B的成员函数
```

从类B中可以访问到类A中的成员变量和成员函数，例如：

```
objB = B()     #定义一个类B的对象objB
print(objB.propertyA)      #访问类A的成员变量
objB.functionA()  #访问类A的成员函数
```

因为类B是从类A派生出来的，所以它继承了类A的属性和方法。

课堂练习（2）

2.1 定一个动物类Animal，再定义两个子类：Human、Dog

- Animal，属性： id，name, life, power, weight, pos， 方法： move(self, pos)
- Human，新增加属性： father, mother，新增加方法: shout(self, info)， attack(self, dog)
- Dog， 新增加属性：price, 新增加方法: bark(self, info) ， bite(self, human)
- 生成两个队列， 各有10名成员，随机碰撞， 碰到则互相进攻，血条降为0，则死亡，最终剩余的一队，宣布胜利。

2.2 选课系统（教师和学生从Person中继承）

- 人员(Person)、教师(Teacher)、学生(Student)、课程(Course)
- 学生选课（学生课程列表里增加课程， 课程学生列表里增加学生）
- 教师选择授课（教师课程列表里增加课程， 课程教师里增加教师）
- 教师给学生打分

2.3 从list，派生出一个新的类IntList，要求:

- 重载已有的append()函数， 支持输入整形数，不支持其他类型。
- 新增加mid()函数，返回中间元素的结果。
- 不允许调用clear()函数。

面向对象3大特性之2：多态

- 所谓多态，指抽象类中定义的一个方法，可以在其子类中重新实现，不同子类中的实现方法也不相同。

首先创建一个抽象类Shape，它定义了一个画图类的基本框架，代码如下：

```
class Shape(object):
    def __init__(self):
        self.color = 'black'  #默认使用黑色
    def draw(self):
        pass
```

再从类Shape中派生出画直线的类line

```
class line(Shape):
    def __init__(self,x1,y1,x2,y2):  #定义起止坐标值
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
    def draw(self):
        print("Draw Line:(%d,%d,%d,%d)"  %(self.x1,self.y1,self.x2,self.y2))
```

创建类Shape的子类circle

```
class circle (Shape):
    def __init__(self, x, y, r):  #定义圆心坐标和半径
        self.x = x
        self.y = y
        self.r = r
    def draw(self):
        print("Draw Circle: (%d, %d, %d)"  %(self.x, self.y, self.r))
```

从上可以看到，在不同的子类中，抽象方法draw()有不同的实现，这就是类的多态。

面向对象3大特性之2：多态

- 所谓多态，指抽象类中定义的一个方法，可以在其子类中重新实现，不同子类中的实现方法也不相同。
- 从上可以看到，在不同的子类中，抽象方法draw()有不同的实现。

定义一个类circle的对象c，然后调用draw()方法，代码如下：

```
c = circle(10,10,5)
c.draw()
```

定义一个类line的对象l，然后调用draw()函数，代码如下：

```
l = line(10,10,20,20)
l.draw()
```

输出结果如下：

```
Draw Circle: (10,10,5)
Draw Line: (10,10,20,20)
```

面向对象3大特性之2：多态

多态性是指在不考虑实例类型的情况下使用实例，用于动态确定对象类型。

```
class Animal(){
public:
   Animal() {};
   virtual void run() {
      printf("Animal is running!\n");
   }
}
class Dog(Animal){
public:
   Dog() {};
   virtual void run() {
      printf("Dog is running!\n");
   }
}
class Pig(){
public:
   Pig() {};
   virtual void run() {
      printf("Pig is running!\n");
   }
}
Animal * get_anminal(PCSTR name){
   if (!strcmp(name, "pig")){
      return new Pig();
   }
   if (!strcmp(name, "dog")) {
      return new Dog();
   }
   return new Animal();
}
Animal * p3 = get_p("pig")
p3->run();
Animal * p4 = get_p("dog")
p4->run();
```

```
class Anminal:
    def run(self):
        print("Anminal is running...")

class Pig(Anminal):
    def run(self):
        print("Pig is running...")

class Dog(Anminal):
    def run(self):
        print("Dog is running...")

def get_anminal(name):
    if name == 'pig':
        return Pig()
    if name == 'dog':
        return Dog()
    return Anminal()

p3 = get_p("pig")
p3.run()
p4 = get_p("dog")
p4.run()
```

优点：

1.增加了程序的灵活性，不论对象千变万化，使用者都是同一种形式去调用，如func(animal)

2.增加了程序的可扩展性, 通过继承animal类创建了一个新的类，使用者无需更改自己的代码，还是用func(animal)去调用。

面向对象3大特性之2：多态

多态性是指在不考虑实例类型的情况下使用实例。

```
class Anminal:
    def run(self):
        print("Anminal is running...")

class Pig(Anminal):
    def run(self):
        print("Pig is running...")

class Dog(Anminal):
    def run(self):
        print("Dog is running...")

