Python 脚本编程及系统大规模自动化运维-面对对象编程

class

- 定义类
- 使用类产生对象

```
>>> class Person:
...     pass
...
>>> p = Person()
>>> print(p)
<__main__.Person instance at 0xce3560>
```

类和函数是什么关系呢？

方法

- 定义类的成员方法

```
>>> class Person:
...     def say_hi(self):
...             print('hello,how are you?')
...
>>> p=Person()
>>> p.say_hi()
hello,how are you?
```

数据

对象里包含的数据。

可以用.访问。

和方法的主要差别在于。方法需要使用()来调用，而数据不需要。

```
>>> class Person: # 定义一个类Person
...     def set_name(self,name): # 定义一个函数set_name
...             self.name=name
...     def say_hi(self): # 定义一个函数say_hi
...             print('hello,{},how are you!'.format(self.name))
...
>>> p=Person()
>>> p.set_name('shell')
>>> print(p.name)
shell
>>> p.say_hi()
hello,shell,how are you!
```

self

在python中，成员函数使用”self”来指代对象自身。类似于java和C++中的”this”。

在调用时，p.say_hi()即可。不需要传递p.say_hi(self)或者p.say_hi(p)。

在使用时，需要先将self定义为第一个参数。例如def say_hi(self):。

在函数内使用时，对象的成员需要用self.xx的方式使用，例如self.name。

self不是关键词，也不是强制名称。它只是函数的第一个参数。但是为了保持传统，请不要将他改为其他名称。

init方法

__init__是对象的第一个方法，用于初始化对象数据。

__init__函数是初始化方法，而非构造方法。

在__init__函数被调用时，self对象已经创建完毕。

在python中，对象的属性可以自由添加删除，不需要先在类里面声明。

一般而言，对象里的所有数据会在__init__方法中赋值，后面可以变更。

但是不在__init__中赋值，后面直接赋值创建属性也是合法的。

```
>>> class Person:
...     def __init__(self,name):
...             self.name=name
...     def set_name(self,name):
...             self.name=name
...     def say_hi(self):
...             print('hello,{},how are you?'.format(self.name))
...
>>> p=Person('Swaroop')
>>> print(p.name)
Swaroop
>>> p.say_hi()
hello,Swaroop,how are you?
>>>
>>> p.set_name('shell')
>>> print(p.name)
shell
>>> p.say_hi()
hello,shell,how are you?
>>> p.__init__('tom')
>>> print(p.name)
tom
>>> p.say_hi()
hello,tom,how are you?
```

练习

定义一个马克杯类，定义一个加水方法和一个喝水方法。

```
class Mug:
    def __init__(self):
        self.water = 0
    def drink(self, mass):
        self.water -= mass
    def watering(self, mass):
        self.water += mass
```

类成员和对象成员

类和对象分别拥有成员，例如数据和方法。

对象可以引用类成员，例如p.say_hi()。也可以引用对象成员，或者self.name。

当有重名时优先引用对象成员。

类成员在所有对象间共享，而对象成员只是它自己的。

继承

继承关系分为基类和继承类（也叫父类和子类）。

子类可以继承父类的成员。当子类和父类定义同名成员时，优先引用子类的。

原则上说，父类能做的事，子类一定能做（虽然行为可能有差异）。父类能出现的地方，子类一定能出现。

因为子类拥有父类的所有成员。

继承的写法为：

class Person(Animal):

```
pass
```

class 子类（父类）:

案例：学生和老师的信息分别为：

```
姓名name，年龄age，成绩mark；
姓名name，年龄age，工资salary；
```

可以看到，不同的身份信息有重复的部分，name和age；因此可以做成父类，让子类继承。

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))

    def tell(self):
        '''Tell my details.'''
        print('Name:"{}" Age:"{}"'.format(self.name, self.age))


class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Initialized Teacher: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Salary: "{:d}"'.format(self.salary))

class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Initialized Student: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Marks: "{:d}"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

# prints a blank line
print()
members = [t, s]
for member in members:
    # Works for both Teachers and Students
    member.tell()
```

```
[root@workstation0 ~]# vim jc1.py
[root@workstation0 ~]# python jc1.py
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)
()
Name:"Mrs. Shrividya" Age:"40"
Salary: "30000"
Name:"Swaroop" Age:"25"
Marks: "75"
```

练习

请为马克杯增加一个“添满”方法，可以求出需要添多少水到加满，并自动为杯子加水。

