Python进阶学习笔记

第1章　课程介绍

Python 入门中我们学习了：
如何安装 Python 环境
变量和数据类型：Python 内置的基本类型
List 列表和 Tuple 元组：顺序的集合类型
条件判断和循环：控制程序流程
函数：定义和调用函数
切片：如何对 list 进行切片
迭代：如何用 for 循环迭代集合类型
列表生成式：如何快速生成列表
Python 进阶课程我们讲要学习：
函数式编程：不是函数编程哦，是函数式编程
模块：如何使用模块
面向对象编程：面向对象的概念、属性、方法
定制类：利用 Python 的特殊方法定制类

第2章 函数式编程

1. 什么是函数式编程：
   函数：function
   函数式：functional，一种编程范式
   函数式编程特点：把计算视为函数而非指令，贴近计算
   　　纯函数式编程：不需要变量，没有副作用，测试简单
   　　支持告诫函数，代码简洁
   Python 支持的函数式编程特点：
   　　不是纯函数式编程：允许有变量
   　　支持高阶函数：函数也可以作为变量传入
   　　支持闭包：有了闭包就能返回函数
   　　有限度的支持匿名函数
2. 高阶函数
   变量可以指向函数，函数名其实就是指向函数的变量,而高阶函数其实就是可以接收函数做参数的函数。
   Len([1,2,3])=3
   Demo：接收 abs 函数
   　　定义一个函数，接收想，x，y，f三个参数，其中 x，y是普通参数，z是函数。
   def add（x，y，f）
   return f（x）+f（y）

把函数作为参数

传入另一个函数

创建实例属性

虽然可以通过 **Person** 类创建出 **xiaoming**、**xiaohong** 等实例，但是这些实例看上除了地址不同外，没有什么其他不同。在现实世界中，区分 **xiaoming**、**xiaohong** 要依靠他们各自的名字、性别、生日等属性。

如何让每个实例拥有各自不同的属性？由于 Python 是动态语言，对每一个实例，都可以直接给他们的属性赋值，例如，给 **xiaoming** 这个实例加上 **name**、**gender** 和 **birth** 属性：

```
xiaoming = Person()
xiaoming.name = 'Xiao Ming'
xiaoming.gender = 'Male'
xiaoming.birth = '1990-1-1'
```

给 **xiaohong** 加上的属性不一定要和 **xiaoming** 相同：

```
xiaohong = Person()
xiaohong.name = 'Xiao Hong'
xiaohong.school = 'No. 1 High School'
xiaohong.grade = 2
```

实例的属性可以像普通变量一样进行操作：

```
xiaohong.grade = xiaohong.grade + 1
```

任务

请创建包含两个 **Person** 类的实例的 **list**，并给两个实例的 **name** 赋值，然后按照 **name** 进行排序。

sorted() 是高阶函数，接受一个比较函数。

参考代码:

```
class Person(object):
    pass
p1 = Person()
p1.name = 'Bart'

p2 = Person()
p2.name = 'Adam'

p3 = Person()
p3.name = 'Lisa'

L1 = [p1, p2, p3]
L2 = sorted(L1, lambda p1, p2: cmp(p1.name, p2.name))

print L2[0].name
print L2[1].name
print L2[2].name
```

初始化实例属性

虽然我们可以自由地给一个实例绑定各种属性，但是，现实世界中，一种类型的实例应该拥有相同名字的属性。例如，**Person 类**应该在创建的时候就拥有 **name**、**gender** 和 **birth** 属性，怎么办？

在定义 Person 类时，可以为 Person 类添加一个特殊的**__init__()**方法，当创建实例时，**__init__()**方法被自动调用，我们就能在此为每个实例都统一加上以下属性：

```
class Person(object):
    def __init__(self, name, gender, birth):
        self.name = name
        self.gender = gender
        self.birth = birth
```

__init__() 方法的第一个参数必须是 **self**（也可以用别的名字，但建议使用习惯用法），后续参数则可以自由指定，和定义函数没有任何区别。

相应地，创建实例时，就必须要提供除 **self** 以外的参数：

```
xiaoming = Person('Xiao Ming', 'Male', '1991-1-1')
xiaohong = Person('Xiao Hong', 'Female', '1992-2-2')
```

有了**__init__()**方法，每个 Person 实例在创建时，都会有 **name**、**gender** 和 **birth** 这 3 个属性，并且，被赋予不同的属性值，访问属性使用.操作符：

```
print xiaoming.name
# 输 出 'Xiao Ming'
print xiaohong.birth
# 输出 '1992-2-2'
```