def get_anminal(name):
    if name == 'pig':
        return Pig()
    if name == 'dog':
        return Dog()
    return Anminal()

p3 = get_p("pig")
p3.run()
p4 = get_p("dog")
p4.run()
```

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

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

def get_anminal(name):
    if name == 'pig':
        return Pig()
    if name == 'dog':
        return Dog()
    return Anminal()

p3 = get_p("pig")
p3.run()
p4 = get_p("dog")
p4.run()
```

鸭子类型

如果想编写现有对象的自定义版本:

- 继承该对象
- 创建一个外观和行为像，但与它无任何关系的全新对象，后者通常用于保存程序组件的松耦合度。

```
#str,list,tuple都是序列类型
s=str('hello')
l=list([1,2,3])
t=tuple((4,5,6))

#我们可以在不考虑三者类型的前提下使用s,l,t
s.__len__()
l.__len__()
t.__len__()
```

面向对象3大特性之2：多态

- Python崇尚鸭子类型，即‘如果看起来像、叫声像而且走起路来像鸭子，那么它就是鸭子’
- python程序员通常根据这种行为来编写程序。例如，如果想编写现有对象的自定义版本，可以继承该对象
- 也可以创建一个外观和行为像，但与它无任何关系的全新对象，后者通常用于保存程序组件的松耦合度。
- 例1：利用标准库中定义的各种‘与文件类似’的对象，尽管这些对象的工作方式像文件，但他们没有继承内置文件对象的方法
- 例2：序列类型有多种形态：字符串，列表，元组，但他们直接没有直接的继承关系

课堂练习（3）

3.1 定一个动物类Animal，再定义两个子类：Human、Cat、Dog

- 1、定义Animal类，至少包含一个属性和一个方法
- 2、定义Cat类和Dog类，使这两个类继承自Animal类
- 3、定义Human类，使人可以通过Animal喂食Cat类和Dog类的实例

3.2 几何形状

- 定义3个类： Rect, Circle, Square
- 实现两个函数功能的多态化：
  - 求解周长：perimeter()
  - 求解面积：area()
- 两种方式来生成对象：静态、动态。

面向对象3大特性之3：封装

- 把属性或方法隐藏起来，外部不能直接访问使用，以保证用户隐私和信息安全。

- C++/Java的做法： public / private/protect
- Python的做法： 属性 和方法 以“__”开头。

```
class TestClass:
    __A = 0
    def __init__(self, num):
        self.__num = num


t1 = TestClass(1)
# print(t1.__num)
print(t1.__dict__)

print(t1._TestClass__num)
print(TestClass.__dict__)
```

特点：

- 注意__dict__的用法。
- 类中__num只能在内部使用，引用的就是变形的结果。
- 在外部是无法通过__num这个名字访问到的。
- 在子类定义的__num不会覆盖在父类定义的__num，即双下滑线开头的属性在继承给子类时，子类是无法覆盖的。
- 父类的方法若不想被子类覆盖，可以将该方法设为私有的。
- 这种机制也并没有真正意义上限制我们从外部直接访问属性，知道了类名和属性名就可以拼出名字并访问（君子封装）。

面向对象3大特性之3：使用封装后数据的特殊方法： property

■ property是一种特殊的属性，访问它时会执行一段功能（函数）然后返回值

例一：BMI指数，(计算而来，并非方法)

成人的BMI数值：

体质指数（BMI）=体重（kg）÷身高^2

```
class People:
    def __init__(self,name,weight,height):
        self.name=name
        self.weight=weight
        self.height=height
    @property
    def bmi(self):
        return self.weight / (self.height**2)

p1=People('wangbin',85,1.75)
print(p1.bmi)
```

```
class Peson:

    def __init__(self, val):
        self.__name = val

    @property       # name = property(name)
    def name(self):
        return self.__name

    @name.setter    # name = name.setter(name())
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('%s must be str' % value)
        self.__name = value

    @name.delete    # name = name.delete(name())
    def name(self):
        raise TypeError('Can not delete')

f = Peson('wangbin')
print(f.name)
# f.name=10 #抛出异常'TypeError: 10 must be str'
del f.name  # 抛出异常'TypeError: Can not delete'
```

