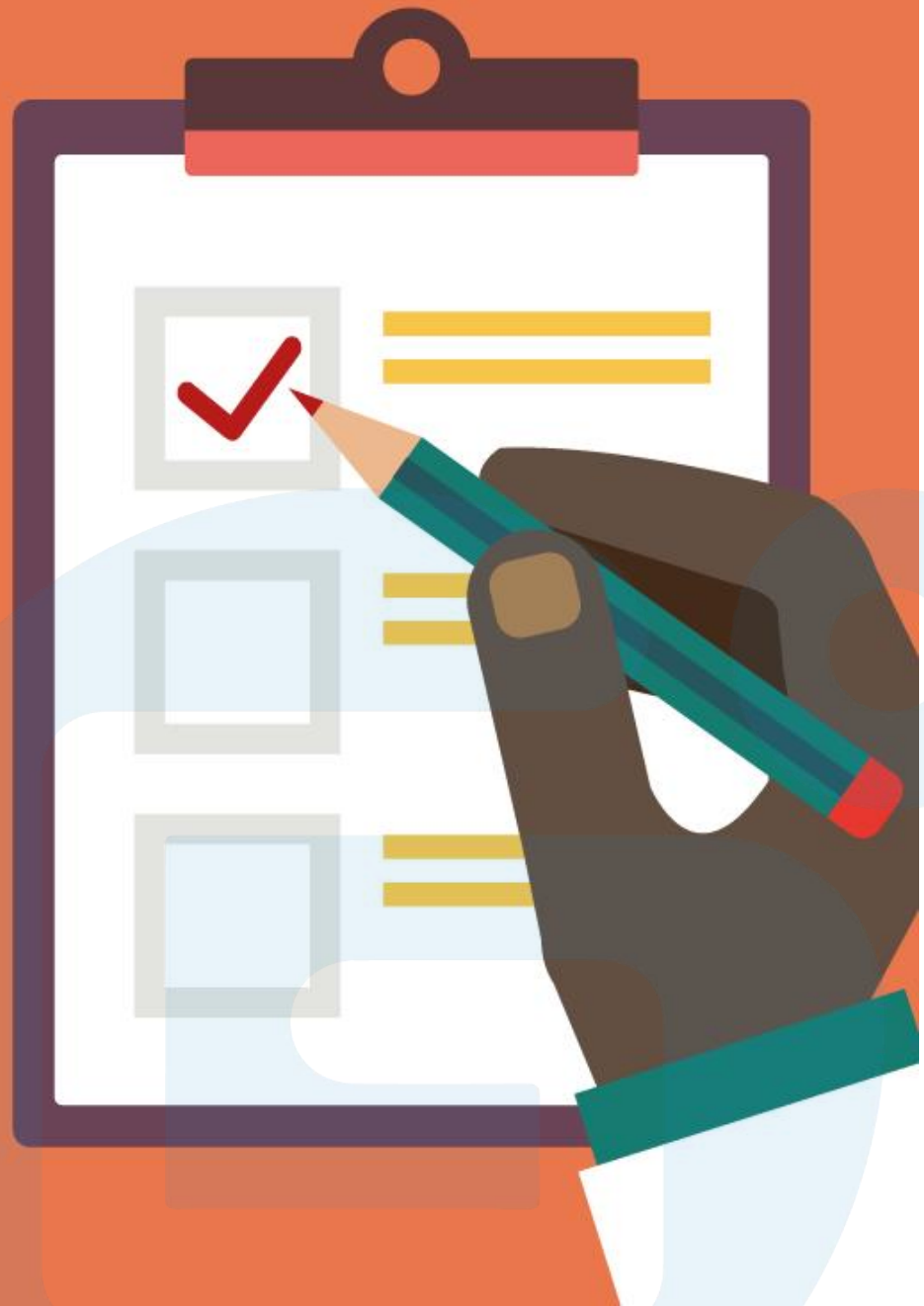
Talk is cheap, show me the code

第三课：面向对象初步

Python进阶课程系列



OUTLINE

- 面向对象的概念
- 类和对象
- 三大特性：继承、多态、封装
- 额外说明

一 面向对象的概念

面向对象的概念

面向过程的程序设计主要是分析实现需求所需要的步骤，通过函数一步一步来实现这些步骤，接着依次调用函数即可。

面向对象的程序设计则是分析出实现需求中设计哪些对象，这些对象各自有哪些特征，有什么功能，对象之间存在哪些联系等，将存在共性的事物或关系抽象成类。最后通过对象的组合和调用完成需求。

