Python程序设计

陈远祥

chenyxmail@gmail.com

北京邮电大学 电子工程学院

Python程序设计

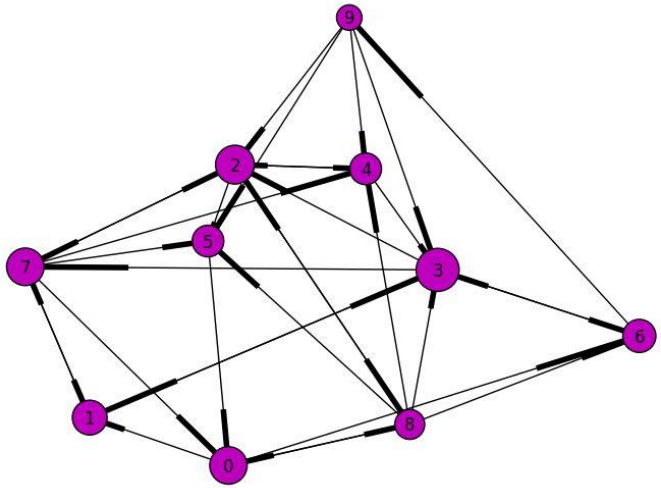
上周主要内容

- ✓ 文件的操作：文本文件，Excel，Word，Pdf

本周主要内容

- ✓ 面向对象思想和编程

面向过程与面向对象



面向过程与面向对象

- 程序包括
  - ✓ 数据:数据类型,数据结构
  - ✓ 处理过程:算法
- 两种程序设计思想
  - ✓ 面向过程:以操作为中心
  - ✓ 面向对象:以数据为中心

面向过程与面向对象

- 面向过程编程：Procedure Oriented Programming(POP)
  - ✓ 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。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度（C语言）

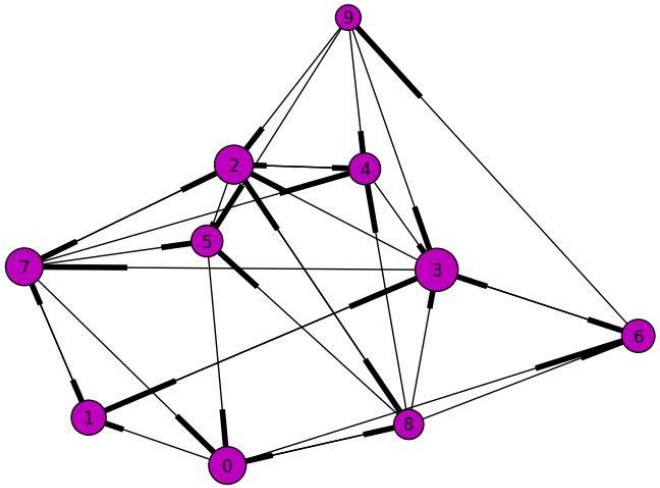
面向过程与面向对象

- 面向对象编程：Object Oriented Programming(OOP)
  - ✓ 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（Python）

面向过程与面向对象

- Python虽然是解释型语言，但从设计之初就已经是一门面向对象的语言，对于Python来说一切皆为对象
- Python中对象的概念很广泛，Python中的一切内容都可以称为对象，而不一定必须是某个类的实例。例如，字符串、列表、字典、元组等内置数据类型都具有和类完全相似的语法和用法

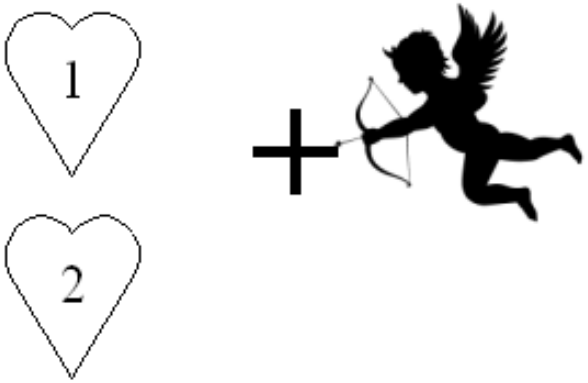
面向过程观点



面向过程观点

- 程序就是对数据进行一系列的操作
  - ✓ 先表示数据:常量,变量
  - ✓ 再来处理数据

```
x = 1
y = 2
z = x + y
print z
```



- 特点:数据与操作分离
  - ✓ 数据是被动的,操作是主动的

面向过程观点

- 准备好数据之后，主要是进行数据处理过程的设计，即算法设计
- 这种设计方法中，数据通常对整个处理过程都是公开的，不能隐藏数据

```
x = 1
y = 2
z = x + y
w = x - y      # 和上一行处理同样的数据x, y
z = z * w
print z
```

面向过程观点

- 复杂处理过程的设计
  - ✓ 模块化

```
def op1(a,b):
    return a * a - b * b
def op2(a,b):
    return a ** b + b ** a
x = 1
y = 2
z = 3
result1 = op1(x,y)
result2 = op2(x,z)
print result1 + result2
```

函数可以看作是更高抽象级的操作,与普通操作似乎并无本质差别.

```
x = 1
y = 2
res = x + y
print res
```

但函数有数据隐藏功能

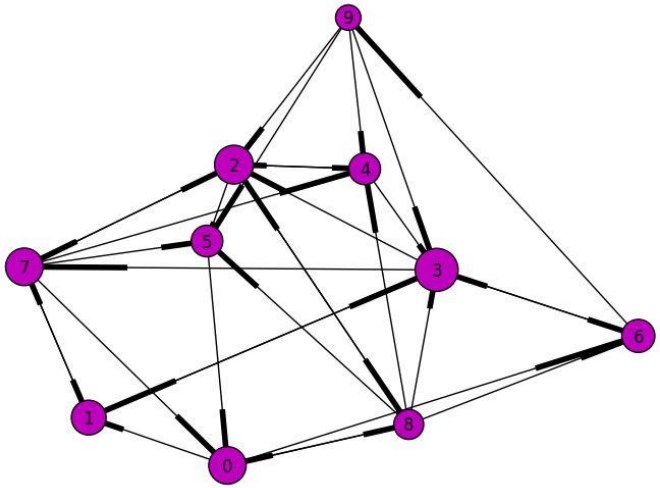
面向过程观点

- 函数是功能黑箱
  - ✓ 使用者需要的只是函数的功能，并不需要知晓它内部是如何实现功能的
  - ✓ 函数内部处理的数据不对函数外部开放，一个函数不能直接访问另一个函数内部的数据