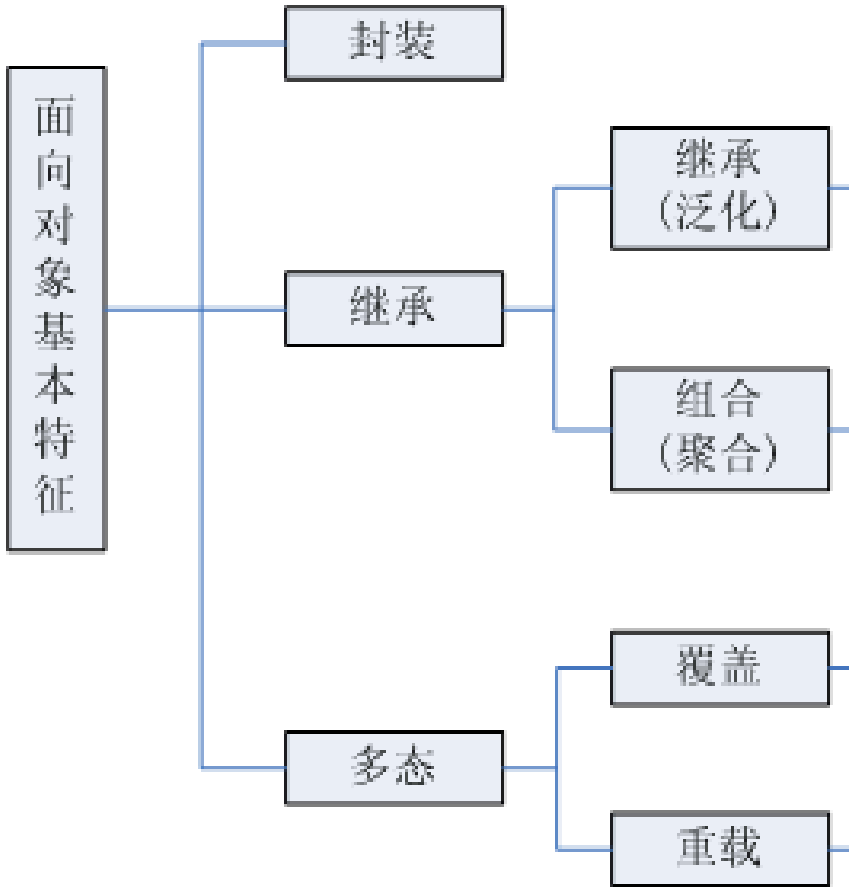
面向对象3大特性之3：封装

- 封装数据，对外提供操作接口，在接口中增加操作限制，以此完成对数据属性操作的严格控制。
- 封装方法：保护隐私性、隔离复杂性、提升安全性。

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

    def tell_info(self):
        print('姓名:%s,年龄:%s' % (self.__name,
self.__age))
    def set_info(self, name, age):
        if not isinstance(name, str):
            raise TypeError('姓名必须是字符串类型')
        if not isinstance(age, int):
            raise TypeError('年龄必须是整型')
        self.__name = name
        self.__age = age

t = Teacher('wangbin', 18)
t.tell_info()
t.set_info('wangbin', 19)
t.tell_info()
```

```
class ATM:
    def __card(self):
        print('插卡')

    def __auth(self):
        print('用户认证')

    def __input(self):
        print('输入取款金额')

    def __take_money(self):
        print('取款')

    def withdraw(self):
        self.__card()
        self.__auth()
        self.__input()
        self.__take_money()
a = ATM()
a.withdraw()
```

面向对象3大特性小结



- 继承，可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。
- 继承可以扩展已存在的代码模块（类）；它们的目的都是为了——代码重用；

- 封装，把客观事物封装成抽象类，并只让可信的类或者对象操作，对不可信的进行信息隐藏。
- 封装可以隐藏实现细节，使得代码模块化和调用安全；

- 多态，是允许你将父对象设置成为和一个或更多的他的子对象相等的技术。
- 多态保证使用“家谱”中任一类的实例的某一属性时的正确调用——接口重用！

OO开发范式大致为：划分对象（建模）→抽象类（定义）→将类组织成为层次化结构(继承和合成) →用类与实例进行设计和实现（coding逻辑）。

课堂练习（4）

4.1 定一个类（Person)：

- 把年龄和体重属性隐藏起来
- 通过property，确保name不允许非字符串式赋值。。。
- 通过property，确保输入每个月的salary值不允许非数字赋值
- 通过property，确保height不允许非数字赋值
- 通过property，确保age不允许非整形数字赋值
- 通过property，返回人体的BMI指标（体重千克数除以身高米数的平方得出的数字）
- 通过property，返回当年的平均月工资、年工资总和。

课堂练习（4）

4.2 定义MySQL类

- 要求：
- 对象有__id、__host、__port三个属性
- 在实例化时为每个对象生成id，id要保持唯一（提示：时间字符串的MD5 码值）
- 提供两种实例化方式（提示：使用**kwargs），
  - 方式一：用户传入host和port
  - 方式二：从配置文件中读取host和port进行实例化
- 为对象定制方法：
  - id()，返回 __id
  - host(),返回 __host
  - port() 返回 __port
  - save()能自动将对象信息序列化到文件中，文件名为id号，保存之前验证对象是否已经存在，若存在则抛出异常;

Python面向对象编程

-语法糖

本节重点：

- 属性绑定与方法绑定
- 内置函数与反射
- 解读“.”魔法
- 解读“[]”魔法
- 解读“for … in …”魔法
- 解读“with”魔法

面向对象编程-静态成员

静态变量和静态方法是类的静态成员，它们与普通的成员变量和成员方法不同，静态类变量和成员与具体的对象没有关系，而是只属于定义它们的类。

- Python不需要显式定义静态变量，任何公有变量都可以作为静态变量使用。访问静态变量的方法：类名.变量名
- 虽然也可以通过对象名访问静态变量，但是同一个变量，通过类名访问与通过对象名访问的实例不同，而且不互相干扰。

定义一个类Users，使用静态变量online_count记录当前在线的用户数量。