- **以五子棋程序为例，面向过程的设计思路就是首先分析问题的步骤：1、开始游戏，2、黑子先走，3、绘制画面，4、判断输赢，5、轮到白子，6、绘制画面，7、判断输赢，8、返回步骤2，9、输出最后结果。把上面每个步骤用不同的方法来实现。**
- **面向对象则是：1、黑白双方，这两方的行为是一模一样的，2、棋盘系统，负责绘制画面，3、规则系统，负责判定诸如犯规、输赢等。第一类对象（玩家对象）负责接受用户输入，并告知第二类对象（棋盘对象）棋子布局的变化，棋盘对象接收到了棋子的变化就要负责在屏幕上面显示出这种变化，同时利用第三类对象（规则系统）来对棋局进行判定。**

面向对象的概念

面向过程的程序设计的核心是过程（流水线式思维），过程即解决问题的步骤，面向过程的设计就好比精心设计好一条流水线，考虑周全什么时候处理什么东西。

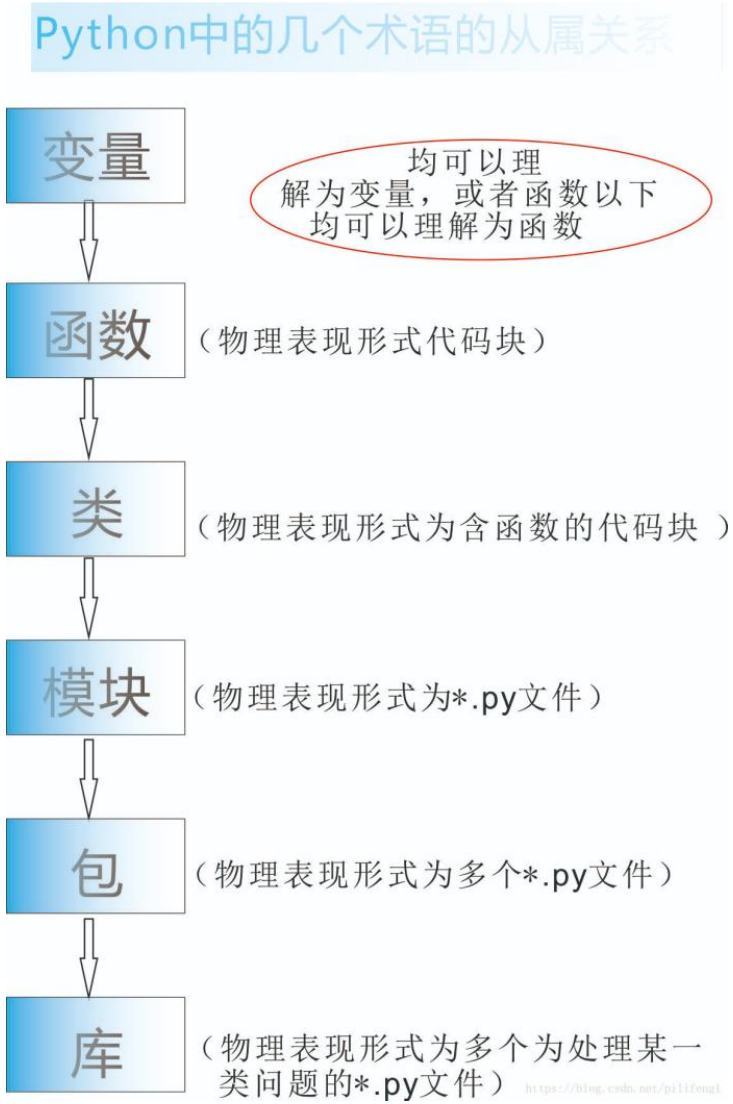
- 优点是：极大的降低了写程序的复杂度，只需要顺着要执行的步骤，堆叠代码即可。
- 缺点是：一套流水线或者流程就是用来解决一个问题，代码牵一发而动全身。
- **应用场景**：一旦完成基本很少改变的场景，著名的例子有Linux内核，git，以及Apache HTTP Server等。

面向对象的程序设计是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

- 优点是：解决了程序的扩展性。对某一个对象单独修改，会立刻反映到整个体系中。
- 缺点是：可控性差，无法向面向过程的程序设计流水线式的可以很精准的预测问题的处理流程与结果，面向对象的程序一旦开始就由对象之间的交互解决问题，即便是上帝也无法预测最终结果。
- **应用场景**：需求经常变化的软件，一般需求的变化都集中在用户层，互联网应用，企业内部软件，游戏等都是面向对象的程序设计大显身手的好地方。