```
[root@workstation0 ~]# cat jc2.py
class Mug(object):
    capacity=300
    def __init__(self):
        self.water=0
    def drink(self,mass):
        self.water -=mass
        if self.water < 0:
            self.water=0
    def watering(self,mass):
        self.water +=mass
        if self.water > self.capacity:
            self.water=self.capacity
    def full(self):
        mass=self.capacity-self.water
        self.watering(mass)

m=Mug()
m.full()
print(m.water)
[root@workstation0 ~]# python jc2.py
300
```

练习

请定义一个运动杯类，并同样实现“喝水”，“加水”和“添满”方法。

```
class SPO(object):
    capacity=500
    def __init__(self):
        self.water=0
    def drink(self,mass):
        self.water -=mass
        if self.water < 0:
            self.water=0
    def watering(self,mass):
        self.water +=mass
        if self.water > self.capacity:
            self.water=self.capacity
    def full(self):
        mass=self.capacity-self.water
        self.watering(mass)
```

练习

考虑一下，如果“马克杯”和“运动杯”的“喝水”和“添水”行为是不一样的。

那么，“添满”行为是否一样，两边是否可以重用“添满”？

如果可以，怎么做？

```
[root@workstation0 ~]# cat jc4.py
class SPO(object):
    capacity=500
    def __init__(self):
        self.water=0
    def drink(self,mass):
        self.water -=mass
        if self.water < 0:
            self.water=0
    def watering(self,mass):
        self.water +=mass
        if self.water > self.capacity:
            self.water=self.capacity
    def full(self):
        mass=self.capacity-self.water
        self.watering(mass)

class Mug(object):
    capacity=300
    def __init__(self):
        self.water=0
    def drink(self,mass):
        self.water -=mass
        if self.water < 0:
            self.water=0
    def watering(self,mass):
        self.water +=mass
        if self.water > self.capacity:
            self.water=self.capacity
    def full(self):
        mass=self.capacity-self.water
        self.watering(mass)

m=Mug()
s=SPO()

i=100
m.watering(i)
s.watering(i)
print(m.water)
print(s.water)

m.watering(200)
s.watering(300)
print(m.water)
print(s.water)
[root@workstation0 ~]# python jc4.py
100
100
300
400


[root@workstation0 ~]# cat jc6.py
```

```
class Cup(object):

    def __init__(self):
        self.water = 0   This m

    def full(self):
        mass = self.capacity - self.water
        self.watering(mass)

    def drink(self, mass):
        self.water -= mass
        if self.water < 0:
            self.water = 0

    def watering(self, mass):
        self.water += mass
        if self.water > self.capacity:
            self.water = self.capacity


class Mug(Cup):
    capacity = 300


class SPO(Cup):
    capacity = 500


m=Mug()
s=SPO()

m.watering(200)
s.watering(300)
print(m.water)
print(s.water)
m.full()
s.full()
print(m.water)
print(s.water)
[root@workstation0 ~]# python jc6.py
200
300
300
500

[root@workstation0 ~]# cat jc5.py
class Cup(object):

    def __init__(self):
        self.water = 0

    def full(self):

        mass = self.capacity - self.water
```

```
        self.watering(mass)

class Mug(Cup):
    capacity = 300

    def drink(self, mass):
        self.water -= mass
        if self.water < 0:
            self.water = 0

    def watering(self, mass):
        self.water += mass
        if self.water > self.capacity:
            self.water = self.capacity

class SPO(Cup):
    capacity = 500

    def drink(self, mass):
        self.water -= mass
        if self.water < 0:
            self.water = 0

    def watering(self, mass):
        self.water += mass
        if self.water > self.capacity:
            self.water = self.capacity

m=Mug()
s=SPO()

m.watering(200)
s.watering(300)
print(m.water)
print(s.water)
m.full()
s.full()
print(m.water)
print(s.water)
[root@workstation0 ~]# python jc5.py
200
300
300
500
```

booboo:不同的类中如果有相同的部分，可以单独写成一个基类，这样就不用在子类中重复了。

异常

简介

Python允许在出现错误的时候，“抛”出这个错误。

错误按照调用顺序依次向上找，找到第一个合适的处理方法对错误进行处理。如果无人处理错误，则程序崩溃。

这种错误处理机制允许调用者对函数深层的错误进行容错，同时中间所有代码对这个过程无需干预。

