Python教程

目录

[Python教程 1](#_Toc470266139)

[1 Python简介 4](#_Toc470266140)

[2 模块 4](#_Toc470266141)

[2.1 使用模块 4](#_Toc470266142)

[2.1.1 作用域 6](#_Toc470266143)

[2.2 安装第三方模块 7](#_Toc470266144)

[2.2.1 模块搜索路径 7](#_Toc470266145)

[3 面向对象编程 8](#_Toc470266146)

[3.1 类和实例 9](#_Toc470266147)

[3.1.1 数据封装 10](#_Toc470266148)

[3.2 访问限制 10](#_Toc470266149)

[3.3 继承和多态 12](#_Toc470266150)

[3.4 获取对象信息 13](#_Toc470266151)

[3.4.1 判断对象类型type() 13](#_Toc470266152)

[3.4.2 使用isinstance() 14](#_Toc470266153)

[3.4.3 使用dir() 15](#_Toc470266154)

[3.5 实例属性和类属性 17](#_Toc470266155)

[4 面向对象高级编程 17](#_Toc470266156)

[4.1 使用\_\_slots\_\_ 17](#_Toc470266157)

[4.2 使用@property 19](#_Toc470266158)

[4.3 多重继承 21](#_Toc470266159)

[4.3.1 MixIn 22](#_Toc470266160)

[4.4 定制类 22](#_Toc470266161)

[4.4.1 \_\_str\_\_ 23](#_Toc470266162)

[4.4.2 \_\_iter\_\_ 24](#_Toc470266163)

[4.4.3 \_\_getitem\_\_ 24](#_Toc470266164)

[4.4.4 \_\_getattr\_\_ 26](#_Toc470266165)

[4.4.5 \_\_call\_\_ 28](#_Toc470266166)

[4.5 使用枚举类 29](#_Toc470266167)

[5 错误、调试和测试 30](#_Toc470266168)

[5.1 错误处理 30](#_Toc470266169)

[5.2 调试 31](#_Toc470266170)

[5.2.1 print() 31](#_Toc470266171)

[5.2.2 断言 31](#_Toc470266172)

[5.2.3 logging 32](#_Toc470266173)

[5.2.4 pdb 32](#_Toc470266174)

[5.2.5 pdb.set\_trace() 33](#_Toc470266175)

[5.2.6 IDE 33](#_Toc470266176)

[6 项目练习 34](#_Toc470266177)

[6.1 图片转字符画 34](#_Toc470266178)

[6.2 200行Python代码实现2048 35](#_Toc470266179)

[6.2.1 用户行为 35](#_Toc470266180)

[6.2.2 状态机 36](#_Toc470266181)

[6.2.3 用户输入处理 36](#_Toc470266182)

[6.2.4 矩阵转置与矩阵逆转 36](#_Toc470266183)

[6.2.5 创建棋盘 37](#_Toc470266184)

[6.2.6 棋盘操作 37](#_Toc470266185)

[6.2.7 重置棋盘 37](#_Toc470266186)

[6.2.8 一行向左合并 37](#_Toc470266187)

[6.2.9 棋盘走一步 38](#_Toc470266188)

[6.2.10 判断输赢 39](#_Toc470266189)

[6.2.11 判断能否移动 39](#_Toc470266190)

[6.2.12 绘制游戏界面 39](#_Toc470266191)

[6.2.13 全部代码 40](#_Toc470266192)

[6.3 pygame 开发打飞机游戏 45](#_Toc470266193)

[6.3.1 游戏角色 46](#_Toc470266194)

[6.3.2 游戏功能 46](#_Toc470266195)

[6.3.3 代码实现 46](#_Toc470266196)

[6.4 生成汉字图片字库 58](#_Toc470266197)

[6.4.1 环境搭建 58](#_Toc470266198)

[6.4.2 使用pygame渲染字体 58](#_Toc470266199)

[6.5 基于 Python 的文件备份 60](#_Toc470266200)

[6.6 Kaggle入门：泰坦尼克号幸存者项目 61](#_Toc470266201)

[6.7 基于scrapy爬虫的天气数据采集 61](#_Toc470266202)

[6.7.1 安装所需软件 61](#_Toc470266203)

[6.7.2 安装Scrapy 62](#_Toc470266204)

[6.7.3 创建scrapy项目 62](#_Toc470266205)

[6.7.4 实现爬虫 62](#_Toc470266206)

[6.7.5 中文处理 64](#_Toc470266207)

[6.7.6 保存结果 64](#_Toc470266208)

[6.8 Python文本解析器 65](#_Toc470266209)

[6.8.1 文本块生成器 65](#_Toc470266210)

[6.8.2 处理程序 66](#_Toc470266211)

[6.8.3 规则 68](#_Toc470266212)

[6.8.4 解析 70](#_Toc470266213)

[6.8.5 运行 71](#_Toc470266214)

[6.8.6 程序说明 72](#_Toc470266215)

[6.9 破解验证码 74](#_Toc470266216)

[6.9.1 提取文本图片 74](#_Toc470266217)

[6.9.2 提取单个字符图片 76](#_Toc470266218)