要特别注意的是，初学者定义**__init__()**方法常常忘记了 self 参数：

```
>>> class Person(object):
...     def __init__(name, gender, birth):
...         pass
...
>>> xiaoming = Person('Xiao Ming', 'Male', '1990-1-1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3 arguments (4 given)
```

这会导致创建失败或运行不正常，因为第一个参数 name 被 Python 解释器传入了实例的引用，从而导致整个方法的调用参数位置全部没有对上。

任务

请定义 **Person** 类的__init__方法，除了接受 **name**、**gender** 和 **birth** 外，还可接受任意关键字参数，并把他们都作为属性赋值给实例。

要定义关键字参数，使用 ****kw**；

除了可以直接使用 **self.name = 'xxx'**设置一个属性外，还可以通过 **setattr(self, 'name', 'xxx')** 设置属性。

参考代码:

```
class Person(object):
    def __init__(self, name, gender, birth, **kw):
        self.name = name
        self.gender = gender
        self.birth = birth
        for k, v in kw.iteritems():
            setattr(self, k, v)
xiaoming = Person('Xiao Ming', 'Male', '1990-1-1', job='Student')
print xiaoming.name
print xiaoming.job
```

访问限制

我们可以给一个实例绑定很多属性，如果有些属性不希望被外部访问到怎么办？Python 对属性权限的控制是通过属性名来实现的，如果一个属性由双下划线开头(__)，该属性就无法被外部访问。看例子：

```
class Person(object):
    def __init__(self, name):
        self.name = name
        self._title = 'Mr'
        self.__job = 'Student'
p = Person('Bob')
print p.name
# => Bob
print p._title
# => Mr
print p.__job
# => Error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute '__job'
```

可见，只有以双下划线开头的"**__job**"不能直接被外部访问。

但是，如果一个属性以"__xxx__"的形式定义，那它又可以被外部访问了，以"xxx__"定义的属性在 Python 的类中被称为特殊属性，有很多预定义的特殊属性可以使用，通常我们不要把普通属性用"**__xxx__**"定义。

以单下划线开头的属性"_xxx"虽然也可以被外部访问，但是，按照习惯，他们不应该被外部访问。

任务

请给 **Person 类**的__init__方法中添加 **name** 和 **score** 参数，并把 score 绑定到 **__score** 属性上，看看外部是否能访问到。

以双下划线开头的属性无法被外部访问，"**__xxx__**"除外。

参考代码:

```
class Person(object):
```

```
    def init (self, name, score):
        self.name = name
        self._score = score

p = Person('Bob', 59)

print p.name
print p._score
```

输出结果：

```
Bob

Traceback (most recent call last):

  File "./6172/58Wr/index.py", line 9, in

    print p._score

AttributeError: 'Person' object has no attribute '_score'
```

创建类属性

类是模板，而实例则是根据类创建的对象。

绑定在一个实例上的属性不会影响其他实例，但是，类本身也是一个对象，如果在类上绑定一个属性，则所有实例都可以访问类的属性，并且，所有实例访问的类属性都是同一个！也就是说，实例属性每个实例各自拥有，互相独立，而类属性有且只有一份。

定义类属性可以直接在 **class** 中定义：

```
class Person(object):
    address = 'Earth'
    def init (self, name):
        self.name = name
```

因为类属性是直接绑定在类上的，所以，访问类属性不需要创建实例，就可以直接访问：

```
print Person.address
# => Earth
```

对一个实例调用类的属性也是可以访问的，所有实例都可以访问到它所属的类的属性：

```
p1 = Person('Bob')
p2 = Person('Alice')
print p1.address
# => Earth
print p2.address
```

```
# => Earth
```
由于 Python 是动态语言，类属性也是可以动态添加和修改的：
```
Person.address = 'China'
print p1.address
# => 'China'
print p2.address
# => 'China'
```
因为类属性只有一份，所以，当 **Person** 类的 **address** 改变时，所有实例访问到的类属性都改变了。

任务

请给 **Person** 类添加一个类属性 **count**，每创建一个实例，**count** 属性就加 **1**，这样就可以统计出一共创建了多少个 **Person** 的实例。