```
class Users:
    online_count = 0
    def __init__(self): #构造函数，创建对象时Users.online_count加1
        Users.online_count += 1
    def __del__(self): #析构函数，释放对象时Users.online_count减1
        Users.online_count -= 1
a = Users()  #创建Users对象a
a.online_count += 1
print(Users.online_count)
```

面向对象编程-静态方法

与静态变量相同，静态方法只属于定义它的类，而不属于任何一个具体的对象。静态方法具有如下特点：

（1）静态方法无需传入self参数，因此在静态方法中无法访问实例变量。

（2）在静态方法中不可以直接访问类的静态变量，但可以通过类名引用静态变量。

因为静态方法既无法访问实例变量，也不能直接访问类的静态变量，所以静态方法与定义它的类没有直接关系，而是起到了类似函数工具库的作用。

使用装饰符@staticmethod定义静态方法：

```
class 类名:
        @staticmethod
        def 静态方法名():
                方法体
```

可以通过对象名调用静态方法，也可以通过类名调用静态方法。而且这两种方法没有区别。

```
class MyClass: #定义类
    var1 = 'String 1'
    @staticmethod  #静态方法
    def staticmd():
        print("我是静态方法")
MyClass.staticmd()
c = MyClass()
c.staticmd()
```

程序中分别使用类和对象调用静态方法staticmd()，运行结果如下：

```
我是静态方法
我是静态方法
```

面向对象编程-类方法

类方法是Python的一个新概念。类方法具有如下特性：

（1）与静态方法一样，可以使用类名调用类方法。

（2）与静态方法一样，类成员方法也无法访问实例变量，但可以访问类的静态变量。

（3）类方法需传入代表本类的cls参数。

使用装饰符@classmethod定义类方法：

```
class 类名:
        @classmethod
        def 类方法名(cls):
                方法体
```

可以通过对象名调用类方法，也可以通过类名调用类方法。而且这两种方法没有什么区别。

类方法有一个参数cls，代表定义类方法的类，可以通过cls访问类的静态变量。

```
class MyClass:  #定义类
    val1 = 'String 1'
    def __init__(self):
        self.val2 = 'Value 2'
    @classmethod  #类方法
    def classmd(cls):
        print(cls.val1) #通过参数cls访问类的静态变量
MyClass.classmd()
c = MyClass()
c.classmd()
```

Python的类绑定：属性

由于Python是动态语言，类以及根据类创建的实例可以任意绑定属性以及方法。

- 类绑定属性可以直接在class中定义属性，这种属性是类属。
- 类的所有实例都可以访问类属性。

```
class Student(object):
    name = 'Student'

s1 = Student()  # 创建实例s
print(s1.name)
s2 = Student()  # 创建实例s
print(s2.name)
# 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性
print(Student.name)
# 打印类的name属性
```

- 相同名称的实例属性将覆盖类属性，删除实例属性后，实例将向上访问到类属性。实例绑定属性的方法有两种，一是通过类的self变量，二是直接给实例赋值。

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

s = Student('Bob')  # 方法一 通过类的self变量绑定属性
s.score = 90  # 方法二 直接赋值
```

```
class ParentClass:
    name = "aa"
    def __init__(self, name):
        ParentClass.name = "parent:" + name
    def show(self):
        print("ParentClass show ", self.name, "...")
class ClassTest(ParentClass):
    name = "aa"
    def __init__(self, name):
        super().__init__(name)
        self.name = name
    def parent(self):
        return super(ClassTest, self)

    def show(self):
        print("ClassTest show ", self.name, "...")

c = ClassTest("bb")
print(c.__dict__)
del c.name
print(c.__dict__)
print(c.name)
d = c.parent()
print(d.__dict__)
```

Python的类绑定：方法

一、类中定义的函数分成两大类：

1、绑定方法（绑定给谁，谁来调用就自动将它本身当作第一个参数传入

1）绑定到对象的方法：没有被任何装饰器装饰的方法。

为对象量身定制

对象.boud_method(),自动将对象当作第一个参数传入

（属于类的函数，类可以调用，但是必须按照函数的规则来，没有自动传值那么一说）

在使用时会将对象本身当做参数传给方法的第一个参数，是Python默认的绑定方法。

Python的类绑定：方法

一、类中定义的函数分成两大类：

1、绑定方法（绑定给谁，谁来调用就自动将它本身当作第一个参数传入

2）绑定到类的方法：用classmethod装饰器装饰的方法。

为类量身定制

类.boud_method(),自动将类当作第一个参数传入

（其实对象也可调用，但仍将类当作第一个参数传入）

在使用时会将类本身当做参数传给类方法的第一个参数（即便是对象来调用也会将类当作第一个参数传入），python为我们内置了函数classmethod来把类中的函数定义成类方法。