[6.9.3 AI 与向量空间图像识别 77](#_Toc470266219)

[6.10 端口扫描 80](#_Toc470266220)

[6.11 Python实现简单的Web服务器 81](#_Toc470266221)

[6.11.1 你好, web 83](#_Toc470266222)

[6.11.2 显示请求的信息 85](#_Toc470266223)

[6.11.3 响应静态页面 87](#_Toc470266224)

[6.11.4 在根url显示首页内容 89](#_Toc470266225)

[6.11.5 CGI协议 93](#_Toc470266226)

[6.11.6 代码重构 98](#_Toc470266227)

# Python简介

# 模块

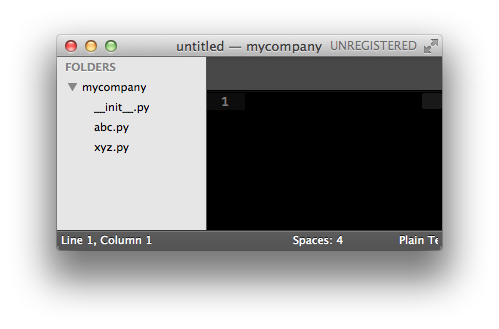
为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里。在Python中，一个.py文件就称之为一个模块（Module）。

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。使用模块还可以避免函数名和变量名冲突。但是也要注意，尽量不要与内置函数名字冲突。

Python所有内置函数：

<https://docs.python.org/3/library/functions.html>

Python引入了按目录来组织模块的方法，称为包（Package）。引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。



请注意，每一个包目录下面都会有一个\_\_init\_\_.py的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。\_\_init\_\_.py可以是空文件，也可以有Python代码，因为\_\_init\_\_.py本身就是一个模块，而它的模块名就是mycompany。

自己创建模块时要注意命名，不能和Python自带的模块名称冲突。例如，系统自带了sys模块，自己的模块就不可命名为sys.py，否则将无法导入系统自带的sys模块。

## 使用模块

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

我们以内建的sys模块为例，编写一个hello的模块：

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

#表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

' a test module '

\_\_author\_\_ **=** 'Michael Liao' #使用\_\_author\_\_变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

#以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

#后面开始就是真正的代码部分。

**import** sys #使用sys模块的第一步，就是导入该模块

**def** test**():**

args **=** sys**.**argv

**if** len**(**args**)==**1**:**

**print(**'Hello, world!'**)**

**elif** len**(**args**)==**2**:**

**print(**'Hello, %s!' **%** args**[**1**])**

**else:**

**print(**'Too many arguments!'**)**

**if** \_\_name\_\_**==**'\_\_main\_\_'**:**

test**()**

导入sys模块后，我们就有了变量sys指向该模块，利用sys这个变量，就可以访问sys模块的所有功能。

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python3 hello.py获得的sys.argv就是['hello.py']；

运行python3 hello.py Michael获得的sys.argv就是['hello.py', 'Michael]。

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

我们可以用命令行运行hello.py看看效果：

$ python3 hello.py

Hello, world!

$ python hello.py Michael

Hello, Michael!

如果启动Python交互环境，再导入hello模块：

$ python3

Python 3.4.3 **(**v3.4.3**:**9b73f1c3e601**,** Feb 23 2015**,** 02**:**52**:**03**)**

**[**GCC 4.2.1 **(**Apple Inc**.** build 5666**)** **(**dot 3**)]** on darwin

Type "help"**,** "copyright"**,** "credits" **or** "license" **for** more information**.**

**>>>** **import** hello

**>>>**

导入时，没有打印Hello, word!，因为没有执行test()函数。

调用hello.test()时，才能打印出Hello, word!：

**>>>** hello**.**test**()**

Hello**,** world**!**

### 作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过\_前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：abc，x123，PI等；

类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的\_\_author\_\_，\_\_name\_\_就是特殊变量，hello模块定义的文档注释也可以用特殊变量\_\_doc\_\_访问，我们自己的变量一般不要用这种变量名；

类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

**def** \_private\_1**(**name**):**

**return** 'Hello, %s' **%** name

**def** \_private\_2**(**name**):**

**return** 'Hi, %s' **%** name

**def** greeting**(**name**):**

**if** len**(**name**)** **>** 3**:**

**return** \_private\_1**(**name**)**

**else:**

**return** \_private\_2**(**name**)**

我们在模块里公开greeting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用greeting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

## 安装第三方模块

在Python中，安装第三方模块，是通过包管理工具pip完成的。

如果你正在使用Mac或Linux，安装pip本身这个步骤就可以跳过了。

如果你正在使用Windows，请参考[安装Python](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014316090478912dab2a3a9e8f4ed49d28854b292f85bb000)一节的内容，确保安装时勾选了pip和Add python.exe to Path。

在命令提示符窗口下尝试运行pip，如果Windows提示未找到命令，可以重新运行安装程序添加pip。

注意：Mac或Linux上有可能并存Python 3.x和Python 2.x，因此对应的pip命令是pip3。

现在，让我们来安装一个第三方库——Python Imaging Library，这是Python下非常强大的处理图像的工具库。不过，PIL目前只支持到Python 2.7，并且有年头没有更新了，因此，基于PIL的Pillow项目开发非常活跃，并且支持最新的Python 3。