面向过程观点

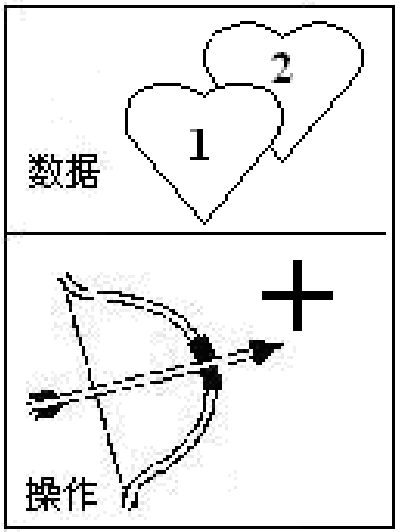
- 数据与操作分离，以操作过程为中心
  - ✓ 先表示数据
  - ✓ 主要精力放在设计数据操作及其流程控制
  - ✓ 对复杂程序采用自顶向下设计和模块化设计
  - ✓ 将使用低级别操作的复杂过程设计成使用高级别操作的简单过程
- 不适合的应用：如GUI程序（图形用户接口）
  - ✓ 没有明确的执行流程，由不可预知的事件驱动处理过程

面向对象观点



面向对象观点

- 数据与操作不可分离
  - ✓ 数据类型概念已经提示我们:特定数据值和特定操作是不可分割的两件事情
  - ✓ x+y是什么意思?
- 既然如此,何不将特定数据值与特定操作捆绑在一起,形成一种新型"数据"?
  - ✓ 由此产生了对象概念



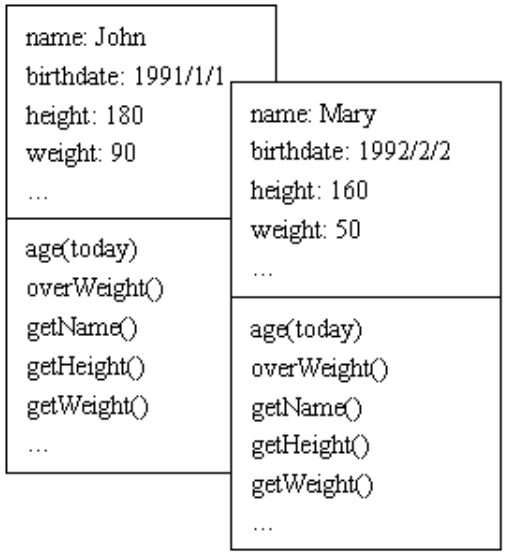
面向对象观点

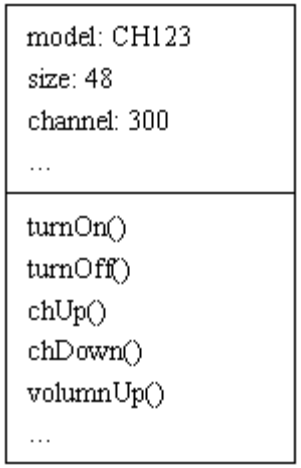
- 对象(Object):集数据与操作于一身
  - ✓ 对象拥有特定数据
  - ✓ 对象能对其数据进行特定操作
- 计算:向对象发操作请求消息
  - ✓ 对象是主动的,自己掌控对其数据的操作
  - ✓ 对象将自己能执行的操作对外公开
- 面向对象(Object-Oriented)编程:软件系统由各种对象组成,对象之间通过消息进行交互

面向对象观点

对象例子

- 人
  - ✓ 数据:姓名, 出生日期, 身高, 体重, . . .
  - ✓ 操作:计算年龄, 判断体重是否标准, . . .
- 电视机
  - ✓ 数据:型号, 厂商, 尺寸, 频道数, . . .
  - ✓ 操作:开机, 关机, 调频道, 调音量, . . .
- 室内环境
  - ✓ 数据:温度, 湿度, 容积, . . .
  - ✓ 操作:调节温度, 调节湿度, 换算容积单位

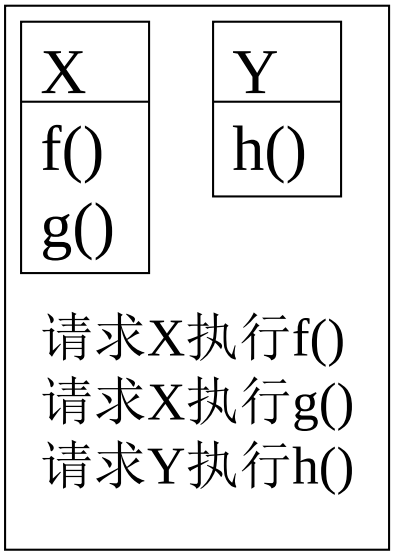




面向对象观点

- 面向过程vs面向对象编程
  - ✓ 假设一个问题涉及数据X和Y，对X要进行的操作为f()，g()，对Y的操作为h()
  - ✓ 面向过程设计和面向对象得到的程序分别形如

```
X = ...
Y = ...
f(X)
g(X)
h(Y)
```

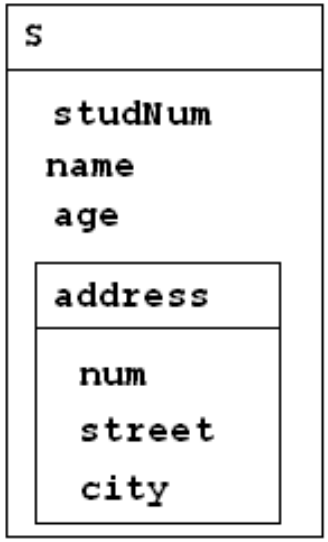


面向对象观点

- 简单数据可以用现成的数据类型表示
- 每个数据类型都包括
  - ✓ 定义一个值的集合:如int
  - ✓ 定义一些对值的运算(操作):如+, -, *, /
- 复杂数据如何表示，两种方法
  - ✓ 拆成简单数据，例如"学生"拆成name, age, addr等简单数据
  - ✓ 定义新类型，例如定义类型S, 其值是由name, age, addr等构成的整体