Python的类绑定：方法

一、类中定义的函数分成两大类：

1、绑定方法（绑定给谁，谁来调用就自动将它本身当作第一个参数传入

2）绑定到类的方法：用classmethod装饰器装饰的方法。

```
#settings.py
HOST='127.0.0.1'
PORT=3306
DB_PATH=r'C:\Users\Administrator\PycharmProjects\test\面向对

#test.py
import settings
class MySQL:
    def __init__(self,host,port):
        self.host=host
        self.port=port

    @classmethod
    def from_conf(cls):
        print(cls)
        return cls(settings.HOST,settings.PORT)

print(MySQL.from_conf) #<bound method MySQL.from_conf of <cl
conn=MySQL.from_conf()

conn.from_conf() #对象也可以调用，但是默认传的第一个参数仍然是类
```

classmehtod是给类用的，即绑定到类，类在使用时会将类本身当做参数传给类方法的第一个参数（即便是对象来调用也会将类当作第一个参数传入），python为我们内置了函数classmethod来把类中的函数定义成类方法

Python的类绑定：方法

一、类中定义的函数分成两大类

2、非绑定方法：用staticmethod装饰器装饰的方法

不与类或对象绑定，类和对象都可以调用，但是没有自动传值那么一说。就是一个普通工具而已

注：与绑定到对象方法区分开，在类中直接定义的函数，没有被任何装饰器装饰的，都是绑定到对象的方法，可不是普通函数，对象调用该方法会自动传值，而staticmethod装饰的方法，不管谁来调用，都没有自动传值一说。

```
import hashlib
import time
class MySQL:
    def __init__(self,host,port):
        self.id=self.create_id()
        self.host=host
        self.port=port
    @staticmethod
    def create_id(): #就是一个普通工具
        m=hashlib.md5(str(time.time()).encode('utf-8'))
        return m.hexdigest()


print(MySQL.create_id) #<function MySQL.create_id at 0x0000000001E6B9D8> #查看结果为普通函数
conn=MySQL('127.0.0.1',3306)
print(conn.create_id) #<function MySQL.create_id at 0x00000000026FB9D8> #查看结果为普通函数
```

在类内部用staticmethod装饰的函数即非绑定方法，就是普通函数，statimethod不与类或对象绑定，谁都可以调用，没有自动传值效果

Python的类绑定：方法

classmethod与staticmethod的对比

```
class MySQL:
    def __init__(self,host,port):
        self.host=host
        self.port=port

    @staticmethod
    def from_conf():
        return MySQL(settings.HOST,settings.PORT)

    # @classmethod #哪个类来调用,就将哪个类当做第一个参数传入
    # def from_conf(cls):
    #     return cls(settings.HOST,settings.PORT)

    def __str__(self):
        return '就不告诉你'

class Mariadb(MySQL):
    def __str__(self):
        return '<%s:%s>' %(self.host,self.port)


m=Mariadb.from_conf()
print(m) #我们的意图是想触发Mariadb.__str__,但是结果触发了MySQL.__str__的执行，打印就不告诉你:
```

课堂练习（1）：属性和方法的绑定

定义一个MySQL类

- 成员变量： id、host、port、schema、login_status
- 绑定到类的变量： login_status、login_conn_num
- 成员函数：get_id()、get_host()、get_port()、get_schema()
- 静态成员函数： create_id()，生成1个全局唯一的id
  - m = hashlib.md5(str(time.time()).encode('utf-8'))
  - return m.hexdigest()
- 绑定到类的函数： create_sql(host, port, schema)，返回1个实例对象，其id自动生成
- 登录/注销功能：
  - 增加login()函数， login()时:
    - 若self.login_status == False，则self.login_status =True, 全局login_status = True，同时login_conn_num += 1
    - 若self.login_status == True，则直接退出。
  - 增加logout()函数，logout()时：
    - 若已经登录，则self.login_status = False， login_conn_num -= 1，若全局链接数为0，则退出登录：login_status = False
    - 若没有登录，则返回。
  - 析构时，自动执行 logout() 函数

与类相关的内置函数

内置函数名称	功能		
isinstance	obj是否cls的实例	isinstance(obj, cls)	
issubclass	sub是否super的子类	issubclass(sub, super)	
hasattr	检测是否有某属性/方法	hasattr(obj, “name”)	重点关注 obj.__dict__ 类或对象的作用域。。
getattr	获取某属性/方法	getattr(obj, “name”)	
delattr	删除属性/方法	delattr(obj, “name”) (不存在则报错)	
setattr	设置某属性/方法	setattr(obj, “name”, …)	

python中的一切事物都是对象，通过字符串的形式动态操作（增删改查）对象的属性/方法，称为反射。

与类相关的内置函数-反射

反射的概念是由Smith在1982年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力（自省）。这一概念的提出很快引发了计算机科学领域关于应用反射性的研究。它首先被程序语言的设计领域所采用,并在Java等语言中取得巨大成功（ Java reflection 机制）：