一般来说，第三方库都会在Python官方的[pypi.python.org](https://pypi.python.org/)网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Pillow的名称叫[Pillow](https://pypi.python.org/pypi/Pillow/)，因此，安装Pillow的命令就是：

pip install Pillow

有了Pillow，处理图片易如反掌。随便找个图片生成缩略图：

**>>>** **from** PIL **import** Image

**>>>** im **=** Image**.**open**(**'test.png'**)**

**>>>** **print(**im**.**format**,** im**.**size**,** im**.**mode**)**

PNG **(**400**,** 300**)** RGB

**>>>** im**.**thumbnail**((**200**,** 100**))**

**>>>** im**.**save**(**'thumb.jpg'**,** 'JPEG'**)**

其他常用的第三方库还有MySQL的驱动：mysql-connector-python，用于科学计算的NumPy库：numpy，用于生成文本的模板工具Jinja2，等等。

### 模块搜索路径

当我们试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

**>>>** **import** mymodule

Traceback **(**most recent call last**):**

File "<stdin>"**,** line 1**,** **in** **<**module**>**

ImportError**:** No module named mymodule

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在sys模块的path变量中：

**>>>** **import** sys

**>>>** sys**.**path

**[**''**,** '/usr/lib/python3.4'**,** '/usr/lib/python3.4/plat-x86\_64-linux-gnu'**,** '/usr/lib/python3.4/lib-dynload'**,** '/usr/local/lib/python3.4/dist-packages'**,** '/usr/lib/python3/dist-packages'**]**

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改sys.path，添加要搜索的目录：

**>>>** **import** sys

**>>>** sys**.**path**.**append**(**'/Users/michael/my\_py\_scripts'**)**

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量PYTHONPATH，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

# 面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计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。

而面向对象的程序设计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。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个dict表示：

std1 **=** **{** 'name'**:** 'Michael'**,** 'score'**:** 98 **}**

std2 **=** **{** 'name'**:** 'Bob'**,** 'score'**:** 81 **}**

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

**def** print\_score**(**std**):**

**print(**'%s: %s' **%** **(**std**[**'name'**],** std**[**'score'**]))**

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是Student这种数据类型应该被视为一个对象，这个对象拥有name和score这两个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个print\_score消息，让对象自己把自己的数据打印出来。

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**,** name**,** score**):**

self**.**name **=** name

self**.**score **=** score

**def** print\_score**(**self**):**

**print(**'%s: %s' **%** **(**self**.**name**,** self**.**score**))**

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

bart **=** Student**(**'Bart Simpson'**,** 59**)**

lisa **=** Student**(**'Lisa Simpson'**,** 87**)**

bart**.**print\_score**()**

lisa**.**print\_score**()**

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student。所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

## 类和实例

在Python中，定义类是通过class关键字：

**class** **Student(**object**):**

**pass**

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，score等属性绑上去：

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**,** name**,** score**):**

self**.**name **=** name

self**.**score **=** score

注意到\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去：

**>>>** bart **=** Student**(**'Bart Simpson'**,** 59**)**

**>>>** bart**.**name

'Bart Simpson'

**>>>** bart**.**score

59

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

### 数据封装

Student实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法：

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**,** name**,** score**):**

self**.**name **=** name

self**.**score **=** score

**def** print\_score**(**self**):**

**print(**'%s: %s' **%** **(**self**.**name**,** self**.**score**))**

要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入：

**>>>** bart**.**print\_score**()**

Bart Simpson**:** 59

这样一来，我们从外部看Student类，就只需要知道，创建实例需要给出name和score，而如何打印，都是在Student类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给Student类增加新的方法，比如get\_grade：

**class** **Student(**object**):**

**...**

**def** get\_grade**(**self**):**

**if** self**.**score **>=** 90**:**

**return** 'A'

**elif** self**.**score **>=** 60**:**

**return** 'B'

**else:**

**return** 'C'

## 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，所以，我们把Student类改一改：

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**,** name**,** score**):**

self**.**\_\_name **=** name

self**.**\_\_score **=** score

**def** print\_score**(**self**):**

**print(**'%s: %s' **%** **(**self**.**\_\_name**,** self**.**\_\_score**))**

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.\_\_name和实例变量.\_\_score了：

**>>>** bart **=** Student**(**'Bart Simpson'**,** 98**)**

**>>>** bart**.**\_\_name

Traceback **(**most recent call last**):**

File "<stdin>"**,** line 1**,** **in** **<**module**>**

AttributeError**:** 'Student' object has no attribute '\_\_name'

这样就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。

但是如果外部代码要获取和修改name和score怎么办？可以给Student类增加get\_name/set\_name和get\_score/set\_score这样的方法：