面向对象的概念

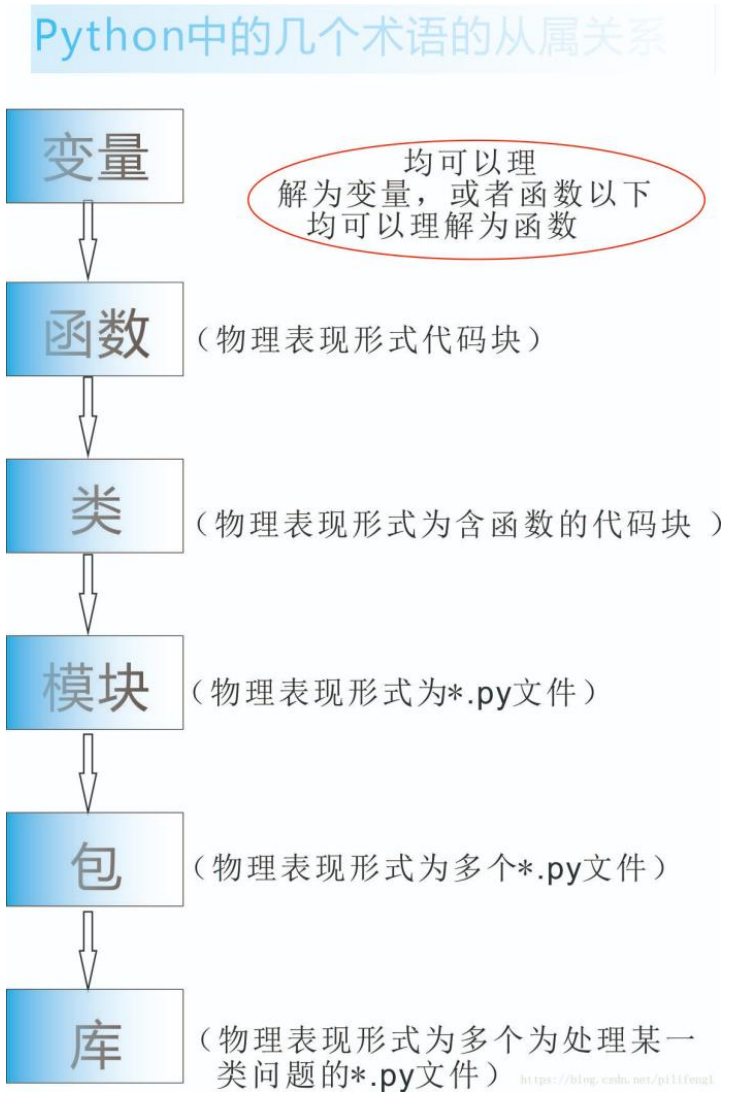
- 面向对象是以功能来划分问题，而不是步骤，这可以使程序的维护和扩展变得更简单，并且可以大大提高程序开发效率，另外，基于面向对象的程序可以使它人更加容易理解你的代码逻辑，从而使团队开发变得更从容。
- 了解一些名词：类、对象、实例、实例化
  - 类（class）：具有相同特征的一类事物(人、狗、鱼、棋子)
  - 对象 / 实例：具体的某一个事物（小明同学、一条金枪鱼）
  - 实例化：类——>对象的过程（这在生活中表现的不明显，我们在后面再慢慢解释）



二 类和对象

类和对象

- 类实际上是一种数据类型，与普通的数据类型不同的是类不仅包含数据，还包含对数据的操作（函数），类把数据和数据操作方法封装在一起，作为一个整体参与程序的运行。类具有可继承性，创建一个新的类的时候，可以在一个基类中添加成员派生出新类。
- 什么时候用类？可复用性是大前提。如果就算个数就结束了，怎么写都随便。那问题就是，什么情况下用function什么情况下用class。如果算法涉及到不断update某些参数或者数据，那么class要好于function。如果一进一出不含记忆的计算，function就可以。



类和对象

- 类也可以理解为一组具有相同属性和行为的对象的模板，是对这组对象的概况、归纳和抽象表达。
- 现实世界里，先有对象后有类。比如我们观察到了鲤鱼、鲫鱼、草鱼、鲢鱼等鱼类之后，把它们统一归类到了淡水鱼的“种类”里。想一想，它们的共同点是什么？
- 在计算机编程的世界里，是先设计了类之后才有对象。在面向对象的程序设计过程中，先在类里面定义共同的属性和行为，然后通过类来创建具有特定属性值和行为的实例，这就是对象。
- 大部分时候，定义一个类就是为了反复创建该类的实例（复用性），类是多个实例的共同特征，类本身不是一种具体的存在，实例才是。

