Python Package & OOP

Python module and package

- 模块（module）：用来从逻辑（实现一个功能）上组织Python代码（变量、函数、类），本质就是*.py文件。文件是物理上组织方式"module_name.py"。 模块就是一个保存了Python代码的文件。
- 模块能定义函数，类和变量, 模块里也能包含可执行的代码
- 其他可作为module的文件类型还有".pyo"、".pyc"、".pyd"、".so"、".dll",

module来源有3种：

① Python内置的模块（标准库）；

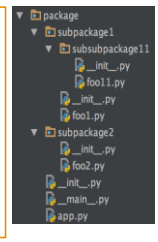
② 第三方模块；

③ 自定义模块。

Python module and package

1. 包 package： 为避免模块名冲突，Python引入了按目录组织模块的方法，称之为 包（package）。包是含有模块的文件夹。

2. 当一个文件夹下有 init .py时，意为该文件夹是一个包（package），其下的多个模块（module）构成一个整体，而这些模块（module）都可通过同一个包（package）导入其他代码中。



Python module and package

命名空间

- 每个函数function 有自己的命名空间，称local namespace，记录函数的变量。
- 每个模块module 有自己的命名空间，称global namespace，记录模块的变量，包括functions、classes、导入的modules、module级别的变量和常量。
- build-in命名空间，它包含build-in function和exceptions，可被任意模块访问。

代码访问变量x 时，Python会在所有的命名空间中查找该变量，顺序是：

- local namespace 即当前函数或类方法。若找到，则停止搜索；
- global namespace 即当前模块。若找到，则停止搜索；
- build-in namespace Python会假设变量x是build-in的函数函数或变量。若变量x不是build-in的内置函数或变量，Python将报错NameError。

Python module and package

命名空间

命名空间在from module_name import 、import module_name中的体现：from关键词是导入模块或包中的某个部分。

- from module_A import X：会将该模块的函数/变量导入到当前模块的命名空间中，无须用module_A.X访问了。
- import module_A：modules_A本身被导入，但保存它原有的命名空间，得用module_A.X方式访问其函数或变量。

Import module and package

导入模块

绝对导入：所有的模块import都从"根节点"开始。根节点的位置由sys.path中的路径决定， 需手动修改sys.path。

相对导入：只关心相对自己当前目录的模块位置就好。不管根节点在哪儿，包内的模块相对位置都是正确的。

from . 和 from .. import 等通过点表示层级的显示相对导入。.表示当前模块，.. 表示上一级模块，... 表示上上一级模

有点复杂

Import module and package

- 若想使用from pacakge import * 这种形式的写法，需在 init .py中加上：__all__= ['file_a', 'file_b'] 在导入时 init .py文件将被执行。
- 形如from package import *，*是由__all__定义的。

单独导入包（package）：单独import某个包名称时，不会导入该包中所包含的所有子模块。除非在__init__.py中加入 from . import xxx

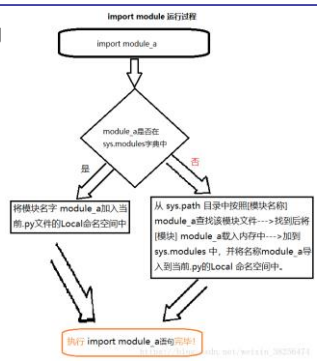
Python module and package

Python运行机制 import语句进行了啥操作?

- step1：创建一个新的、空的module对象（它可能包含多个module）；
- step2：将该module对象 插入sys.modules中；
- step3：装载module的代码（如果需要，需先编译）；
- step4：执行新的module中对应的代码。

Python module and package

Python运行机制



Python OOP

面向对象编程是模拟人类认识事物的方式的编程方法，是最有效的编程方法之一。人类通过将事物进行分类来认识世界，

比如，人类将自然界中的事物分类生物和非生物，又将生物分为动物、植物、微生物，又将动物分为有脊椎动物和无脊椎动物，继而又分为哺乳类、鸟类、鱼类、爬行类等，哺乳类又分为猫、狗、牛、羊等。