**class** **Student(**object**):**

**...**

**def** get\_name**(**self**):**

**return** self**.**\_\_name

**def** set\_name**(**self**,** name**):**

self**.**\_\_name **=** name

**def** get\_score**(**self**):**

**return** self**.**\_\_score

**def** set\_score**(**self**,** score**):**

**if** 0 **<=** score **<=** 100**:**

self**.**\_\_score **=** score

**else:**

**raise** ValueError**(**'bad score'**)**

需要注意的是，在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。

## 继承和多态

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）。

比如，我们已经编写了一个名为Animal的class，有一个run()方法可以直接打印：

**class** **Animal(**object**):**

**def** run**(**self**):**

**print(**'Animal is running...'**)**

当我们需要编写Dog和Cat类时，就可以直接从Animal类继承：

**class** **Dog(**Animal**):**

**pass**

**class** **Cat(**Animal**):**

**pass**

继承的好处：

1. 子类获得了父类的全部功能。
2. 多态。

**class** **Dog(**Animal**):**

**def** run**(**self**):**

**print(**'Dog is running...'**)**

**class** **Cat(**Animal**):**

**def** run**(**self**):**

**print(**'Cat is running...'**)**

再次运行，结果如下：

Dog **is** running**...**

Cat **is** running**...**

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()。

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。但是，反过来就不行：

a **=** list**()** # a是list类型

b **=** Animal**()** # b是Animal类型

c **=** Dog**()** # c是Dog类型

**>>>** isinstance**(**a**,** list**)**

**True**

**>>>** isinstance**(**b**,** Animal**)**

**True**

**>>>** isinstance**(**c**,** Dog**)**

**True**

**>>>** isinstance**(**c**,** Animal**)**

**True**

**>>>** b **=** Animal**()**

**>>>** isinstance**(**b**,** Dog**)**

**False**

## 获取对象信息

### 判断对象类型type()

首先，我们来判断对象类型，使用type()函数：

基本类型都可以用type()判断：

**>>>** type**(**123**)**

**<class** 'int'**>**

**>>>** type**(**'str'**)**

**<class** 'str'**>**

**>>>** type**(None)**

**<**type**(None)** 'NoneType'**>**

**>>>** type**(**abs**)**

**<class** 'builtin\_function\_or\_method'**>**

**>>>** type**(**a**)**

**<class** '\_\_main\_\_.Animal'**>**

但是type()函数返回的是什么类型呢？它返回对应的Class类型。如果我们要在if语句中判断，就需要比较两个变量的type类型是否相同：

**>>>** type**(**123**)==**type**(**456**)**

**True**

**>>>** type**(**123**)==**int

**True**

**>>>** type**(**'abc'**)==**type**(**'123'**)**

**True**

**>>>** type**(**'abc'**)==**str

**True**

**>>>** type**(**'abc'**)==**type**(**123**)**

**False**

判断基本数据类型可以直接写int，str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

**>>>** **import** types

**>>>** **def** fn**():**

**...** **pass**

**...**

**>>>** type**(**fn**)==**types**.**FunctionType

**True**

**>>>** type**(**abs**)==**types**.**BuiltinFunctionType

**True**

**>>>** type**(lambda** x**:** x**)==**types**.**LambdaType

**True**

**>>>** type**((**x **for** x **in** range**(**10**)))==**types**.**GeneratorType

**True**

### 使用isinstance()

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。

object **->** Animal **->** Dog **->** Husky

**>>>** a **=** Animal**()**

**>>>** d **=** Dog**()**

**>>>** h **=** Husky**()**

**>>>** isinstance**(**h**,** Husky**)**

**True**

**>>>** isinstance**(**h**,** Dog**)**

**True**

**>>>** isinstance**(**h**,** Animal**)**

**True**

**>>>** isinstance**(**d**,** Dog**)** **and** isinstance**(**d**,** Animal**)**

**True**

**>>>** isinstance**(**d**,** Husky**)**

**False**

#能用type()判断的基本类型也可以用isinstance()判断

**>>>** isinstance**(**'a'**,** str**)**

**True**

**>>>** isinstance**(**123**,** int**)**

**True**

**>>>** isinstance**(**b'a'**,** bytes**)**

**True**

#并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple

**>>>** isinstance**([**1**,** 2**,** 3**],** **(**list**,** tuple**))**

**True**

**>>>** isinstance**((**1**,** 2**,** 3**),** **(**list**,** tuple**))**

**True**

### 使用dir()

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