类和对象

Python里使用关键字class来定义类。语法如下：

```
class <类名>：
    类属性1
    ……
    类属性n
    <方法定义1>
    ……
    <方法定义n>
```

例子

定义一个矩形类，包含宽度和高度两个数据成员，role和method两个类属性，提供两个方法（函数），面积和周长。

temp.py

```
1  class Rectangle:
2      "用于计算矩形的面积和周长"
3      role="这是一个矩形"
4      method="计算面积和周长"
5      def __init__(self,width,height): #构造函数，构建初始化方法，传入宽度和高度
6          self.width=width
7          self.height=height
8      def get_area(self):
9          return (self.width*self.height)
10     def get_perimeter(self):
11         return (2*(self.width+self.height))
```

类和对象

temp.py

```
1  class Rectangle:
2      "用于计算矩形的面积和周长"
3      role="这是一个矩形"
4      method="计算面积和周长"
5      def __init__(self,width,height): #构造函数，构建初始化方法，传入宽度和高度
6          self.width=width
7          self.height=height
8      def get_area(self):
9          return (self.width*self.height)
10     def get_perimeter(self):
11         return (2*(self.width+self.height))
12
13 print(Rectangle.role)  #属性查看
14 print(Rectangle.method)
15 print(Rectangle.get_area) #方法引用，注意这里不是调用，因为没有实例
16
17 rect_1=Rectangle(3,4) #实例化，rect_1是一个对象
18 print("矩形的宽度为",rect_1.width)
19 print("矩形的周长",rect_1.get_perimeter())
```

```
In [1]: runfile('C:/Users/Administrator/.spyder-py3/temp.py', wdir='C:/Users/Administrator/.spyder-py3')
这是一个矩形
计算面积和周长
<function Rectangle.get_area at 0x00000201EB08EBF8>
矩形的宽度为 3
矩形的周长 14
```

实例化

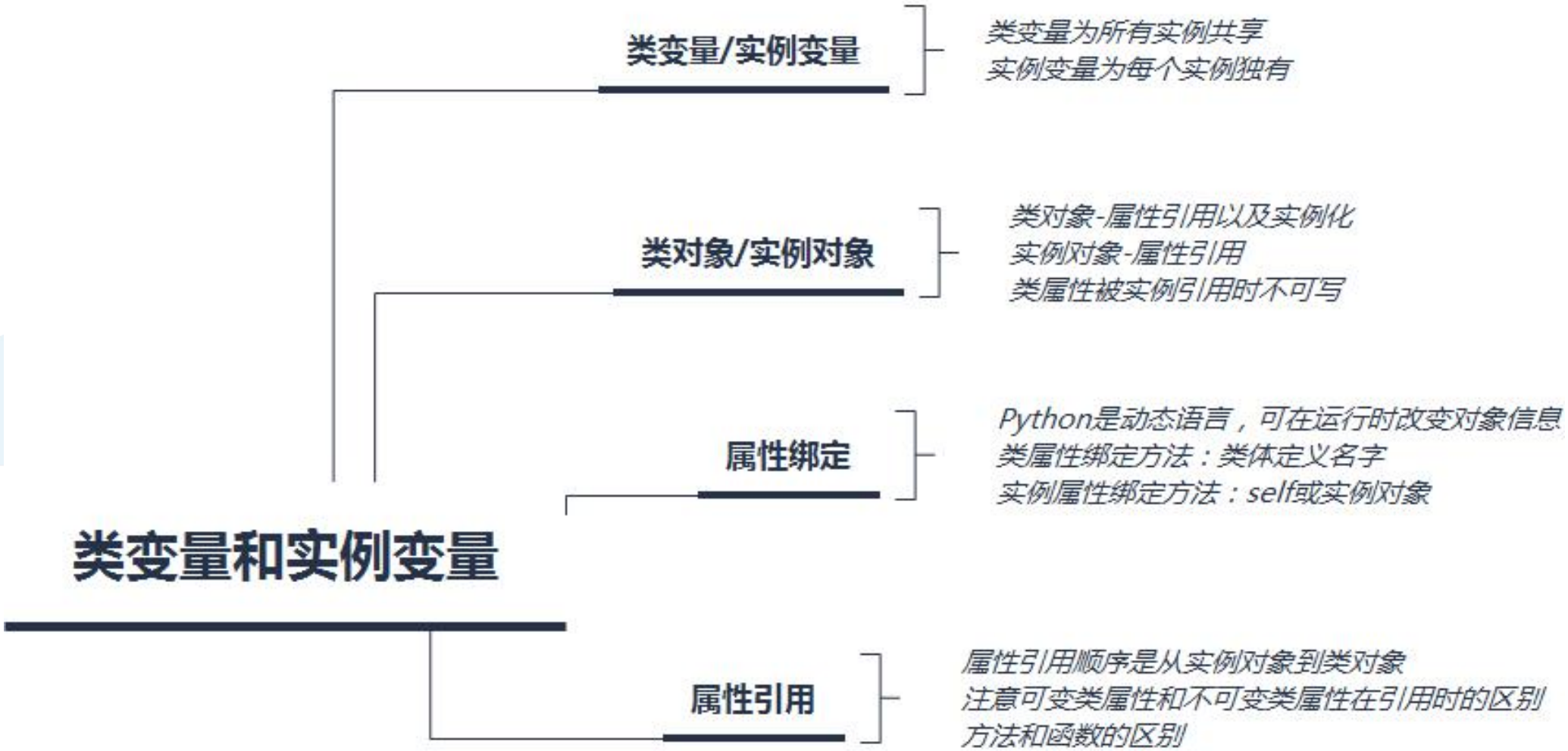
- 要使用类的概念，首先需要实例化，即创建类的对象。创建类的对象类似于函数调用。
- 注意程序是通过类的构造函数__init__接受参数列表里的参数，参数列表里的参数必须与self以外的参数匹配。