每一个类的个体都具有一些共同的属性，在面向对象编程中，个体被称为对象，又称为实例。

Python OOP

- 面向对象程序设计（Object Oriented Programming，OOP）的思想主要针对大型软件设计而提出，使得软件设计更加灵活，支持代码复用和设计复用，代码具有更好的可读性和可扩展性。
- OOP的一条基本原则是计算机程序由多个能够起到子程序作用的单元或对象组合而成，这大大地降低了软件开发的难度，使得编程就像搭积木一样简单。
- OOP的一个关键性观念是将数据以及对数据的操作封装在一起，组成一个相互依存、不可分割的整体，即对象。对于相同类型的对象进行分类、抽象后，得出共同的特征而形成了类，面向对象程序设计的关键就是如何合理地定义和组织这些类以及类之间的关系。

Python OOP

几个术语

对象

- 描述事物的实体，是构成程序的基本单位。
- 对象由一组属性（数据）和一组行为（函数或称为方法）构成。属性用来描述事物的静态特征，行为用来描述对象的动态特征。

类

- 类是具有相同属性和行为的一组对象的集合，为全部对象提供抽象的描述，包括属性和行为。
- 类和对象的关系是抽象与具体的关系，一个属于某一类的对象称为该类的一个实例（instance）。

Python OOP

几个术语

封装: 封装是面向对象程序设计方法的一个特点和重要原则，它将对象的属性和行为集合成一个独立的单元，并尽可能隐藏对象的内部细节。

封装有两个特点:

1. 将对象的全部属性和行为组合在一起，形成一个不可分割的独立单元;
2. 对这个独立单元进行信息的隐藏，使得外界无法轻易获得单元中的信息，实现信息保护，外界只有通过单元提供的某些特定接口（函数或方法）与其发生联系。

Python OOP

几个术语

继承: 一个类拥有另一个类的全部或部分属性和行为，则可以将这个类声明为继承自另一个类。继承能够提高程序的可重用性和开发效率。

多态: 几个相似而不完全相同的对象，我们在向它们发出同一消息时，它们的反应不同，分别执行不同的操作，这种情况称为多态。让具有不同功能的函数可以使用相同的函数名，这样就可以用一个函数名调用不同内容(功能)的函数。

Python OOP

- Python完全采用了面向对象程序设计的思想，是真正面向对象的高级动态编程语言，完全支持面向对象的基本功能，如封装、继承、多态以及对基类方法的覆盖或重写。
- 但与其他面向对象程序设计语言不同的是，Python中对象的概念很广泛，Python中的一切内容都可以称为对象例如，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。

Python OOP

面向过程

面向对象

Python OOP

面向过程:

- 优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源; 比如单片机、嵌入式开发、 Linux/Unix 等一般采用面向过程开发，性能是最重要的因素。
- 缺点：没有面向对象易维护、易复用、易扩展

面向对象:

- 优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统 更加灵活、更加易于维护
- 缺点：性能比面向过程低

Python OOP

类定义

```
class Calculator:#对于类的定义我们要求首字母大写
    '''这是一个计算器Calculator类'''
    name ='Good calculator' #固有属性项
    price =20
    def __init__(self,name,price):#初始化类的属性
        self.name=name
        self.p=price
    #定义内部函数,实现功能
    def add(self,x,y):#self 表示本类
        print(self.name)#在类中使用self调用它的名字
        result = x+y
        print(result)
    def minus(self,x,y):
        result = x-y
        print(result)
```

- 类的说明文档，放在类声明之后、类体之前
- 根据定义属性位置的不同，在各个类方法之外定义的称为类属性或类变量（如name和price属性），在类方法中定义的属性称为实例属性（或实例变量）。
- add()和minus()是实例方法，Python类中还可以定义类方法和静态方法。