**>>>** dir**(**'ABC'**)**

**[**'\_\_add\_\_'**,** '\_\_class\_\_'**,** '\_\_contains\_\_'**,** '\_\_delattr\_\_'**,** '\_\_dir\_\_'**,** '\_\_doc\_\_'**,** '\_\_eq\_\_'**,** '\_\_format\_\_'**,** '\_\_ge\_\_'**,** '\_\_getattribute\_\_'**,** '\_\_getitem\_\_'**,** '\_\_getnewargs\_\_'**,** '\_\_gt\_\_'**,** '\_\_hash\_\_'**,** '\_\_init\_\_'**,** '\_\_iter\_\_'**,** '\_\_le\_\_'**,** '\_\_len\_\_'**,** '\_\_lt\_\_'**,** '\_\_mod\_\_'**,** '\_\_mul\_\_'**,** '\_\_ne\_\_'**,** '\_\_new\_\_'**,** '\_\_reduce\_\_'**,** '\_\_reduce\_ex\_\_'**,** '\_\_repr\_\_'**,** '\_\_rmod\_\_'**,** '\_\_rmul\_\_'**,** '\_\_setattr\_\_'**,** '\_\_sizeof\_\_'**,** '\_\_str\_\_'**,** '\_\_subclasshook\_\_'**,** 'capitalize'**,** 'casefold'**,** 'center'**,** 'count'**,** 'encode'**,** 'endswith'**,** 'expandtabs'**,** 'find'**,** 'format'**,** 'format\_map'**,** 'index'**,** 'isalnum'**,** 'isalpha'**,** 'isdecimal'**,** 'isdigit'**,** 'isidentifier'**,** 'islower'**,** 'isnumeric'**,** 'isprintable'**,** 'isspace'**,** 'istitle'**,** 'isupper'**,** 'join'**,** 'ljust'**,** 'lower'**,** 'lstrip'**,** 'maketrans'**,** 'partition'**,** 'replace'**,** 'rfind'**,** 'rindex'**,** 'rjust'**,** 'rpartition'**,** 'rsplit'**,** 'rstrip'**,** 'split'**,** 'splitlines'**,** 'startswith'**,** 'strip'**,** 'swapcase'**,** 'title'**,** 'translate'**,** 'upper'**,** 'zfill'**]**

类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法，所以，下面的代码是等价的：

**>>>** len**(**'ABC'**)**

3

**>>>** 'ABC'**.**\_\_len\_\_**()**

3

我们自己写的类，如果也想用len(myObj)的话，就自己写一个\_\_len\_\_()方法：

**>>>** **class** **MyDog(**object**):**

**...** **def** \_\_len\_\_**(**self**):**

**...** **return** 100

**...**

**>>>** dog **=** MyDog**()**

**>>>** len**(**dog**)**

100

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态：

**>>>** **class** **MyObject(**object**):**

**...** **def** \_\_init\_\_**(**self**):**

**...** self**.**x **=** 9

**...** **def** power**(**self**):**

**...** **return** self**.**x **\*** self**.**x

**...**

**>>>** obj **=** MyObject**()**

紧接着，可以测试该对象的属性：

**>>>** hasattr**(**obj**,** 'x'**)** # 有属性'x'吗？

**True**

**>>>** obj**.**x

9

**>>>** hasattr**(**obj**,** 'y'**)** # 有属性'y'吗？

**False**

**>>>** setattr**(**obj**,** 'y'**,** 19**)** # 设置一个属性'y'

**>>>** hasattr**(**obj**,** 'y'**)** # 有属性'y'吗？

**True**

**>>>** getattr**(**obj**,** 'y'**)** # 获取属性'y'

19

**>>>** obj**.**y # 获取属性'y'

19

如果试图获取不存在的属性，会抛出AttributeError的错误：

**>>>** getattr**(**obj**,** 'z'**)** # 获取属性'z'

Traceback **(**most recent call last**):**

File "<stdin>"**,** line 1**,** **in** **<**module**>**

AttributeError**:** 'MyObject' object has no attribute 'z'

可以传入一个default参数，如果属性不存在，就返回默认值：

**>>>** getattr**(**obj**,** 'z'**,** 404**)** # 获取属性'z'，如果不存在，返回默认值404

404

也可以获得对象的方法：

**>>>** hasattr**(**obj**,** 'power'**)** # 有属性'power'吗？

**True**

**>>>** getattr**(**obj**,** 'power'**)** # 获取属性'power'

**<**bound method MyObject**.**power of **<**\_\_main\_\_**.**MyObject object at 0x10077a6a0**>>**

**>>>** fn **=** getattr**(**obj**,** 'power'**)** # 获取属性'power'并赋值到变量fn

**>>>** fn # fn指向obj.power

**<**bound method MyObject**.**power of **<**\_\_main\_\_**.**MyObject object at 0x10077a6a0**>>**

**>>>** fn**()** # 调用fn()与调用obj.power()是一样的

81

一个正确的用法的例子如下：

**def** readImage**(**fp**):**

**if** hasattr**(**fp**,** 'read'**):**

**return** readData**(**fp**)**

**return** **None**

假设我们希望从文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。hasattr()就派上了用场。

## 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

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。

编写程序的时候，千万不要把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

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。

我们会讨论多重继承、定制类、元类等概念。

## 使用\_\_slots\_\_

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：

**class** **Student(**object**):**

**pass**

然后，尝试给实例绑定一个属性：

**>>>** s **=** Student**()**

**>>>** s**.**name **=** 'Michael' # 动态给实例绑定一个属性

**>>>** **print(**s**.**name**)**

Michael

还可以尝试给实例绑定一个方法：

**>>>** **def** set\_age**(**self**,** age**):** # 定义一个函数作为实例方法

**...** self**.**age **=** age

**...**

**>>>** **from** types **import** MethodType

**>>>** s**.**set\_age **=** MethodType**(**set\_age**,** s**)** # 给实例绑定一个方法

**>>>** s**.**set\_age**(**25**)** # 调用实例方法

**>>>** s**.**age # 测试结果

25

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

**>>>** s2 **=** Student**()** # 创建新的实例

**>>>** s2**.**set\_age**(**25**)** # 尝试调用方法

Traceback **(**most recent call last**):**

File "<stdin>"**,** line 1**,** **in** **<**module**>**

AttributeError**:** 'Student' object has no attribute 'set\_age'