?不会了怎么办

由于创建实例必定会调用**__init_()**方法，所以在这里修改类属性 **count** 很合适。

参考代码:
```
class Person(object):
    count = 0
    def__init__(self, name):
        Person.count = Person.count +1
        self.name = name
p1 = Person('Bob')
print Person.count
# => 1
p2 = Person('Alice')
print  Person.count
# => 2
p3 = Person('Tim')
print Person.count
# => 3

#输出结果：
1
2
3
```
类属性和实例属性名字冲突怎么办

修改类属性会导致所有实例访问到的类属性全部都受影响，但是，如果在实例变量上修改类属性会发生什么问题呢？
```
class Person(object):
    address = 'Earth'
    def init (self, name):
        self.name = name
```

```
p1 = Person('Bob')
p2 = Person('Alice')

print 'Person.address = ' + Person.address

p1.address = 'China'
print 'p1.address = ' + p1.address

print 'Person.address = ' + Person.address
print 'p2.address = ' + p2.address
```

结 果 如 下 ：

```
Person.address = Earth
p1.address = China
Person.address = Earth
p2.address = Earth
```

我们发现，在设置了 **p1.address = 'China'** 后，p1访问 address 确实变成了 'China'，但是，Person.address 和 p2.address 仍然是'Earch'，怎么回事？

原因是 **p1.address = 'China'并没有改变 Person 的 address**，而是给 **p1 这个实例绑定了实例属性 address**，对p1 来说，它有一个实例属性 address（值是'China'），而它所属的类 Person 也有一个类属性 address，所以:

访问p1.address 时，优先查找实例属性，返回'China'。

访问p2.address 时，p2 没有实例属性 address，但是有类属性 address，因此返回 'Earth'。

可见，当实例属性和类属性重名时，实例属性优先级高，它将屏蔽掉对类属性的访问。

当我们把 p1 的 address 实例属性删除后，访问 p1.address 就又返回类属性的值 'Earth'了：

```
del p1.address
print p1.address
# => Earth
```

可见，千万不要在实例上修改类属性，它实际上并没有修改类属性，而是给实例绑定了一个实例属性。

任务

请把上节的 **Person** 类属性 **count** 改为**__count**，再试试能否从实例和类访问该属性。

把 **count** 改为私有**__count**，这样实例变量在外部无法修改**__count**

参考代码:

```
class Person(object):
    __count = 0
    def __init__(self, name):
        Person.__count = Person.__count + 1
        self.name = name
        print Person.__count
```

```
p1 = Person('Bob')
p2 = Person('Alice')

print Person._count
#输出结果：
不可以
```

定义实例方法

一个实例的私有属性就是以_开头的属性，无法被外部访问，那这些属性定义有什么用？
虽然私有属性无法从外部访问，但是，从类的内部是可以访问的。除了可以定义实例的属性外，还可以定义实例的方法。
实例的方法就是在类中定义的函数，它的第一个参数永远是 self，指向调用该方法的实例本身，其他参数和一个普通函数是完全一样的：

```
class Person(object):

    def_init_(self, name):
        self.__name = name

    def get_name(self):
        return self.__name
```

get_name(self) 就是一个实例方法，它的第一个参数是 self。**__init_(self, name)** 其实也可看做是一个特殊的实例方法。
调用实例方法必须在实例上调用：

```
p1 = Person('Bob')
print p1.get_name()    # self 不需要显式传入
# => Bob
```

在实例方法内部，可以访问所有实例属性，这样，如果外部需要访问私有属性，可以通过方法调用获得，这种数据封装的形式除了能保护内部数据一致性外，还可以简化外部调用的难度。
任务
请给 **Person** 类增加一个私有属性**_score**，表示分数，再增加一个实例方法 **get_grade()**，能根据**_score** 的值分别返回 **A-优秀, B-及格, C-不及格三档。**

?不会了怎么办
注意 **get_grade()**是实例方法，第一个参数为 **self**。
参考代码:

```
class Person(object):

    def_init_(self, name, score):
        self._name = name
```

```
            self._score = score

        def get_grade(self):
            if self._score >= 80:
                return 'A'
            if self._score >= 60:
                return 'B'
            return 'C'

p1 = Person('Bob', 90)
p2 = Person('Alice', 65)
p3 = Person('Tim', 48)

print p1.get_grade()
print p2.get_grade()
print p3.get_grade()
#输出结果：
A
B
C
```