面向对象观点

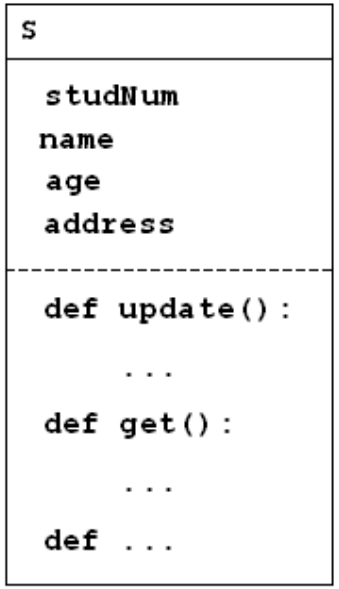
- 分离



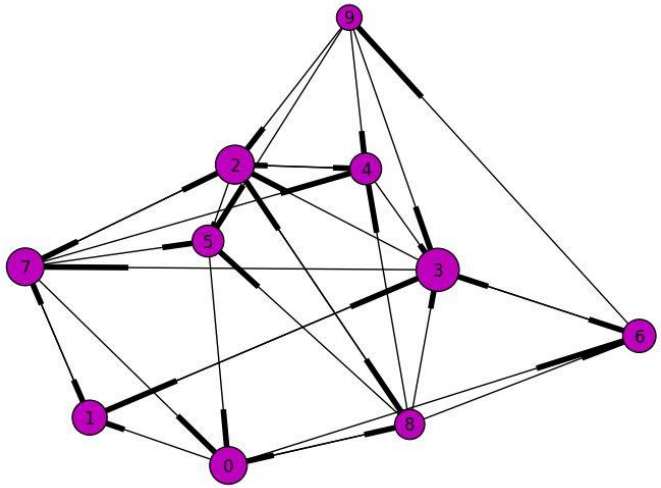
对S型数据的操作

```
def update(s):
    ...
def get(s):
    ...
def ...
```

- 融为一体



类



类

- 类是类型概念的发展
  - ✓ 对象是广义的"数据值"
  - ✓ 对象所属的数据类型就是"类"
  - ✓ 用于描述复杂数据的静态和动态行为
- 类(class):描述相似对象的共性。包括
  - ✓ 数据
  - ✓ 操作:方法(method)
- 对象是类的实例（instance）
  - ✓ 创建对象

类

- 类与抽象
- 类是对现实事物的抽象
  - ✓ 数据抽象，例如:从具体学生抽象出姓名,年龄,地址等数据
  - ✓ 行为抽象，例如:从学生日常行为抽象出选课,加入社团等操作
  - ✓ 于是产生了类Student的定义
- 抽象可以在多个层次上进行
  - ✓ 例如:学生-人-动物-生物

类

- 类的封装：数据和操作结合成一个程序单元，对外部隐藏内部实现细节
  - ✓ 不允许用户直接操作类中被隐藏的信息
  - ✓ 用户也无需了解隐藏的信息就能使用该类
- 类对外公开方法名称和调用格式，即界面
  - ✓ 外界向对象发消息(方法名及参数)
  - ✓ 对象响应消息，执行相应方法
  - ✓ 外界只能按对象允许的方式来处理对象数据

类

- 封装的好处
  - ✓ 安全:对象自己的方法处理自己的数据
  - ✓ 易用:使用者无需了解内部实现细节
  - ✓ 易维护:实现者修改内部实现不会影响使用者
  - ✓ 标准化:同类甚至不同类的对象对使用者都呈现同样的操作界面

类

- 类定义

```
class <类名>:
    <方法定义>
```

类

- 方法定义同函数定义

```
def <方法名>(<self,其他参数>):
    。。。
```

  - ✓ 方法是依附于类的函数，普通函数则是独立的
  - ✓ 方法的第一个参数是专用的，习惯用名字self
  - ✓ 只能通过向对象发消息来调用方法

类

- 对象是数据和操作的结合.
  - ✓ 类定义中,方法对应于操作。数据呢?
- 对象的数据以实例变量形式定义
  - ✓ 实例变量，self.<变量名>
  - ✓ 在方法中定义，self.<变量名> = <变量值>，主要出现在__init__()方法中

类

- 类定义：类Person(可单独保存为模块person.py)
  - ✓ #person

类

- 类属性：
  - ✓ 属性就是属于另一个对象的数据或者函数元素，可以通过我们熟悉的句点属性标识法来访问
  - ✓ 当你正访问一个属性时，它同时也是一个对象，拥有它自己的属性，可以访问，这导致了一个属性链
  - ✓ myThing, subThing, subSubThing

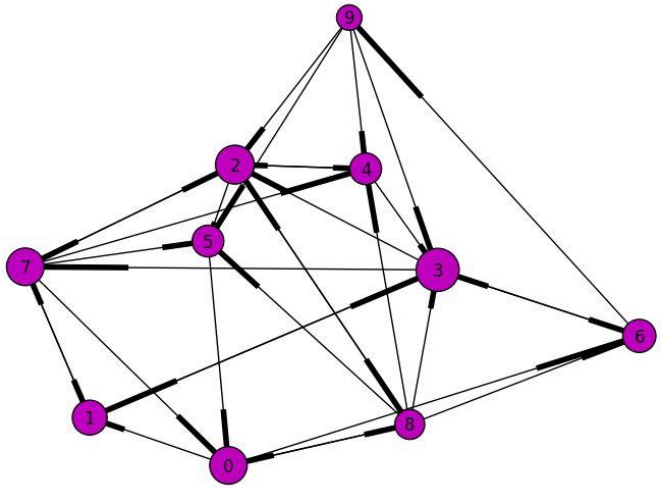
类

- 类的数据属性和方法属性
  - ✓ 数据属性仅仅是所定义的类的变量。它们可以像任何其它变量一样在类创建后被使用，并且，要么是由类中的方法来更新，要么是在主程序其它地方被更新
  - ✓ 定义的函数（方法）

类

- 查看类的属性
  - ✓ dir(Person)
  - ✓ 返回的是对象的属性的一个名字列表

实例



实例

实例创建

- 类与实例:抽象与具体
  - ✓ "人"是类,"张三"是人的实例
  - ✓ 一个类可以创建任意多个实例，各实例具有相同的行为:由方法决定，但具有不同的数据:由实例变量决定

实例

- 实例创建

  <变量> = <类名>(<实参>)

- 这里<类名>相当于一个函数，称为构造器，用来构造实例

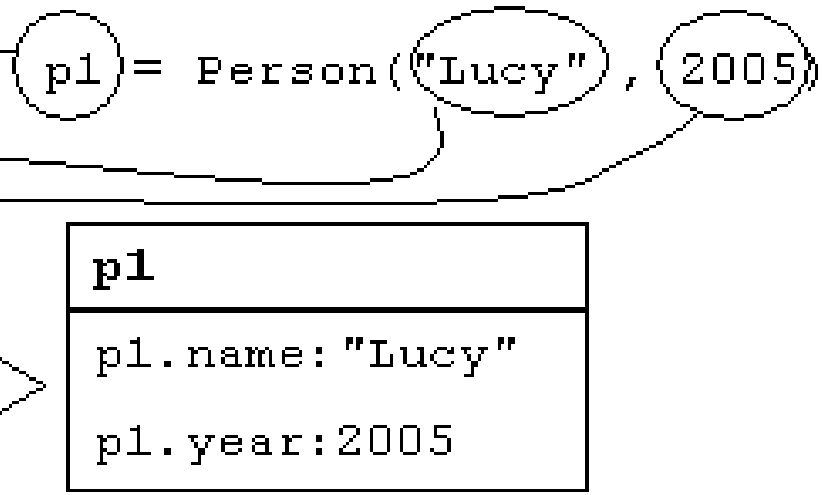
实例

- 创建时对实例进行初始化
  - ✓ 用构造器创建实例时，系统会自动调用__init__方法
  - ✓ 通常在此方法中执行一些初始化操作
  - ✓ __init__所需的参数由构造器提供