为了给所有实例都绑定方法，可以给class绑定方法：

**>>>** **def** set\_score**(**self**,** score**):**

**...** self**.**score **=** score

**...**

**>>>** Student**.**set\_score **=** set\_score

给class绑定方法后，所有实例均可调用：

**>>>** s**.**set\_score**(**100**)**

**>>>** s**.**score

100

**>>>** s2**.**set\_score**(**99**)**

**>>>** s2**.**score

99

通常情况下，上面的set\_score方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

但是，如果我们想要限制实例的属性怎么办？比如，只允许对Student实例添加name和age属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性：

**class** **Student(**object**):**

\_\_slots\_\_ **=** **(**'name'**,** 'age'**)** # 用tuple定义允许绑定的属性名称

然后，我们试试：

**>>>** s **=** Student**()** # 创建新的实例

**>>>** s**.**name **=** 'Michael' # 绑定属性'name'

**>>>** s**.**age **=** 25 # 绑定属性'age'

**>>>** s**.**score **=** 99 # 绑定属性'score'

Traceback **(**most recent call last**):**

File "<stdin>"**,** line 1**,** **in** **<**module**>**

AttributeError**:** 'Student' object has no attribute 'score'

由于'score'没有被放到\_\_slots\_\_中，所以不能绑定score属性，试图绑定score将得到AttributeError的错误。

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

**>>>** **class** **GraduateStudent(**Student**):**

**...** **pass**

**...**

**>>>** g **=** GraduateStudent**()**

**>>>** g**.**score **=** 9999

除非在子类中也定义\_\_slots\_\_，这样，子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_。

## 使用@property

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

s **=** Student**()**

s**.**score **=** 9999

这显然不合逻辑。为了限制score的范围，可以通过一个set\_score()方法来设置成绩，再通过一个get\_score()来获取成绩，这样，在set\_score()方法里，就可以检查参数：

**class** **Student(**object**):**

**def** get\_score**(**self**):**

**return** self**.**\_score

**def** set\_score**(**self**,** value**):**

**if** **not** isinstance**(**value**,** int**):**

**raise** ValueError**(**'score must be an integer!'**)**

**if** value **<** 0 **or** value **>** 100**:**

**raise** ValueError**(**'score must between 0 ~ 100!'**)**

self**.**\_score **=** value

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

**>>>** s **=** Student**()**

**>>>** s**.**set\_score**(**60**)** # ok!

**>>>** s**.**get\_score**()**

60

**>>>** s**.**set\_score**(**9999**)**

Traceback **(**most recent call last**):**

**...**

ValueError**:** score must between 0 **~** 100**!**

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的@property装饰器就是负责把一个方法变成属性调用的：

**class** **Student(**object**):**

*@property*

**def** score**(**self**):**

**return** self**.**\_score

*@score.setter*

**def** score**(**self**,** value**):**

**if** **not** isinstance**(**value**,** int**):**

**raise** ValueError**(**'score must be an integer!'**)**

**if** value **<** 0 **or** value **>** 100**:**

**raise** ValueError**(**'score must between 0 ~ 100!'**)**

self**.**\_score **=** value

@property的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

**>>>** s **=** Student**()**

**>>>** s**.**score **=** 60 # OK，实际转化为s.set\_score(60)

**>>>** s**.**score # OK，实际转化为s.get\_score()

60

**>>>** s**.**score **=** 9999

Traceback **(**most recent call last**):**

**...**

ValueError**:** score must between 0 **~** 100**!**

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

**class** **Student(**object**):**

*@property*

**def** birth**(**self**):**

**return** self**.**\_birth

*@birth.setter*

**def** birth**(**self**,** value**):**

self**.**\_birth **=** value

*@property*

**def** age**(**self**):**

**return** 2015 **-** self**.**\_birth

上面的birth是可读写属性，而age就是一个**只读**属性，因为age可以根据birth和当前时间计算出来。

## 多重继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

假设我们要实现以下4种动物：

* Dog - 狗狗；
* Bat - 蝙蝠；
* Parrot - 鹦鹉；
* Ostrich - 鸵鸟。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

**class** **Animal(**object**):**

**pass**

# 大类:

**class** **Mammal(**Animal**):**

**pass**

**class** **Bird(**Animal**):**

**pass**

# 各种动物:

**class** **Dog(**Mammal**):**

**pass**

**class** **Bat(**Mammal**):**

**pass**

**class** **Parrot(**Bird**):**

**pass**

**class** **Ostrich(**Bird**):**

**pass**

现在，我们要给动物再加上Runnable和Flyable的功能，只需要先定义好Runnable和Flyable的类：

**class** **Runnable(**object**):**

**def** run**(**self**):**

**print(**'Running...'**)**

**class** **Flyable(**object**):**

**def** fly**(**self**):**

**print(**'Flying...'**)**

对于需要Runnable功能的动物，就多继承一个Runnable，例如Dog：

**class** **Dog(**Mammal**,** Runnable**):**

**pass**

对于需要Flyable功能的动物，就多继承一个Flyable，例如Bat：

**class** **Bat(**Mammal**,** Flyable**):**

**pass**

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

### MixIn

在设计类的继承关系时，通常，主线都是单一继承下来的，例如，Ostrich继承自Bird。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让Ostrich除了继承自Bird外，再同时继承Runnable。这种设计通常称之为MixIn。