对象名 = 类名(参数列表)

调用对象的属性和方法的格式：

- 对象名. 对象的属性
- 对象名.对象的方法()

类和对象


类变量/实例变量
类变量为所有实例共享
实例变量为每个实例独有
类对象/实例对象
类对象-属性引用以及实例化
实例对象-属性引用
类属性被实例引用时不可写
属性绑定
Python是动态语言，可在运行时改变对象信息
类属性绑定方法：类体定义名字
实例属性绑定方法：self或实例对象
类变量和实例变量
属性引用
属性引用顺序是从实例对象到类对象
注意可变类属性和不可变类属性在引用时的区别
方法和函数的区别

类和对象

temp.py*

```
class Person:  # 定义一个人类
    role = 'person'  # 人的角色属性都是人

    def __init__(self, name, aggressivity, life_value):
        self.name = name  # 每一个角色都有自己的昵称;
        self.aggressivity = aggressivity  # 每一个角色都有自己的攻击力;
        self.life_value = life_value  # 每一个角色都有自己的生命值;

    def attack(self,dog):
        # 人可以攻击狗，这里的狗也是一个对象。
        # 人攻击狗，那么狗的生命值就会根据人的攻击力而下降
        dog.life_value -= self.aggressivity

class Dog:  # 定义一个狗类
    role = 'dog'  # 狗的角色属性都是狗

    def __init__(self, name, breed, aggressivity, life_value):
        self.name = name  # 每一只狗都有自己的昵称;
        self.breed = breed  # 每一只狗都有自己的品种;
        self.aggressivity = aggressivity  # 每一只狗都有自己的攻击力;
        self.life_value = life_value  # 每一只狗都有自己的生命值;

    def bite(self,someone):
        # 狗可以咬人，这里的狗也是一个对象。
        # 狗咬人，那么人的生命值就会根据狗的攻击力而下降
        someone.life_value -= self.aggressivity

egg = Person('egon',10,1000)  #创造了一个实实在在的人egg
ha2 = Dog('二哈','哈士奇',10,1000)  #创造了一只实实在在的狗ha2
print(ha2.life_value)          #看看ha2的生命值
egg.attack(ha2)                #egg打了ha2一下
print(ha2.life_value)          #ha2掉了10点血
print(egg.life_value)
ha2.bite(egg)
print(egg.life_value)
```

```
In [9]: runfile('C:/Users/A
1000
990
1000
990
```

- 类里面的变量分为类变量和实例变量两种。类变量为方法以外的变量，所有实例共享，不接受实例传参。实例变量则是每个实例独有的。
- 另外还有局部变量，只能在方法内进行访问。
- 类变量采用“类名.类变量名”的方法进行访问和赋值，如果对不存在的类变量赋值，会给该类添加一个变量。实例变量在方法内部用“self.变量名”访问，在外部直接用“对象名.变量名访问”。