- 反射优势：允许程序创建和控制任何类的对象(根据安全性限制)，无需提前硬编码目标类，特别适用于创建以非常普通的方式与对象协作的库。例如，使类和数据结构能按名称动态检索相关信息，并允许在运行着的程序中操作这些信息。
- 反射劣势：1是性能问题：用于字段和方法接入时反射要远慢于直接代码。2是使用反射会模糊程序内部实际要发生的事情。程序员无法通过源代码直观看到程序的运行逻辑，程序可读性差， 难于维护。

```python
class Peson:
    def __init__(self, name, addr):
        self.name = name
        self.addr = addr
    def run(self):
        print('%s is talking...' % self.name)

p1 = Peson('p1', '清华大学')

# 检测是否含有某属性
print(hasattr(p1, 'name'))
print(hasattr(p1, 'age'))

# 获取属性
n = getattr(p1, 'name')
print(n)
func = getattr(p1, 'run')
func()

# getattr(b1,'aaaaaaaa') #报错
print(getattr(p1, 'age', '不存在啊'))

# 设置属性
setattr(p1, 'age', 19.0)
setattr(p1, 'get_name', lambda self: '清华学生：' + self.name)
print(p1.__dict__)
print(p1.change_name(p1))

# 删除属性
delattr(p1, 'addr')
delattr(p1, 'change_name')
delattr(p1, 'change_name123')  # 不存在,则报错

print(p1.__dict__)
```

与类相关的内置函数-反射

作用1：
类属性/方法的反射

```
class Peson(object):

    staticField = "Tsinghua"

    def __init__(self):
        self.name = 'wangbin'

    def func(self):
        return 'func'

    @staticmethod
    def bar():
        return 'bar'

print(getattr(Peson,
'staticField'))
print(getattr(Peson, 'func'))
print(getattr(Peson, 'bar'))
```

作用2：
模块属性/方法的反射

```
def s1():
    print 's1'
def s2():
    print 's2'
this_module = sys.modules[__name__]
hasattr(this_module, 's1')
getattr(this_module, 's2')
```

作用3：
判断某属性/方法是否存在

```
# from module import FtpClient
f1 = FtpClient('192.168.1.1')
if hasattr(f1, 'get'):
    func_get = getattr(f1, 'get')
    func_get()
else:
    print('---->不存在此方法')
    print('处理其他的逻辑')
```

```
class Person:
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)

p1 = Person(name = 'abc', age = 12)
```

类的内置成员函数- attr相关

__setattr__,__delattr__,__getattr__三者的用法演示

```
class Foo:
    x = 1

    def __init__(self, y):
        self.y = y

    def __getattr__(self, item):
        print('----> from getattr:你找的属性不存在')

    def __setattr__(self, key, value):
        print('----> from setattr')
        # self.key=value #这就无限递归了,你好好想想
        # self.__dict__[key]=value #应该使用它

    def __delattr__(self, item):
        print('----> from delattr')
        # del self.item #无限递归了
        self.__dict__.pop(item)
```

```
# __setattr__添加/修改属性会触发它的执行
f1 = Foo(10)
print(f1.__dict__)
# 因为你重写了__setattr__,凡是赋值操作都会触发它的运行,你啥都没写,就是根本没赋值,除非你直接操作属性字典,否则永远无法赋值

f1.z = 3
print(f1.__dict__)

# __delattr__删除属性的时候会触发
f1.__dict__['a'] = 3
# 我们可以直接修改属性字典,来完成添加/修改属性的操作
del f1.a
print(f1.__dict__)

# __getattr__只有在使用点调用属性且属性不存在的时候才会触发
f1.xxxxxx
```

类的内置成员函数-访问控制，__getattribute__

__getattribute__是属性访问拦截器，就是当这个类的属性被访问时，会自动调用类的__getattribute__方法。即在上面代码中，当我调用实例对象aa的name属性时，不会直接打印，而是把name的值作为实参传进__getattribute__方法中（参数obj是我随便定义的，可任意起名），经过一系列操作后，再把name的值返回。Python中只要定义了继承object的类，就默认存在属性拦截器，只不过是拦截后没有进行任何操作，而是直接返回。所以我们可以自己改写__getattribute__方法来实现相关功能，比如查看权限、打印log日志等。如下代码，简单理解即可：

```
class  MyClass:

        def  __init__(self,  x):
                self.x  =  x

        def  __getattribute__(self,  item):
                print('正在获取属性{}'.format(item))
                return  super(MyClass,  self).__getattribute__(item)
>>>  obj  =  MyClass(2)
>>>  obj.x
正在获取属性x
2
```

```
class Tree(object):
        def __init__(self,name):
                self.name = name
                self.cate = "plant"
        def __getattribute__(self,*args,**kwargs):
                if args[0] == "大树"
                        print("log 大树")
                        return "我爱大树"
                else:
                        return object.__getattribute__(self,*args,**kwargs)
aa = Tree("大树")
print(aa.name)
print(aa.cate)
```

类的内置成员函数-访问控制， __getattribute__

初学者用__getattribute__方法时，容易栽进这个坑，什么坑呢，直接看代码：

```
class Tree(object):
    def __init__(self,name):
        self.name = name
        self.cate = "plant"
    def __getattribute__(self,obj):
        if obj.endswith("e"):
            return object.__getattribute__(self,obj)
        else:
            return self.call_wind()
    def call_wind(self):
        return "树大招风"
aa = Tree("大树")
```