为了更好地看出继承关系，我们把Runnable和Flyable改为RunnableMixIn和FlyableMixIn。类似的，你还可以定义出肉食动物CarnivorousMixIn和植食动物HerbivoresMixIn，让某个动物同时拥有好几个MixIn：

**class** **Dog(**Mammal**,** RunnableMixIn**,** CarnivorousMixIn**):**

**pass**

MixIn的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个MixIn的功能，而不是设计多层次的复杂的继承关系。

Python自带的很多库也使用了MixIn。举个例子，Python自带了TCPServer和UDPServer这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由ForkingMixIn和ThreadingMixIn提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

**class** **MyTCPServer(**TCPServer**,** ForkingMixIn**):**

**pass**

编写一个多线程模式的UDP服务，定义如下：

**class** **MyUDPServer(**UDPServer**,** ThreadingMixIn**):**

**pass**

如果你打算搞一个更先进的协程模型，可以编写一个CoroutineMixIn：

**class** **MyTCPServer(**TCPServer**,** CoroutineMixIn**):**

**pass**

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

## 定制类

看到类似\_\_slots\_\_这种形如\_\_xxx\_\_的变量或者函数名就要注意，这些在Python中是有特殊用途的。

\_\_slots\_\_我们已经知道怎么用了，\_\_len\_\_()方法我们也知道是为了能让class作用于len()函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

### \_\_str\_\_

我们先定义一个Student类，打印一个实例：

**>>>** **class** **Student(**object**):**

**...** **def** \_\_init\_\_**(**self**,** name**):**

**...** self**.**name **=** name

**...**

**>>>** **print(**Student**(**'Michael'**))**

**<**\_\_main\_\_**.**Student object at 0x109afb190**>**

打印出一堆<\_\_main\_\_.Student object at 0x109afb190>，不好看。

怎么才能打印得好看呢？只需要定义好\_\_str\_\_()方法，返回一个好看的字符串就可以了：

**>>>** **class** **Student(**object**):**

**...** **def** \_\_init\_\_**(**self**,** name**):**

**...** self**.**name **=** name

**...** **def** \_\_str\_\_**(**self**):**

**...** **return** 'Student object (name: %s)' **%** self**.**name

**...**

**>>>** **print(**Student**(**'Michael'**))**

Student object **(**name**:** Michael**)**

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用print，打印出来的实例还是不好看：

**>>>** s **=** Student**(**'Michael'**)**

**>>>** s

**<**\_\_main\_\_**.**Student object at 0x109afb310**>**

这是因为直接显示变量调用的不是\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。

解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以，有个偷懒的写法：

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**,** name**):**

self**.**name **=** name

**def** \_\_str\_\_**(**self**):**

**return** 'Student object (name=%s)' **%** self**.**name

\_\_repr\_\_ **=** \_\_str\_\_

### \_\_iter\_\_

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

**class** **Fib(**object**):**

**def** \_\_init\_\_**(**self**):**

self**.**a**,** self**.**b **=** 0**,** 1 # 初始化两个计数器a，b

**def** \_\_iter\_\_**(**self**):**

**return** self # 实例本身就是迭代对象，故返回自己

**def** \_\_next\_\_**(**self**):**

self**.**a**,** self**.**b **=** self**.**b**,** self**.**a **+** self**.**b # 计算下一个值