实例

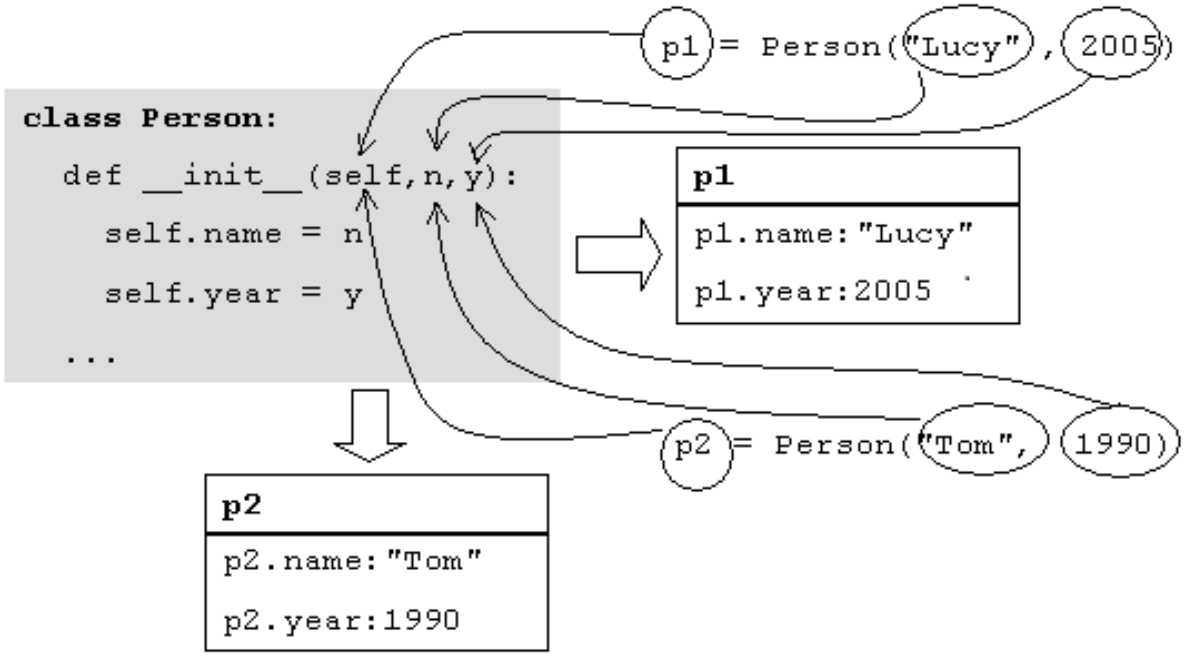
- 创建一个Person实例（实例化）
  - ✓ from person import Person
  - ✓ p1 = Person("Lucy",2005)
  - ✓ #实例化

```
class Person:
  def __init__(self,n,y):
    self.name = n
    self.year = y
  ...
```



面向对象观点

- 创建两个Person实例
  - ✓ from person import Person
  - ✓ p1 = Person("Lucy",2005)
  - ✓ p2 = Person("Tom",1990)



实例

- 每个类实例(对象)具有自己的实例变量副本,用来存储该对象自己的数据
- 对实例变量的访问:
  - ✓ <对象>.<实例变量>
  - ✓ #实例化

实例

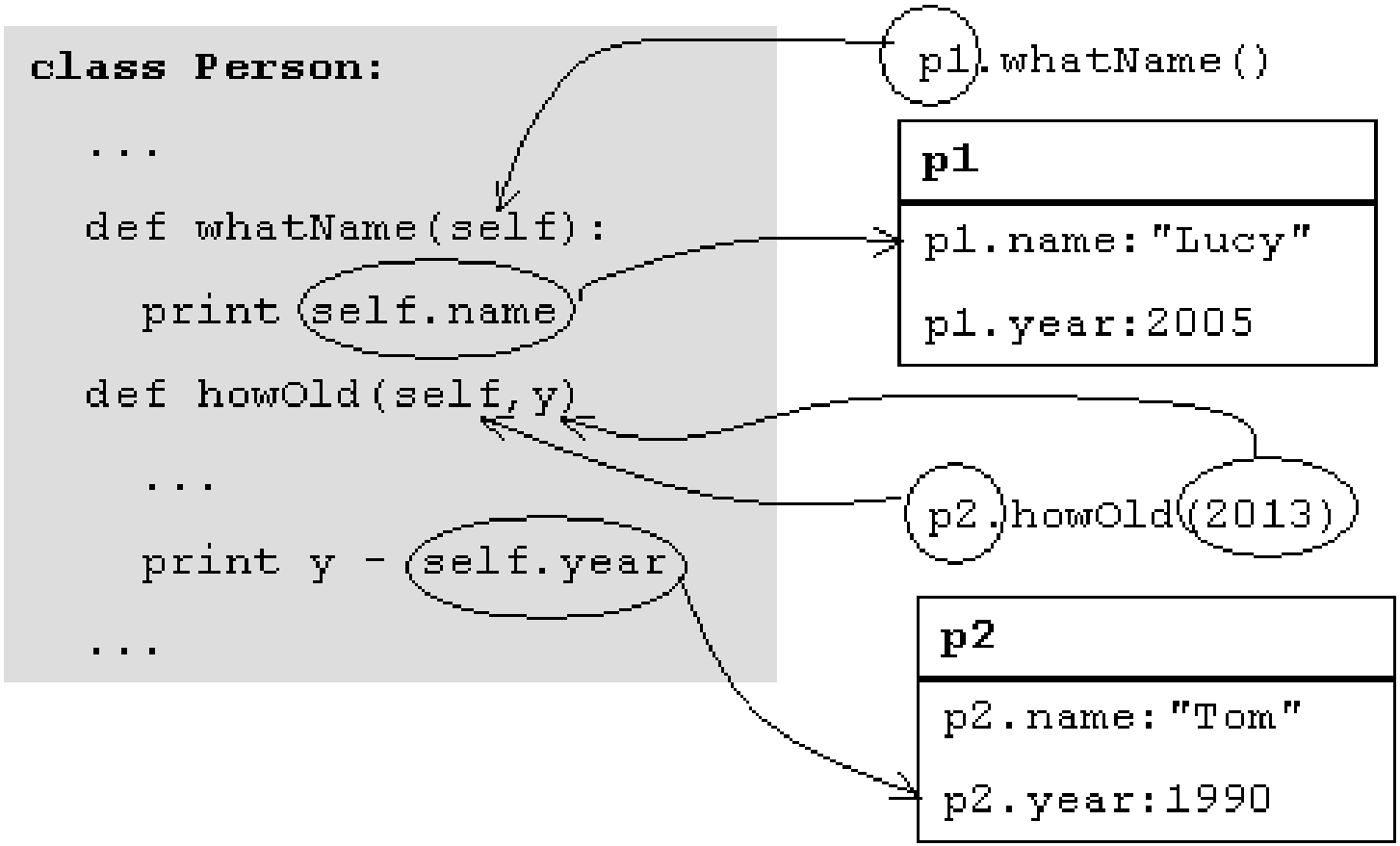
- 实例变量与函数局部变量不同
  - ✓ 同一个类的各个方法都可以访问实例变量
  - ✓ 类的方法中也可以定义局部变量，不能被其他方法访问