Python OOP

构造方法__init()__

- 在创建类时，可以手动添加一个__init__()方法，称为构造函数。
- 与普通函数唯一的差别是调用方式的不同，根据类创建对象时，自动调用。
- 开头和末尾各有两个下划线，避免与Python的默认方法或普通方法发生名称冲突。
- __init__()方法可以包含多个参数，但必须包含一个名为 self 的参数，且必须作为第一个参数。
- 创建类实例对象时，会动传入实参self。
- 如果没有为类定义任何构造方法，那么Python会自动为该类创建一个只包含self参数的默认的构造方法。

Python OOP

self 参数

类的所有实例方法都必须至少有一个名为“self”的参数，并且必须是方法的第一个形参（如果有多个形参的话），“self”参数代表将来要创建的对象本身。
在类的实例方法中访问实例属性时需要以“self”为前缀，

Python OOP

创建对象 对象名 = 类名(参数)

定义了类之后，可以用来实例化对象，并通过“对象名.成员”的方式来访问其中的数据成员或成员方法，例如下面的代码：

```
>>> car = Car()
>>> car.infor()
```

使用内置方法 isinstance()来测试一个对象是否为某个类的实:

```
>>> isinstance(car, Car)    # True
>>> isinstance(car, str)    # False
```


Python OOP

类变量与实例变量

根据定义属性位置的不同，属性可分为类属性（类变量）和实例属性（实例变量）。

- 类变量指的是定义在类中，但在各个类方法外的变量。
- 实例变量指的是定义在类的方法中的变量。

Python OOP

类变量

- 所有实例化对象都可以共享类变量的值，作为公用资源。
- 类变量的访问：推荐用类名.类变量名，也可以使用对象名.类变量或self.变量名，但是不能将类名和对象名都省略，直接通过变量名进行访问。

```
class Calculator:
    name ='Good calculator'
    price =20
    def info(self):
        print(Calculator.name)
        print(Calculator.price)
calc = Calculator()
calc.info()
```

```
class Calculator:
    name ='Good calculator'
    price =20
    def info(self):
        print(self.name)
calc = Calculator()
calc.info()
print(calc.price)
```


Python OOP

类变量

- 可以通过为类名.类变量赋值的方式来修改类变量的值。
- 也可以动态的为类增加类属性（类变量）。
- 改变类变量的值会作用于该类所有的实例化对象。

```
calc = Calculator()
calc.info()
Calculator.name ='Bad calculator'
Calculator.price =10
calc1 = Calculator()
calc1.info()
calc.info()
```

```
class Calculator:
    name ='Good calculator'
    price =20
    def info(self):
        print(Calculator.name)
        print(Calculator.price)
calc = Calculator()
calc.info()
Calculator.msg ='这是一个计算器'
print(Calculator.msg)
print(calc.msg)
```

Python OOP

实例变量　实例变量只能通过对象名访问，无法通过类名直接访问。

```
class Calculator:
    name='Good calculator'
    price=20
    def change(self,name,price):
        self.name=name
        self.price=price
calc = Calculator()
calc.change('Bad calculator',10)
print(calc.name)
print(calc.price)
print(Calculator.name)
print(Calculator.price)
```

- calc.change('Bad calculator',10)更改了对象calc中的name和price两个实例变量的值，但并不会影响Calculator类变量的值。
- self.name=name和self.price=price，它们的作用是：重新定义了两个实例变量(与类变量同名)。这时，对象将无法调用类变量，因为它会首选实例变量。

Python OOP

实例变量

- 通过类修改类变量的值，对象的实例变量的值不会受到任何影响。
- 程序对一个对象的实例变量进行了修改，也不会影响类变量的值，更不会影响其他对象中实例变量的值。

和动态为类添加类变量不同，Python只支持为特定的对象添加实例变量。

```
class Calculator:
    name='Good calculator'
    price=20
    def change(self,name,price):
        self.name=name
        self.price=price
calc = Calculator()
calc.msg='这是一个计算器'
calc1 = Calculator()
print(calc1.msg)
```