```
>>> Print("Hello World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
>>> print("Hello World")
Hello World
```

异常输出阅读

python的大多数错误会伴随抛出异常。因此，为了解决日常在使用中碰到的种种问题，我们需要学会如何阅读异常。

异常输出通常和函数的调用顺序相同，和栈的顺序相反。

最上层的调用（最先发生的）在最上面，最后执行到的地方在最下面。最后一个异常行就是一场发生的精确地点。

最后一行是异常的名字和参数。

异常处理

常规异常处理代码段需要包含三部分内容：被监管代码体，错误类型和异常处理代码体。

当被监管代码体中发生异常时，异常被向上抛出。

向上寻找处理函数的异常属于定义的这个错误类的子类时，异常处理代码体被调用。

一般来说，除非明确知道确实要捕获所有异常，否则严禁用捕获所有异常来处理某种特定问题。

常见异常

```
AttributeError
ImportError
IndexError
KeyError
NameError
SyntaxError
IndentationError
TypeError
ValueError
```

抛出异常

自定义异常

finally

finally用于“无论是异常还是正常，以下内容必然被执行”的情况。

多用于清理。

assert

判断文件是否存在：

编写一个脚本，脚本名为test.sh
判断/tmp/test1至/tmp/test50和/etc/passwd /etc/hosts文件存在不存在，若不存在则提示错误。

[root@workstation0 tmp]# python test.py
plz input file:/etc/passwd
ok
[root@workstation0 tmp]# python test.py
plz input file:/cc
error
[root@workstation0 tmp]# cat test.py
fname=raw_input('plz input file:')
try:

```
f=open(fname)
```

except IOError:

```
print('error')
```

else:

```
print('ok')
```

文档

文档获得和查阅

在线文档：

https://docs.python.org/2/

https://docs.python.org/3/

本地文档：随安装版本变化。

两者冲突以本地为准，本地一定对应安装使用的版本。

库查Library Reference，这是最主要部分。

语法特性查Language Reference。

练习

打开文档，请查阅itertools.permutations的意义，参数，返回值，用法，注意要点等信息。并向大家解释。

在windows上下载http://classroom.example.com/materials/python2.7/python279.chm

第三方文档查询

没有什么固定方法。

在google（注意，不是baidu）上搜索关键词。找一个比较像的官网。找到文档。

可以参考pypi，很多第三方库可以在上面找到。里面往往带有文档地址。

缺点是，pypi上搜出来的重名库太多，很难搞清楚哪个才是你要的。

Python2和Python3的差异

简述

Python2中有很多固有的设计问题，例如：

1. print是内置关键词。一般来说，关键词越少越好，因为关键词越少，语言的内核越简洁。
2. 混同了bytes和unicode。
3. /是整数除法。

这些问题在Python3中逐步给予了修复。

常规来说，修复问题最重要的是“向下兼容，逐步进行”。

然而上述问题几乎全部都是语言本质问题，不对语言进行伤筋动骨的大改是没有办法修复的。

因此Python3的预订是“不和Python2兼容”，这造成了Python社区目前2/3分裂的现状。

Python2和Python3有很多细节差异。但是大致来说，最主要就是上面提到的三项。

- Python2的print由关键字改为了函数。因此print ‘xxx’的写法就不合法了。
- 在Python3中，所有的字符串默认都是unicode。要处理原生数据需要用bytes。因此很多在Python2中能够很方便就处理过去的地方需要仔细思考是unicode还是bytes。
- Python3中/不是整数除法。

在下面这个连接里，收录了Python2和Python3的其他一些细节差异。

https://wiki.python.org/moin/Python2orPython3

Python2到3迁移

Python3带有2to3脚本，可以完成很多项目的迁移。

但是对于某些情况，他仍然不能自动的完成所有工作。

2到3迁移的主要问题在于，目前有很多库，仍然没有完成Python3的迁移工作。这导致使用这些库编写的程序很难在Python3上找到更好的（或者更习惯的）替代产品。

而这件事情是2to3脚本无法自动完成的。

我该用哪个版本/哪个更好

任何能让你不添加额外的麻烦把工作做下来的版本都是好的。

如果Python2和Python3一样好，那么Python3更好。

因为很显然，Python社区已经宣布Python2将停止维护。在可见的时间内，显然都要使用Python3的写法。

如果能够不添加额外麻烦的情况下使用Python3，这可能为将来的维护带来便利。