实例

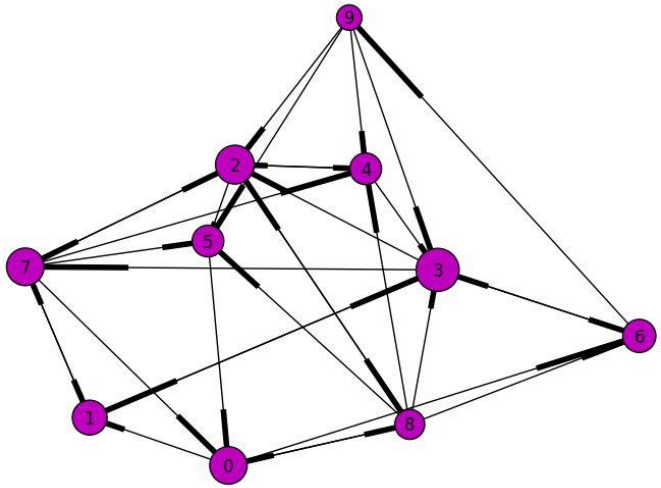
- 方法调用
  - ✓ 类似函数调用，但需指明实例(对象)
  - ✓ <实例>.<方法名>(<实参>)
  - ✓ <实例>就是与形参self对应的实参
  - ✓ #方法调用

实例

- p1.whatName()
- p2.howOld(2013)



炮弹飞行

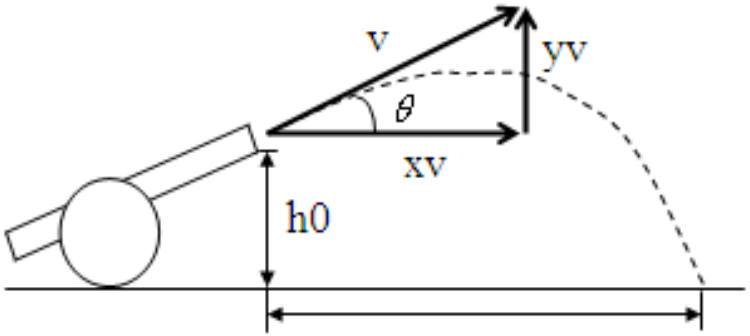


炮弹飞行

- 编程案例:模拟炮弹飞行(1)
- 程序规格
  - ✓ 输入:炮弹的发射角度,初速和高度
  - ✓ 输出:炮弹的射程
- 解决方法:模拟炮弹飞行过程,即计算每一时刻炮弹的位置
  - ✓ 连续运动的离散化
  - ✓ 时间:t, t+Δt, t+2Δt, t+3Δt, ...
  - ✓ 轨迹: (xt, yt), (xt+Δt, yt+Δt), ...

炮弹飞行

- 算法：
  - ✓ 输入角度angle(度)，初速v(米/秒)，高度h0(米)，间隔t(秒)
  - ✓ 将angle换算成弧度单位的theta
  - ✓ xv = v * cos(theta)
  - ✓ yv = v * sin(theta)
  - ✓ 初始位置(xpos, ypos) = (0, h0)
  - ✓ 当炮弹还未落地(即ypos >= 0.0)：
  - ✓ 更新炮弹位置(xpos, ypos)
  - ✓ 更新yv
  - ✓ 输出xpos



炮弹飞行

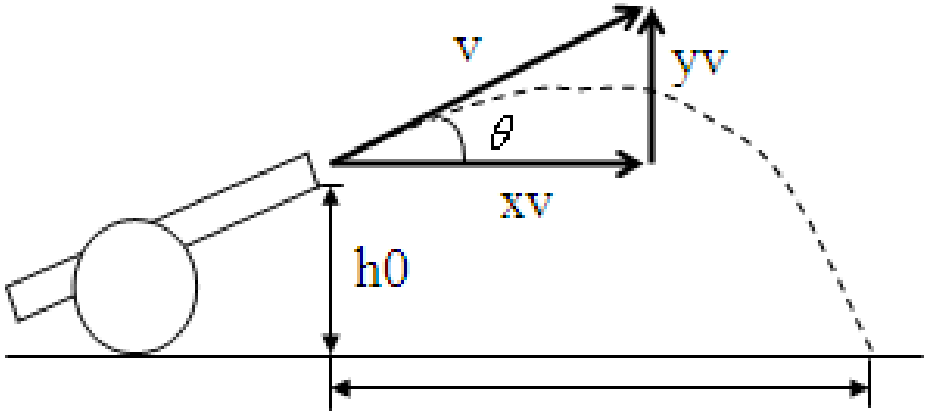
- 核心代码:位置更新
  - ✓ 水平方向

```
xpos = xpos + xv * t
```