- print(aa.name) #因为name是以e结尾，所以返回的还是name，所以打印出"大树“
- print(aa.wind) #这个代码中因为wind不是以e结尾，#所以返回self.call_wind的结果,打印的是"树大招风"

关于print(aa.name)的解释是正确的，但关于print(aa.wind)的解释不对，为什么呢？

- 执行aa.wind时，先调用__getattribute__方法，经过判断后，它返回的是self.call_wind()，即self.call_wind的执行结果
- 当去调用aa这个对象的call_wind属性时，前提是又要去调用__getattribute__方法，反反复复，没完没了，形成了递归调用且没有退出机制，最终程序就挂了！

类的内置成员函数-访问控制

需求：实现访问控制（注意__getattribute__ 与 __getattr__的区别 ）

```
login_status = False
def login(func):
    def inter(*args, **kwargs):
        global login_status
        if not login_status:
            print("登录...")
            login_status = True
        func(*args, **kwargs)
    return inter
class TestClass:
    def __init__(self, name):
        self.name = name
    @login
    def run(self):
        print("%s is running ..." % self.name)
    @login
    def work(self):
        print("%s is working ..." % self.name)
    @login
    def eat(self):
        print("%s is eating ..." % self.name)
    @login
    def sleep(self):
        print("%s is sleeping ..." % self.name)
c = TestClass("agou")
c.run()
```

```
login_status = False
class TestClass:
    control_list = ['run', 'work', 'sleep', 'eat']
    def __init__(self, name):
        self.name = name
    def __getattribute__(self, item):
        attr = object.__getattribute__(self, item)
        if not attr:
            print("%s 不存在....", item)
            return self.__default
        if item not in TestClass.control_list:
            return attr
        global login_status
        if not login_status:
            print("登录...")
            login_status = True
        return attr
    def run(self):
        print("%s is running ..." % self.name)
    def work(self):
        print("%s is working ..." % self.name)
    def eat(self):
        print("%s is eating ..." % self.name)
    def sleep(self):
        print("%s is sleeping ..." % self.name)
c = TestClass("agou")
c.run()
```

课堂练习（2）：反射

定义一个Peron类

- 初始变量： id、namg、age
- 成员函数：run()、say()
- 给某实例增加新的变量： country、school
- 给某实例增加新的方法：swim()、playgame()
- 给某实例删除变量：school
- 给某实例删除方法：playgame
- 监测某属性(school)是否是成员变量
- 监测某方法(playgame)是否是成员函数

课堂练习（2）： attr相关

定义一个Peron类

- 初始变量： id、namg、age
- 成员函数：run()、say()
- 利用__setattr__，给某实例增加新的变量： country、school
- 利用__setattr__，给某实例增加新的方法：swim()、playgame()
- 利用__delattr__，给某实例删除变量：school
- 利用__delattr__，给某实例删除方法：playgame

课堂练习（2）： 访问控制

定义一个Peron类

- 初始变量： id、namg、age、weight
- 成员函数：run()、say()
- 利用__getattribute__ ，禁止访问变量，age、weight
- 利用__getattribute__ ，控制访问方法，run、say， 在访问上述方法之前，先做登录认证，并且所有对象，登录认证只需要做1次。
- 利用__getattribute__，监测某属性(school)是否是成员变量
- 利用__getattribute__，监测某方法(playgame)是否是成员函数

类的内置成员函数- item操作

模仿dict的操作。

```
class Foo:
    def __init__(self, name):
        self.name = name
    def __getitem__(self, item):
        print(self.__dict__[item])
    def __setitem__(self, key, value):
        self.__dict__[key] = value
    def __delitem__(self, key):
        print('del obj[key]时,我执行')
        self.__dict__.pop(key)
    def __delattr__(self, item):
        print('del obj.key时,我执行')
        self.__dict__.pop(item)
```

```
f1 = Foo('f1')
f1['age'] = 18
f1['age1'] = 19
del f1.age1
del f1['age']
f1['name'] = 'p2'
print(f1.__dict__)
```

类的内置成员函数- str、 doc、 slot、module、class、name

输出字符串 __str__操作

```
class School:
    def __init__(self, name, addr):
        self.name = name
        self.addr = addr

    def __str__(self):
        return '(%s,%s)' % (self.name,
self.addr)

s1 = School('清华大学','北京海淀')

print(s1)
```

输出字注释信息 __doc__

```
class School:
    """
    这是School的接口。
    """
    def __init__(self, name, addr):
        self.name = name
        self.addr = addr
    def __str__(self):
        return '(%s,%s)' % (self.name,
self.addr)

print(School.__doc__)
```

替代__dict__的操作 __slot__

```
class Peson:
    __slots__ = ['name', 'age']
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def show_name(self):
        print(self.name)
    def show_age(self):
        print(self.age)

f1 = Peson('p1', 18)
f1.name = 'f1'
f1.age = 19
print(f1.__slots__)
f2 = Peson('p2', 20)
f2.name = 'f2'
f2.age = 21
print(f2.__slots__)
print("dict:", Peson.__dict__)
# print("dict:", f1.__dict__)
# print("dict:", f2.__dict__)
# f1与f2都没有属性字典__dict__了,统一归
__slots__管,节省内存

f1.show_age()
f2.show_age()
```