Python OOP

实例方法

- 在类中定义的方法默认都是实例方法。
- 它最少也要包含一个 self参数，用于绑定调用此方法的实例对象。
- 通常会用对象直接调用，也可以用类名调用。
- 对象在调用实例方法时，不需要指定self参数的值，因为self就是该对象自己，但在使用类名调用实例方法时，Python不会自动为方法的第一个参数self绑定参数值，程序必须显式地为参数self传参。

```
class Calculator:
    name ='Good calculator'
    price =20
    def __init__(self,name,price):
        self.name=name
        self.p=price
    def add(self,x,y):
        result = x+y
        print(result)
calc = Calculator('calc',20)
calc.add(1,2)
Calculator.add(calc,3,4)
```


Python OOP

类方法

- 最少也要包含一个参数，通常将其命名为cls，Python会自动将类本身绑定给cls参数（而不是类对象）。
- 需要使用@classmethod进行修饰。如果没有，认定为实例方法。
- 推荐直接使用类名调用，也可以使用实例对象来调用。

```
class Calculator:
    @classmethod
    def calculate (cls):
        print('类方法calculate: ', cls)
Calculator.calculate()
calc = Calculator()
calc.calculate()
```


Python OOP

静态方法

- 静态方法需要使用@staticmethod修饰，静态方法没有类似 self、cls 这样的特殊参数
- 静态方法中无法调用任何类和对象的属性和方法。
- 静态方法的调用，既可以使用类名，也可以使用类对象。

静态方法，和普通函数没啥区别，唯一的区别就是，定义的位置被放在了类里。业务和设计上的需要

Python OOP

静态方法

```
class Calculator:
    @staticmethod
    def info(p):
        print('静态方法info: ', p)
Calculator.info('类名')
calc = Calculator()
calc.info('对象名')
```

Python OOP

继承

- 继承是面向对象的三大特征之一，也是实现代码复用的重要手段。
- 继承用于创建和原有类功能类似的新类，原有类称为父类（也可称为基类或超类），新类称为子类。
- 子类继承了父类所有的属性和方法，还可以定义自己的属性和方法。
- 继承的优点：避免重复； 提升代码复用程度

```
class 类名(父类1, 父类2,...):
  #类定义部分
```

如果在定义类时，未显式指定这个类的直接父类，则这个类默认继承object类。object类是所有Python类的父类，要么是直接父类，要么是间接父类。

Python OOP

继承

```
class Person():
    desc='人'
    def talk(self):
        print("person is talking....")
class Chinese(Person):
    desc='中国人'
    def walk(self):
        print('chinese is walking...')
c = Chinese()
c.talk()
c.walk()
```

子类可以在继承父类的基础上，添加自己拥有的新属性和方法。

Python OOP

继承、覆盖

- 子类会自动拥有父类定义的方法，但如果父类中的方法不能满足子类的需求，子类可以按照自己的方式重新实现从父类继承的方法，这就是方法的重写，也称为方法覆盖。
- 父类方法被重写后，子类对象调用的是子类而非父类中的方法。
- 子类中重写的方法不会影响父类中对应的方法。
- 在子类中重写的方法要和父类方法具有相同的方法名和参数列表。

Python OOP

继承、覆盖

```
class Person():
    desc='人'
    def talk(self):
        print("person is talking....")
class Chinese(Person):
    desc='中国人'
    def talk(self):
        print("中国人在说话....")
    def walk(self):
        print('chinese is walking...')
class English(Person):
    desc='英国人'
    def talk(self):
        print("Englishman is talking....")
c = Chinese()
c.talk()
e = English()
e.talk()
p = Person()
p.talk()
```


Python OOP

继承、覆盖

- 在子类中调用父类中被重写的实例方法
- 可以通过类名调用。通过类名调用实例方法时，需要程序显式绑定实例方法的第一个参数self。
- 可以使用super来实现调用父类中的方法，如下图示例中，可以将Person.talk(self)替换为super().talk()或super(Chinese,self).talk()。

```
class Person():
    desc='人'
    def talk(self):
        print("person is talking....")
class Chinese(Person):
    desc='中国人'
    def talk(self):
        print("中国人在说话....")
    def walk(self):
        self.talk()
        print('chinese is walking...')
    def eat(self):
        Person.talk(self)
        print('chinese is eating...')
c = Chinese()
c.walk()
c.eat()
```