  - ✓ 垂直方向

```
yv1 = yv - 9.8 * t
ypos = ypos + (yv + yv1) / 2.0 * t
```

  - ✓ #cball1



炮弹飞行

- 这个版本是"流水帐式的", 没有"章法结构"
- 程序不长, 倒有10个变量, 为理解程序需要跟踪这10个数据的变化
  - ✓ #cball1

炮弹飞行

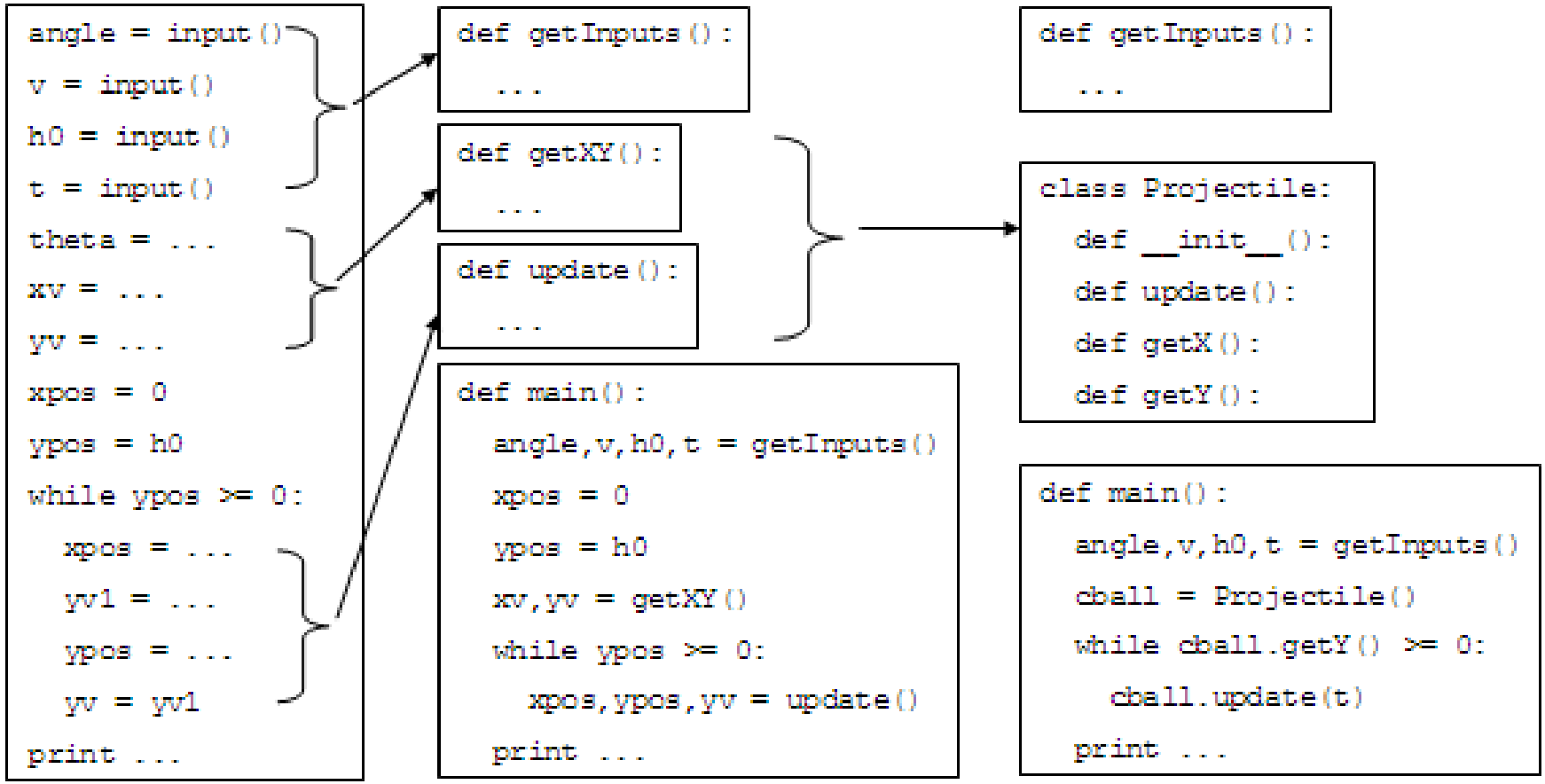
- 模块化版本cball2.py
  - ✓ 主程序(主函数)非常简洁,易理解
  - ✓ 将辅助性变量(theta和yv1)隐藏在辅助函数中
  - ✓ 仍然不够好:update函数界面太复杂
  - ✓ #cball2

炮弹飞行

- 面向对象版本cball3.py
  - ✓ 炮弹是现实实体，用xpos, ypos, xv和yv四个分离的数据来描述它是“只见树木不见森林”
  - ✓ OOP：将炮弹的信息和行为都封装在类中，并创建一个炮弹对象，程序会更加简洁易理解
  - ✓ #cball3

面向对象观点

- 三个版本体现的思想变迁



面向对象观点

- 类与模块化
- 复杂程序的模块化设计，功能模块和数据模块
  - ✓ 功能分解:利用子程序(如函数)概念,以过程为中心设计功能模块
  - ✓ 数据分解:利用类的概念,以数据为中心设计数据模块
- 功能模块不太适合复杂数据的处理,类模块独立性更高,可重用性更好
  - ✓ 类定义可以提供给任何程序使用
  - ✓ 很多OO语言都提供类库

面向对象观点

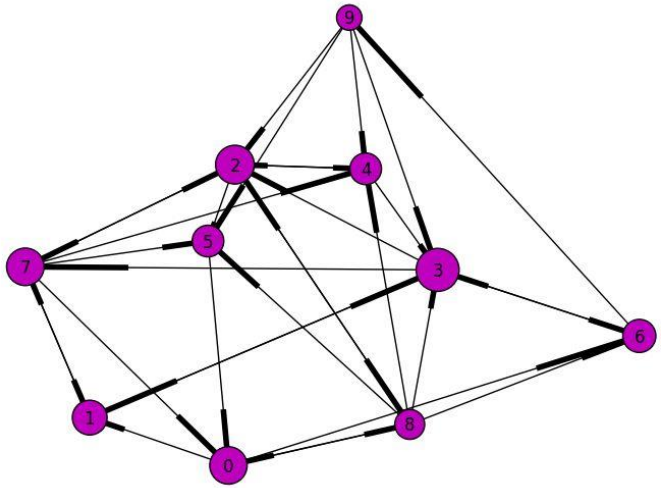
- 例:学生信息处理系统
- 按功能模块分解
  - ✓ 课程注册模块,修改学生信息模块,成绩登录模块等
  - ✓ 每个模块(函数)都需要了解”学生”数据的细节
- 按数据模块分解
  - ✓ 创建”学生”类S,隐藏数据和操作实现细节,使用者无需了解内部细节就能执行操作
  - ✓ 其他数据模块包括”课程”类,”教师”类等

面向对象观点

- 如何表示任意复杂的数据?
- 数据的复杂性表现在
  - ✓ 数量大:用集合体数据类型来表示
  - ✓ 有内部深层结构:用类来表示
- 两种复杂性混合:用"对象的集合"来刻画

```
people = [p1, p2, 。。。]
for p in people:
    p.whatName()
    p.howOld(2013)
      。。。
```