方法也是属性

我们在 **class** 中定义的实例方法其实也是属性，它实际上是一个函数对象：

```
class Person(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def get_grade(self):
        return 'A'

p1 = Person('Bob', 90)
print p1.get_grade
# => <bound method Person.get_grade of <__main__.Person object at 0x109e58510>>
print p1.get_grade()
# => A
```

也就是说，**p1.get_grade** 返回的是一个函数对象，但这个函数是一个绑定到实例的函数，**p1.get_grade()** 才是方法调用。

因为方法也是一个属性，所以， 它也可以动态地添加到实例上，只是需要用 types.MethodType() 把一个函数变为一个方法：

```
import types
def fn_get_grade(self):
    if self.score >= 80:
```

```
        return 'A'
    if self.score >= 60:
        return 'B'
    return 'C'

class Person(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

p1 = Person('Bob', 90)
p1.get_grade = types.MethodType(fn_get_grade, p1, Person)
print p1.get_grade()
# => A
p2 = Person('Alice', 65)
print p2.get_grade()
# ERROR: AttributeError: 'Person' object has no attribute 'get_grade'
```

因为 p2 实例并没有绑定 get_grade

给一个实例动态添加方法并不常见，直接在 class 中定义要更直观。

任务

由于属性可以是普通的值对象，如 **str**，**int** 等，也可以是方法，还可以是函数，大家看看下面代码的运行结果，请想一想 **p1.get_grade** 为什么是函数而不是方法：

```
class Person(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
        self.get_grade = lambda: 'A'

p1 = Person('Bob', 90)
print p1.get_grade
print p1.get_grade()
```

直接把 **lambda** 函数赋值给 **self.get_grade** 和绑定方法有所不同，函数调用不需要传入 **self**，但是方法调用需要传入 **self**。

#输出结果：

```
at 0x12f7e7d0>
  A
```

定义类方法

和属性类似，方法也分实例方法和类方法。

在 **class** 中定义的全部是实例方法，实例方法第一个参数 **self** 是实例本身。

要在 class 中定义类方法，需要这么写：

```
class Person(object):
```

```
    count = 0
    @classmethod
    def how_many(cls):
        return cls.count
    def __init__(self, name):
        self.name = name
        Person.count = Person.count + 1

print Person.how_many()
p1 = Person('Bob')
print Person.how_many()
```

通过标记一个@classmethod，该方法将绑定到 **Person** 类上，而非类的实例。类方法的第一个参数将传入类本身，通常将参数名命名为 **cls**，上面的 **cls.count** 实际上相当于 **Person.count**。

因为是在类上调用，而非实例上调用，因此类方法无法获得任何实例变量，只能获得类的引用。

任务

如果将类属性 **count** 改为私有属性**__count**，则外部无法读取**__score**，但可以通过一个类方法获取，请编写类方法获得**__count** 值。

?不会了怎么办

注意类方法需要添加@classmethod

参考代码:

```
class Person(object):
    __count = 0
    @classmethod
    def how_many(cls):
        return cls.__count
    def __init__(self, name):
        self.name = name
        Person.__count = Person.__count + 1

print Person.how_many()
p1 = Person('Bob')
print Person.how_many()
#输出结果：
0
1
```

第3章　类的继承

5.1 什么是类的继承：

1.如果要编写一个新类：student，需要的属性有：name、gender、school、score，是否需要重新编写呢？
但是我们知道已有的 Person 类：

```
class Person(object):
    def init (self,name,gender):
        self.name = name
        self.gender = gender
```

里面存在已有属性：name，gender
我们可以这样：

```
class Student(Person):
    def init (self,name,gender,school,score):
        super(Student,self). init (name,gender)
        self.school = school
        self.score = score
```

新类可以不用编写，从现有的类继承即可，就自动拥有了父类的所有功能，另外缺少的功能只需要编写即可
2.继承的好处：
复用已有代码，自动拥有现有类的所有功能，只需要编写缺少的功能
3.父类与子类
上一层，也就是被继承的类称为：父类，基类，超类，而继承类被称为：子类，派生类，继承类
4.继承的特点：
（1）.子类和父类是 is 关系：

```
class Student(Person):
    pass
p = Person()
s = Student()
```

“p”是一个 Person 类，而不是一个 Student 类；s 是一个 Student 类，也是一个 Person 类
（2）.错误的继承：
Student 类和 Book 类是 has 关系

```
class Student(Person):
    def __init__(self,bookName):
    self.book = Book(bookName)
```