Python OOP

- 子类不重写__init__()，实例化子类时，会自动调用父类定义的__init__()。
- 如果子类重写了__init()__方法，在创建子类对象时就不会调用父类的方法了。

```
class Person():
    def __init__(self, name):
        self.name=name
        print ('name: '+self.name)
    def getName(self):
        return 'Person ' + self.name
class Chinese(Person):
    def getName(self):
        return 'Chinese ' + self.name
c = Chinese('张三')
print(c.getName())
```

```
class Person():
    def __init__(self, name):
        self.name=name
        print ('name: '+self.name)
    def getName(self):
        return 'Person ' + self.name
class Chinese(Person):
    def __init__(self, name):
        self.name=name
        print ('Chinese name: '+self.name)
    def getName(self):
        return 'Chinese ' + self.name
c = Chinese('张三')
print(c.getName())
```

Python OOP

在创建子类对象时，如果想用父类的构造方法，可以通过父类名.__init__(self,参数)或使用super()来实现。

```
class Person():
    def __init__(self, name):
        self.name=name
        print ('name: '+self.name)
    def getName(self):
        return 'Person ' + self.name
class Chinese(Person):
    def __init__(self, name):
        Person.__init__(self,name)
        #super().__init__(name)
        #super(Chinese,self).__init__(name)
        self.name=name
        print ('Chinese name: '+self.name)
    def getName(self):
        return 'Chinese ' + self.name
c = Chinese('张三')
print(c.getName())
```


Python OOP

多态

```
class Duck(object):
    def fly(self):
        print("鸭子沿着地面飞起来了")

class Swan(object):
    def fly(self):
        print("天鹅在空中翱翔")

class Plane(object):
    def fly(self):
        print("飞机隆隆地起飞了")

def fly(obj):
    obj.fly()

duck = Duck()
fly(duck)

swan = Swan()
fly(swan)

plane = Plane()
fly(plane)
```


Python OOP

私有变量 Python使用下划线作为变量前缀和后缀来指定特殊变量

- __xxx__表示系统定义名字。
- __xxx表示类中的私有变量名。

- 类的成员变量分为，公有变量，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。

Python OOP

析构函数

```
def __del__(self):  #析构函数
    print("byebye~")
```

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。

Python OOP

内置方法

Python类定义了一些专用的方法，这些专用方法丰富了程序设计的功能，用于不同的应用场合。之前介绍的__init__、__del__都是类的内置方法。

内置方法	描述
__init__(self, …)	初始化对象，在创建对象时调用
__del__(self)	释放对象，在对象被删除时调用
__str__(self)	生成对象的字符串表示，在使用print语句时被调用
__repr__(self)	生成对象的官方表示，在使用print语句时被调用
__getitem__(self, key)	获取序列的所有key对应的值，等价于seq[key]
__len__(self)	在调用内联函数len()时被调用

Python OOP

内置方法

内置方法	描述
__cmp__(src, dst)	比较两个对象src和dst
__getattr__(self, name)	获取属性的值
__getattribute__(self, name)	获取属性的值，能更好地控制
__setattr__(self, name, val)	设置属性的值
__delattr__(self,name)	删除name属性
__call__(self, *args)	将实例对象作为函数调用
__gt__(self, other)	判断self对象是否大于other对象
__lt__(self, other)	判断self对象是否小于other对象
__ge__(self, other)	判断self对象是否大于或等于other对象
__le__(self, other)	判断self对象是否小于或等于other对象
__eq__(self, other)	判断self对象是否等于other对象

Python OOP

- 读取对象的某个属性时，Python会自动调用__getattr__()方法。例如，fruit.color将转换为fruit.__getattr__(color)。
- 使用赋值表达式对属性进行设置时，Python会自动调用__setattr__()方法。

Python OOP

- **Practice**

使用List实现Stack:

- 实现成员函数： push, pop, top, is_full, is_empty()
- 能支持 len(stack object)