程序入口



程序入口

- Python中if __name__ == '__main__': 的解析
- 程序入口
  - ✓ 对于很多编程语言来说，程序都必须要有一个入口，比如（C，C++，Java，C# ）。比如C和C++都需要有一个main 函数来作为程序的入口，也就是程序的运行会从main函数开始
  - ✓ Java和C#必须要有一个包含Main方法的主类来作为程序入口

程序入口

- Python则不同，它属于脚本语言，不像编译型语言那样先将程序编译成二进制再运行，而是动态的逐行解释运行。也就是从脚本第一行开始运行，没有统一的入口

程序入口

- Python代码的运行，两种方法
  - ✓ 第一是直接作为脚本执行
  - ✓ 第二是import到其他的python脚本中被调用（模块重用）执行
- if __name__ == 'main': 的作用就是控制这两种情况执行代码的过程
- if __name__ == 'main': 下的代码只有在第一种情况下（即文件作为脚本直接执行）才会被执行，而import到其他脚本中是不会被执行的

程序入口

- 例子1
  - ✓ #test
  - ✓ #import_test.py

程序入口

- 运行原理
  - ✓ 每个Python模块（Python文件，也就是此处的test.py和import_test.py）都包含内置的变量__name__，当运行模块被执行的时候，__name__等于文件名（__main__）
  - ✓ 如果import到其他模块中，则__name__等于模块名称。而“__main__”等于当前执行文件的名称
  - ✓ 进而当模块被直接执行时，__name__ == ‘main’结果为真，import导入时候，__name__ == ‘main’为假

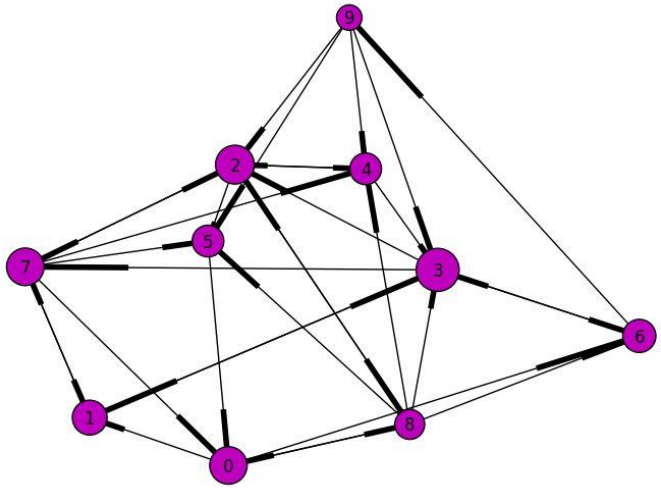
程序入口

- 例子2
  - ✓ #const
  - ✓ #area
- const中的main函数也被运行了，实际上我们是不希望它被运行，提供main也只是为了对常量定义进行下测试
- if __name__ == ‘__main__‘实现main函数的不运行

程序入口

- if __name__ == '__main__' 就相当于是 Python 模拟的程序入口
- Python 本身并没有规定这么写，这只是一种编码习惯
- 由于模块之间相互引用，不同模块可能都有这样的定义，而入口程序只能有一个。到底哪个入口程序被选中，这取决于 __name__ 的值
- 想要运行一些只有在将模块当做程序运行时而非当做模块引用时才执行的命令，只要将它们放到if __name__ == "__main__:"判断语句之后就可以了

实例的访问限制



实例的访问限制

- 实例的访问限制：
  - ✓ 外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。但是，外部代码还是可以自由地修改一个实例的name、score属性
  - ✓ #student

实例的访问限制

- 实例的访问限制：
  - ✓ 如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线__，在Python中，实例的变量名如果以__开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问
  - ✓ #protected_student

实例的访问限制

- 实例的访问限制：
  - ✓ 确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮
  - ✓ 外部代码要获取name和score怎么办？可以给Student类增加get_name和get_score这样的方法
  - ✓ #protected_student

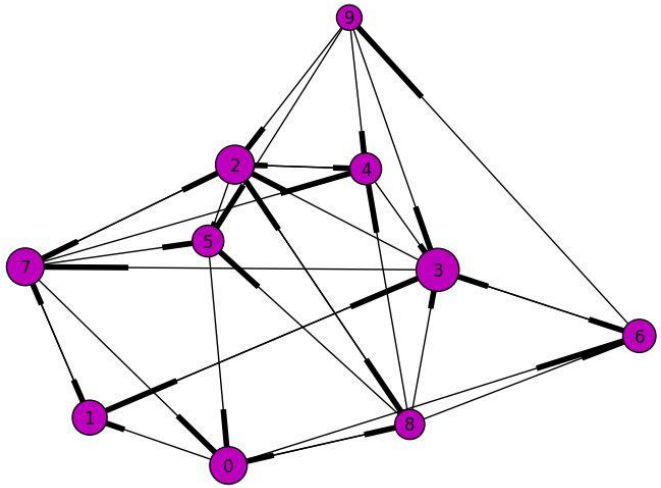
实例的访问限制

- 实例的访问限制：
  - ✓ 如果又要允许外部代码修改score怎么办？可以再给Student类增加set_score方法
  - ✓ #protected_student

实例的访问限制

- 实例的访问限制：
  - ✓ 直接通过bart.score = 99也可以修改啊，为什么要定义一个方法大费周折？
  - ✓ 因为在方法中，可以对参数做检查，避免传入无效的参数

继承



继承

- 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（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）
- 继承是面向对象的重要特征之一，继承是两个类或者多个类之间的父子关系，子类继承了父类的所有公有实例变量和方法

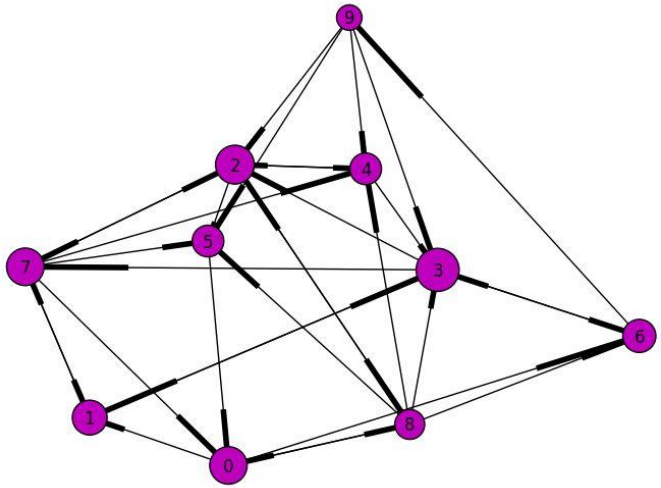
继承

- 继承实现了代码的重用。重用已经存在的数据和行为，减少代码的重新编写
- Python在类名后用一对圆括号表示继承关系，括号中的类表示父类
  - ✓ #继承
  - ✓ 对于Dog来说，Animal就是它的父类，对于Animal来说，Dog就是它的子类
  - ✓ 子类获得了父类的全部功能