类和对象

temp.py*

```python
class Person:  # 定义一个人类
    role = 'person'  # 人的角色属性都是人

    def __init__(self, name, aggressivity, life_value):
        self.name = name  # 每一个角色都有自己的昵称;
        self.aggressivity = aggressivity  # 每一个角色都有自己的攻击力;
        self.life_value = life_value  # 每一个角色都有自己的生命值;

    def attack(self,dog):
        # 人可以攻击狗，这里的狗也是一个对象。
        # 人攻击狗，那么狗的生命值就会根据人的攻击力而下降
        dog.life_value -= self.aggressivity

class Dog:  # 定义一个狗类
    role = 'dog'  # 狗的角色属性都是狗

    def __init__(self, name, breed, aggressivity, life_value):
        self.name = name  # 每一只狗都有自己的昵称;
        self.breed = breed  # 每一只狗都有自己的品种;
        self.aggressivity = aggressivity  # 每一只狗都有自己的攻击力;
        self.life_value = life_value  # 每一只狗都有自己的生命值;

    def bite(self,someone):
        # 狗可以咬人，这里的狗也是一个对象。
        # 狗咬人，那么人的生命值就会根据狗的攻击力而下降
        someone.life_value -= self.aggressivity

egg = Person('egon',10,1000)  #创造了一个实实在在的人egg
ha2 = Dog('二哈','哈士奇',10,1000)  #创造了一只实实在在的狗ha2
print(ha2.life_value)          #看看ha2的生命值
egg.attack(ha2)                #egg打了ha2一下
print(ha2.life_value)          #ha2掉了10点血
print(egg.life_value)
ha2.bite(egg)
print(egg.life_value)
```

```
In [9]: runfile('C:/Users/A
1000
990
1000
990
```

- 类里面的方法就是与类有关的函数，大致可分为三种。实例方法、类方法和静态方法。
- 实例方法是最常用的一种，类方法和静态方法一般用于类的内部，类方法用于传递类的属性和方法，静态方法是独立的、单纯的函数，仅仅托管于某个类的名称空间中，便于使用和维护。

类和对象

temp.py*

```python
class Person:  # 定义一个人类
    role = 'person'  # 人的角色属性都是人

    def __init__(self, name, aggressivity, life_value):
        self.name = name  # 每一个角色都有自己的昵称;
        self.aggressivity = aggressivity  # 每一个角色都有自己的攻击力;
        self.life_value = life_value  # 每一个角色都有自己的生命值;

    def attack(self,dog):
        # 人可以攻击狗，这里的狗也是一个对象。
        # 人攻击狗，那么狗的生命值就会根据人的攻击力而下降
        dog.life_value -= self.aggressivity

class Dog:  # 定义一个狗类
    role = 'dog'  # 狗的角色属性都是狗

    def __init__(self, name, breed, aggressivity, life_value):
        self.name = name  # 每一只狗都有自己的昵称;
        self.breed = breed  # 每一只狗都有自己的品种;
        self.aggressivity = aggressivity  # 每一只狗都有自己的攻击力;
        self.life_value = life_value  # 每一只狗都有自己的生命值;

    def bite(self,someone):
        # 狗可以咬人，这里的狗也是一个对象。
        # 狗咬人，那么人的生命值就会根据狗的攻击力而下降
        someone.life_value -= self.aggressivity

egg = Person('egon',10,1000)  #创造了一个实实在在的人egg
ha2 = Dog('二哈','哈士奇',10,1000)  #创造了一只实实在在的狗ha2
print(ha2.life_value)          #看看ha2的生命值
egg.attack(ha2)                #egg打了ha2一下
print(ha2.life_value)          #ha2掉了10点血
print(egg.life_value)
ha2.bite(egg)
print(egg.life_value)
```

```
In [9]: runfile('C:/Users/A
1000
990
1000
990
```

- 实例方法和函数的定义类似，但第一个参数必须为实例对象，参数名通常是self，表示当前调用这个方法对象。语法：

  def 实例方法名(self,[形参列表])：

  方法体

- 类的内部通过self.方法名(参数)来调用实例方法，类的外部用对象名.方法名(参数)调用实例方法，调用时无需传递self参数。

类和对象

```python
class Person:  # 定义一个人类
    role = 'person'  # 人的角色属性都是人

    def __init__(self, name, aggressivity, life_value, money):
        self.name = name  # 每一个角色都有自己的昵称;
        self.aggressivity = aggressivity  # 每一个角色都有自己的攻击力;
        self.life_value = life_value  # 每一个角色都有自己的生命值;
        self.money = money

    def attack(self,dog):
        # 人可以攻击狗，这里的狗也是一个对象。
        # 人攻击狗，那么狗的生命值就会根据人的攻击力而下降
        dog.life_value -= self.aggressivity

class Dog:  # 定义一个狗类
    role = 'dog'  # 狗的角色属性都是狗

    def __init__(self, name, breed, aggressivity, life_value):
        self.name = name  # 每一只狗都有自己的昵称;
        self.breed = breed  # 每一只狗都有自己的品种;
        self.aggressivity = aggressivity  # 每一只狗都有自己的攻击力;
        self.life_value = life_value  # 每一只狗都有自己的生命值;

    def bite(self,people):
        # 狗可以咬人，这里的狗也是一个对象。
        # 狗咬人，那么人的生命值就会根据狗的攻击力而下降
        people.life_value -= self.aggressivity