（3）.Python 的继承特点：
总是从某个类继承，如果没有合适的类，就从 object 类继承：

```
class MyClass(object):
    pass
```

不要忘记调用 super()._init_类，这个方法是用来初始化的：

```
def_init_(self,args):
    super(SubClass,self)._init_(args)
    pass
```

如果没有使用，那么父类的属性很有可能就没有初始化。

5.2 继承一个类

如果已经定义了 **Person** 类，需要定义新的 **Student** 和 **Teacher** 类时，可以直接从 Person 类继承：

```
class Person(object):
    def_init_(self, name, gender):
        self.name = name
        self.gender = gender
```

定义 **Student** 类时，只需要把额外的属性加上，例如 score：

```
class Student(Person):
    def_init_(self, name, gender, score):
        super(Student, self)._init_(name, gender)
        self.score = score
```

一定要用super(Student, self)._init__(name, gender) 去初始化父类，否则，继承自 **Person** 的 **Student** 将没有 **name** 和 **gender**。

函数 **super(Student, self)**将返回当前类继承的父类，即 **Person** ，然后调用**__init__()** 方法，注意 self 参数已在 super()中传入，在__init__()中将隐式传递，不需要写出（也不能写）。

任务

请参考 **Student** 类，编写一个 **Teacher** 类，也继承自 **Person**。

要正确调用 **super()** 的**__init_**方法。

参考代码:

```
class Person(object):
    def_init_(self, name, gender):
        self.name = name
        self.gender = gender
class Teacher(Person):
    def_init_(self, name, gender, course):
        super(Teacher, self)._init_(name, gender)
        self.course = course

t = Teacher('Alice', 'Female','English')
print t.name
print t.course
#输出结果：
Alice
```

English

判断类型

函数 **isinstance()**可以判断一个变量的类型，既可以用在 Python 内置的数据类型如 **str**、**list**、**dict**，也可以用在我们自定义的类，它们本质上都是数据类型。假设有如下的 **Person**、**Student** 和 **Teacher** 的定义及继承关系如下：

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

class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score

class Teacher(Person):
    def __init__(self, name, gender, course):
        super(Teacher, self).__init__(name, gender)
        self.course = course

p = Person('Tim', 'Male')
s = Student('Bob', 'Male', 88)
t = Teacher('Alice', 'Female', 'English')
```

当我们拿到变量 **p**、**s**、**t** 时，可以使用 **isinstance** 判断类型：

```
>>> isinstance(p, Person)
True      # p 是 Person 类型
>>> isinstance(p, Student)
False     # p 不是 Student 类型
>>> isinstance(p, Teacher)
False     # p 不是 Teacher 类型
```

这说明在继承链上，一个父类的实例不能是子类类型，因为子类比父类多了一些属性和方法。

我们再考察s ：

```
>>> isinstance(s, Person)
True      # s 是 Person 类型
>>> isinstance(s, Student)
True      # s 是 Student 类型
>>> isinstance(s, Teacher)
False     # s 不是 Teacher 类型
```

s 是 Student 类型，不是 Teacher 类型，这很容易理解。但是，**s**也是 Person 类型，因为 Student 继承自 Person，虽然它比 Person 多了一些属性和方法，但是，把 **s** 看

成 Person 的实例也是可以的。
这说明在一条继承链上，一个实例可以看成它本身的类型，也可以看成它父类的类型。
任务
请根据继承链的类型转换，依次思考 **t** 是否是 **Person**，**Student**，**Teacher**，**object** 类型，并使用 **isinstance()**判断来验证您的答案。

注意 **t** 是 Teacher 的实例，继承链是：
object <- Person <- Teacher
参考代码:

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

class Student(Person):
    def __init__(self, name, gender, score):
        super(Student, self).__init__(name, gender)
        self.score = score

class Teacher(Person):
    def __init__(self, name, gender, course):
        super(Teacher, self).__init__(name, gender)
        self.course = course

t = Teacher('Alice', 'Female', 'English')

print isinstance(t, Person)
print isinstance(t, Student)
print isinstance(t, Teacher)
print isinstance(t, object)
#输出结果：
True
True
False
True
```

多态

类具有继承关系，并且子类类型可以向上转型看做父类类型，如果我们从 **Person** 派生出 **Student** 和 **Teacher**，并都写了一个 **whoAmI()** 方法：

```
class Person(object):
    def __init__(self, name, gender):
```

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：
https://d.book118.com/718034127125007004