继承

- 继承可以对子类增加一些方法和修改
  - ✓ #继承1

多态



多态

- 多态：当子类和父类都存在相同的run()方法时，我们说，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()。这样，我们就获得了继承的另一个好处：多态

多态

- 如何理解多态？
  - ✓ 定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两
  - ✓ a = list() # a是list类型
  - ✓ b = Animal() # b是Animal类型
  - ✓ c = Dog() # c是Dog类型

多态

- 如何理解多态？
  - ✓ 判断一个变量是否是某个类型可以用isinstance()判断
  - ✓ isinstance(a, list)
  - ✓ isinstance(b, Animal)
  - ✓ isinstance(c, Dog)
  - ✓ #多态

多态

- 如何理解多态？
  - ✓ isinstance(c, Dog)
  - ✓ isinstance(c, Animal)
- Dog是从Animal继承下来的，当我们创建了一个Dog的实例c时，我们认为c的数据类型是Dog没错，但c同时也是Animal也没错，Dog本来就是Animal的一种！
  - ✓ #多态

多态

- 如何理解多态？
  - ✓ 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，反之则不行
  - ✓ #多态

多态

- 多态的好处
  - ✓ 要理解多态的好处，我们还需要再编写一个函数（run_twice），这个函数接受一个Animal类型的变量
  - ✓ 我们再定义一个Turtle类型，也从Animal派生
  - ✓ #多态的好处

多态

- 多态的好处
  - ✓ 新增一个Animal的子类，不必对run_twice()做任何修改，实际上，任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态

多态

- 多态的好处
  - ✓ 当我们需要传入Dog、Cat、Turtle……时，我们只需要函数能接收Animal类型就可以了，因为Dog、Cat、Turtle……都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法

多态

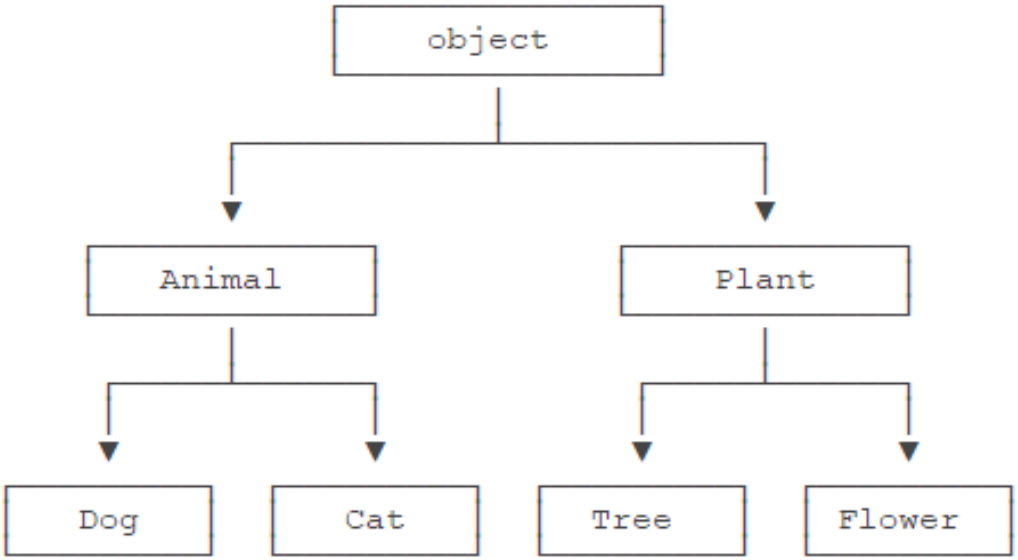
- 多态的好处
  - ✓ 对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Turtle对象上，由运行时该对象的确切类型决定
  - ✓ 这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的

多态

- 多态的“开闭”原则
  - ✓ 对扩展开放：允许新增Animal子类
  - ✓ 对修改封闭：不需要修改依赖Animal类型的run_twice()等函数

多态

- 继承树：继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树



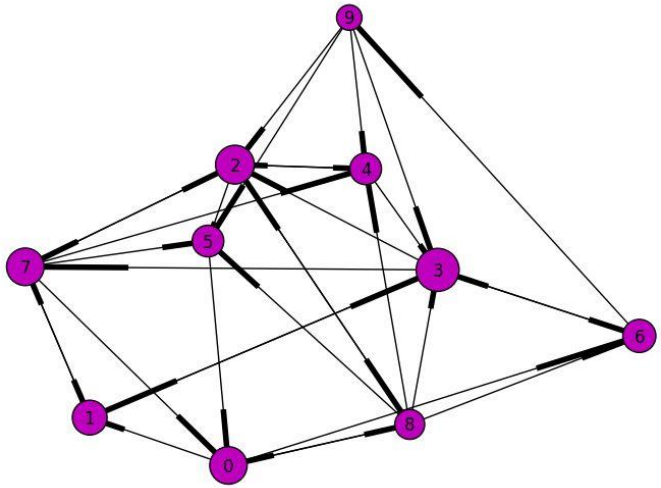
多态

- 对于静态语言（例如Java）来说，如果需要传入Animal类型，则传入的对象必须是Animal类型或者它的子类，否则，将无法调用run()方法
- 对于Python这样的动态语言来说，则不一定需要传入Animal类型。我们只需要保证传入的对象有一个run()方法就可以了

```
class Timer(object):
    def run(self):
        print('Start...')
```

- 这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子

实例属性和类属性



实例属性和类属性

- 实例属性和类属性
  - ✓ 由于Python是动态语言，根据类创建的实例可以任意绑定属性
  - ✓ 给实例绑定属性的方法是通过实例变量，或者通过self变量：

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

s = Student('Bob')
s.score = 90
```

实例属性和类属性

- 实例属性和类属性
  - ✓ 但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

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

  - ✓ 当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到
  - ✓ #类属性

实例属性和类属性

- 实例属性和类属性
  - ✓ 在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性

面向对象高级编程

- 面向对象高级编程
  - ✓ 数据封装、继承和多态只是面向对象程序设计中最基础的3个概念
  - ✓ 在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能：多重继承、定制类、枚举类、元类等概念

谢谢