```

```python
class Weapon:
    def __init__(self,name, price, aggrev, life_value):
        self.name = name
        self.price = price
        self.aggrev = aggrev
        self.life_value = life_value

    def update(self, obj):  #obj就是要带这个装备的人
        obj.money -= self.price  # 用这个武器的人花钱买所以对应的钱要减少
        obj.aggressivity += self.aggrev  # 带上这个装备可以让人增加攻击
        obj.life_value += self.life_value  # 带上这个装备可以让人增加生命值

    def prick(self, obj):  # 这是该装备的主动技能,扎死对方
        obj.life_value -= 500  # 假设攻击力是500

lance = Weapon('长矛',200,6,100)
egg = Person('egon',10,1000,600)  #创造了一个实实在在的人egg
ha2 = Dog('二愣子','哈士奇',10,1000)  #创造了一只实实在在的狗ha2

#egg独自力战"二愣子"深感吃力，决定穷毕生积蓄买一把武器
if egg.money > lance.price: #如果egg的钱比装备的价格多，可以买一把长矛
    lance.update(egg) #egg花钱买了一个长矛防身，且自身属性得到了提高
    egg.weapon = lance #egg装备上了长矛

print(egg.money,egg.life_value,egg.aggressivity)
print(ha2.life_value)
egg.attack(ha2)   #egg打了ha2一下
print(ha2.life_value)
egg.weapon.prick(ha2) #发动武器技能
print(ha2.life_value) #ha2不敌狡猾的人类用武器取胜，血槽空了一半
```

类和对象

```python
class Weapon:
    def __init__(self,name, price, aggrev, life_value):
        self.name = name
        self.price = price
        self.aggrev = aggrev
        self.life_value = life_value

    def update(self, obj):  #obj就是要带这个装备的人
        obj.money -= self.price  # 用这个武器的人花钱买所以对应的钱要减少
        obj.aggressivity += self.aggrev  # 带上这个装备可以让人增加攻击
        obj.life_value += self.life_value  # 带上这个装备可以让人增加生命值

    def prick(self, obj):  # 这是该装备的主动技能,扎死对方
        obj.life_value -= 500  # 假设攻击力是500

lance = Weapon('长矛',200,6,100)
egg = Person('egon',10,1000,600)  #创造了一个实实在在的人egg
ha2 = Dog('二愣子','哈士奇',10,1000)  #创造了一只实实在在的狗ha2

#egg独自力战"二愣子"探感吃力，决定穷毕生积蓄买一把武器
if egg.money > lance.price: #如果egg的钱比装备的价格多，可以买一把长矛
    lance.update(egg) #egg花钱买了一个长矛防身，且自身属性得到了提高
    egg.weapon = lance #egg装备上了长矛

print(egg.money,egg.life_value,egg.aggressivity)
print(ha2.life_value)
egg.attack(ha2)   #egg打了ha2一下
print(ha2.life_value)
egg.weapon.prick(ha2) #发动武器技能
print(ha2.life_value) #ha2不敌狡猾的人类用武器取胜，血槽空了一半
```

```
In [6]: runfile('C:/Users/Administrator/.spyder
400 1100 16
1000
984
484
```

类的组合

组合指的是，在一个类中以另外一个类的对象作为数据属性，称为类的组合。这是一种“有”的概念。例如这里person类的实例egg，拥有了一个新的属性“weapon”，weapon的值由“Weapon”类的实例lance确定。

三 三大特性：继承、多态、封装

三大特性：继承、多态、封装

面向对象的三大特点：

- 继承：复用性和扩展性。
- 多态：通过各自不同的实现可以具有不同的数据类型或具有不同的行为，产生不同效果。
- 封装：尽量避免模块之间的干扰，让模块只公开外界所需的内容，隐藏其内部内容。