__module__、__class__、__name__

```
class C:
    def __init__(self, name):
        self.name = name
    def get_name(self, name):
        return name
c = C('a')
```

```
print(c.__module__)  # 获取所在模块
print(c.__class__)  # 获取类
print(type(c))
print(type(c).__name__)  # 获取类的名称
get_n = getattr(c, 'get_name')
print(get_n)
print(get_n.__name__)  #获取方法的名称
```

类的内置成员函数- __slot__操作

替代__dict__的操作。作用：安全、减少内存。

```
class Foo:
    def __init__(self, name):
        self.name = name
    def __getitem__(self, item):
        print(self.__dict__[item])
    def __setitem__(self, key, value):
        self.__dict__[key] = value
    def __delitem__(self, key):
        print('del obj[key]时,我执行')
        self.__dict__.pop(key)
    def __delattr__(self, item):
        print('del obj.key时,我执行')
        self.__dict__.pop(item)
```

```
f1 = Foo('f1')
f1['age'] = 18
f1['age1'] = 19
del f1.age1
del f1['age']
f1['name'] = 'p2'
print(f1.__dict__)
```

课堂练习（3）： item操作

定义一个Peron类

- 初始变量： id、namg、age、weight
- 成员函数：run()、say()
- 实现类似于字典一样的操作: dic[key] = value

课堂练习（3）：类的内置成员函数

- 自定义1个Person类
  - 自定义print输出（姓名、年龄、体重、身高、国家）
  - 自定义__doc__，并输出
  - 用__slot__，来控制其成员变量
  - 打印其所在的模块名称
  - 自定义一个实例， 并打印其所属类、以及类名
  - 定义一个成员函数，获取该函数的访问接口，并打印函数名称。

类的内置成员函数- 迭代器协议(__next__ and __iter__)

■ 为了让一个对象兼容 for 语句，生成类迭代器协议。

```
class Fib:
    def __init__(self):
        self._a = 0
        self._b = 1

    def __iter__(self):
        return self

    def __next__(self):
        self._a, self._b = self._b, self._a + self._b
        return self._a

f1 = Fib()

print(f1.__next__())
print(next(f1))
print(next(f1))

for i in f1:
    if i > 100:
        break
    print('%s ' % i, end='')
```

```
class Foo:
    def __init__(self, start, stop):
        self.num = start
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        if self.num >= self.stop:
            raise StopIteration
        n = self.num
        self.num += 1
        return n

f = Foo(1, 5)
from collections.abc import Iterable, Iterator

print(isinstance(f, Iterator))

for i in Foo(1, 5):
    print(i)
```

类的内置成员函数- 对with的使用(enter and exit 两个接口)

■ 为了让一个对象兼容with语句，需要使用上下文管理协议。

```
class Open:
    def __init__(self, filepath, mode='r', encoding='utf-8'):
        self.filepath = filepath
        self.mode = mode
        self.encoding = encoding

    def __enter__(self):
        print('出现with语句,对象的__enter__被触发,有返回值则赋值给as声明的变量')
        self.f = open(self.filepath, mode=self.mode, encoding=self.encoding)
        return self.f

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('with中代码块执行完毕时执行我啊')
        print(exc_type)
        print(exc_val)
        print(exc_tb)
        self.f.close()
        return True
with Open('a.txt', 'w') as f:
    print(f)
    f.write('aaaaaa')
    f.wasdf  # 抛出异常，交给__exit__处理
```

用途或者说好处：

■ 使用with语句的目的就是把代码块放入with中执行，with结束后，自动完成清理工作，无须手动干预

■ 在需要访问一些资源比如文件，网络连接和锁等需求时，可以在__exit__中定制自动释放资源的机制，你无须再去关心这个问题，这将大有用处

课堂练习（4）：迭代器协议的使用案例

- 自定义1个MyIntList类， 实现 for … in 功能， 按照 0/2/4/6/8。。。 这样顺序来访问数据
  - ✓ li = MyIntList([1,2,3,4,5,6,7,8,9])
- 自定义1个MyPrime类， 实现 for … in 功能， 获取质数流，一直到给定的最大数（比如：1000）
  - ✓ prime = MyPrime(1000)
- 自定义1个MyFile类， 实现 for … in 功能， 获取文档的数据流，一直到监测到某特定字符串（比如：”aaa”）
  - ✓ File = MyFile(file_name, filter_str_name)

课堂练习（4）：给 MySQL类 增加 with 操作

- 定义1个MySQL类
  - 定义一个 select()函数，返回MySQLResult类
- 定义1个MySQLResult类
  - 属性：rows、cols
  - 方法： get_row(i)，如果 i < 0 或者 i >= rows，则抛出异常
  - 方法： get_col(i)，如果 i < 0 或者 i >= cols，则抛出异常
  - 方法： close()， 退出时调用
- with 里对 MySQLResult 进行操作， 抛出异常