三大特性：继承、多态、封装

继承的作用

- 减少代码的重用
- 提高代码可读性
- 规范编程模式

几个关键名词

- 抽象：抽象即抽取类似或者说比较像的部分。是一个从具体到抽象的过程
- 继承：子类继承了父类的方法和属性
- 派生：子类在父类方法和属性的基础上产生了新的方法和属性

三大特性：继承、多态、封装

多态的概念

多态性依赖于继承。从一个父类派生出多个子类，可以使子类之间有不同的行为，这种行为称之为多态。实际上子类重写父类的方法，子类如果与父类拥有同一个方法，子类的方法优先级高于父类，子类覆盖父类。

一个接口，多种实现的好处

- 增加了程序的灵活性,以不变应万变，不论对象千变万化，使用者都是同一种形式去调用。
- 增加了程序额可扩展性，通过继承基类创建了一个新的类，使用者无需更改自己的代码，还可以用原方法去调用。

三大特性：继承、多态、封装

封装的概念

- 将不需要对外提供的内容都隐藏起来；
- 把属性都隐藏，提供公共方法对其访问。

封装的好处

- 将变化隔离；
- 便于使用；
- 提高复用性；
- 提高安全性；

封装在于明确区分内外，使得类实现者可以修改封装内的东西而不影响外部调用者的代码；而外部使用用者只知道一个接口(函数)，只要接口（函数）名、参数不变，使用者的代码永远无需改变。这就提供一个良好的合作基础——或者说，只要接口这个基础约定不变，则代码改变不足为虑。

四 额外说明

额外说明

- 很多人在学完了python的class机制之后，遇到一个实际问题还是会迷惑，这太正常了，因为任何程序的开发都是先设计后编程，python的class机制只不过是一种编程方式。过去软件的开发相对简单，从任务的分析到编写程序，再到程序的调试，可以由一个人或一个小组去完成。但是随着软件规模的迅速增大，软件任意面临的问题十分复杂，需要考虑的因素太多，在一个软件中所产生的错误和隐藏的错误、未知的错误可能达到惊人的程度，这也不是在设计阶段就完全解决的。
- 所以软件的开发其实一整套规范，我们所学的只是其中的一小部分，一个完整的开发过程，需要明确每个阶段的任务，在保证一个阶段正确的前提下再进行下一个阶段的工作，称之为软件工程，包括下面几个部分：

  1.面向对象分析（object oriented analysis，OOA）

  2 面向对象设计（object oriented design，OOD）

  3 面向对象编程（object oriented programming，OOP）

  4 面向对象测试（object oriented test，OOT）

  5 面向对象维护（object oriendted soft maintenance，OOSM）
- 在面向对象方法中，最早发展的肯定是面向对象编程(OOP),那时OOA和OOD都还没有发展起来，因此程序设计者为了写出面向对象的程序，还必须深入到分析和设计领域，尤其是设计领域，那时的OOP实际上包含了现在的OOD和OOP两个阶段，这对程序设计者要求比较高，许多人感到很难掌握。
- 现在设计一个大的软件，是严格按照面向对象软件工程的5个阶段进行的，这个5个阶段的工作不是由一个人从头到尾完成的，而是由不同的人分别完成，这样OOP阶段的任务就比较简单了。程序编写者只需要根据OOD提出的思路，用面向对象语言编写出程序既可。在一个大型软件开发过程中，OOP只是很小的一个部分。

额外说明

- 面向对象的程序设计看起来高大上，所以我在编程时就应该保证通篇class，这样写出的程序一定是好的程序？
  - 事实上，面向对象只适合那些可扩展性要求比较高的场景，也就是所谓的一段代码要反复使用。

- 很多人喜欢说面向对象三大特性，那么我在基于面向对象编程时，我一定要让我定义的类中完整的包含这三种特性，这样写肯定是好的程序？
  - 方法不在多，够用就行。写程序不是炫耀技巧，所有特性都是为了方便编程而出现的，结合实际选用。

- 类有类属性，实例有实例属性，所以我们在开始定义class时一定要定义出那么几个类属性，想不到怎么办，那就使劲的想，定义的越多越好？
  - 这就犯了一个严重的教条主义错误，实际上程序越早面向对象，死的越早，为啥面向对象，因为我们要将数据与功能结合到一起，作为一个整体去考虑。假如程序整体的结构都没有出来，或者说需要考虑的问题都没有搞清楚，就开始面向对象了，这就导致了，你在那里干想，自以为想通了，定义了一堆属性，结果后来又都用不到，或者想不通到底应该定义啥，那就一直想吧，想着想着就疯了。所以实际上所有公司在开发一个软件的时候都不是上来就开写，而是频繁的开会讨论计划和构架。

感谢参与 下堂课见