**if** self**.**a **>** 100000**:** # 退出循环的条件

**raise** StopIteration**();**

**return** self**.**a # 返回下一个值

**>>>** **for** n **in** Fib**():**

**...** **print(**n**)**

**...**

1

1

2

3

5

**...**

46368

75025

### \_\_getitem\_\_

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

**>>>** Fib**()[**5**]**

Traceback **(**most recent call last**):**

File "<stdin>"**,** line 1**,** **in** **<**module**>**

TypeError**:** 'Fib' object does **not** support indexing

要表现得像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法：

**class** **Fib(**object**):**

**def** \_\_getitem\_\_**(**self**,** n**):**

a**,** b **=** 1**,** 1

**for** x **in** range**(**n**):**

a**,** b **=** b**,** a **+** b

**return** a

现在，就可以按下标访问数列的任意一项了：

**>>>** f **=** Fib**()**

**>>>** f**[**0**]**

1

**>>>** f**[**1**]**

1

**>>>** f**[**2**]**

2

**>>>** f**[**3**]**

3

**>>>** f**[**10**]**

89

**>>>** f**[**100**]**

573147844013817084101

但是list有个神奇的切片方法：

**>>>** list**(**range**(**100**))[**5**:**10**]**

**[**5**,** 6**,** 7**,** 8**,** 9**]**

对于Fib却报错。原因是\_\_getitem\_\_()传入的参数可能是一个int，也可能是一个切片对象slice，所以要做判断：

**class** **Fib(**object**):**

**def** \_\_getitem\_\_**(**self**,** n**):**

**if** isinstance**(**n**,** int**):** # n是索引

a**,** b **=** 1**,** 1

**for** x **in** range**(**n**):**

a**,** b **=** b**,** a **+** b

**return** a

**if** isinstance**(**n**,** slice**):** # n是切片

start **=** n**.**start

stop **=** n**.**stop

**if** start **is** **None:**

start **=** 0

a**,** b **=** 1**,** 1

L **=** **[]**

**for** x **in** range**(**stop**):**

**if** x **>=** start**:**

L**.**append**(**a**)**

a**,** b **=** b**,** a **+** b

**return** L

现在试试Fib的切片：

**>>>** f **=** Fib**()**

**>>>** f**[**0**:**5**]**

**[**1**,** 1**,** 2**,** 3**,** 5**]**

**>>>** f**[:**10**]**

**[**1**,** 1**,** 2**,** 3**,** 5**,** 8**,** 13**,** 21**,** 34**,** 55**]**

但是没有对step参数作处理：

**>>>** f**[:**10**:**2**]**

**[**1**,** 1**,** 2**,** 3**,** 5**,** 8**,** 13**,** 21**,** 34**,** 55**,** 89**]**

也没有对负数作处理，所以，要正确实现一个\_\_getitem\_\_()还是有很多工作要做的。

此外，如果把对象看成dict，\_\_getitem\_\_()的参数也可能是一个可以作key的object，例如str。

与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

### \_\_getattr\_\_

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义Student类：

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**):**

self**.**name **=** 'Michael'

调用name属性，没问题，但是，调用不存在的score属性，就有问题了：

**>>>** s **=** Student**()**

**>>>** **print(**s**.**name**)**

Michael

**>>>** **print(**s**.**score**)**

Traceback **(**most recent call last**):**

**...**

AttributeError**:** 'Student' object has no attribute 'score'

错误信息很清楚地告诉我们，没有找到score这个attribute。

要避免这个错误，除了可以加上一个score属性外，Python还有另一个机制，那就是写一个\_\_getattr\_\_()方法，动态返回一个属性。修改如下：

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**):**

self**.**name **=** 'Michael'

**def** \_\_getattr\_\_**(**self**,** attr**):**

**if** attr**==**'score'**:**

**return** 99

当调用不存在的属性时，比如score，Python解释器会试图调用\_\_getattr\_\_(self, 'score')来尝试获得属性，这样，我们就有机会返回score的值：

**>>>** s **=** Student**()**

**>>>** s**.**name

'Michael'

**>>>** s**.**score

99

返回函数也是完全可以的：

**class** **Student(**object**):**

**def** \_\_getattr\_\_**(**self**,** attr**):**

**if** attr**==**'age'**:**

**return** **lambda:** 25

只是调用方式要变为：

**>>>** s**.**age**()**

25

注意，只有在没有找到属性的情况下，才调用\_\_getattr\_\_，已有的属性，比如name，不会在\_\_getattr\_\_中查找。

此外，注意到任意调用如s.abc都会返回None，这是因为我们定义的\_\_getattr\_\_默认返回就是None。要让class只响应特定的几个属性，我们就要按照约定，抛出AttributeError的错误：

**class** **Student(**object**):**

**def** \_\_getattr\_\_**(**self**,** attr**):**

**if** attr**==**'age'**:**

**return** **lambda:** 25

**raise** AttributeError**(**'\'Student\' object has no attribute \'%s\'' **%** attr**)**

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

* <http://api.server/user/friends>
* <http://api.server/user/timeline/list>

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的\_\_getattr\_\_，我们可以写出一个链式调用：

**class** **Chain(**object**):**

**def** \_\_init\_\_**(**self**,** path**=**''**):**

self**.**\_path **=** path

**def** \_\_getattr\_\_**(**self**,** path**):**

**return** Chain**(**'%s/%s' **%** **(**self**.**\_path**,** path**))**

**def** \_\_str\_\_**(**self**):**

**return** self**.**\_path

\_\_repr\_\_ **=** \_\_str\_\_

试试：

**>>>** Chain**().**status**.**user**.**timeline**.**list

'/status/user/timeline/list'

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

GET **/**users**/:**user**/**repos

调用时，需要把:user替换为实际用户名。如果我们能写出这样的链式调用：

Chain**().**users**(**'michael'**).**repos

### \_\_call\_\_

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用instance.method()来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

任何类，只需要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用。请看示例：

**class** **Student(**object**):**

**def** \_\_init\_\_**(**self**,** name**):**

self**.**name **=** name

**def** \_\_call\_\_**(**self**):**

**print(**'My name is %s.' **%** self**.**name**)**

**>>>** s **=** Student**(**'Michael'**)**

**>>>** s**()** # self参数不要传入

My name **is** Michael**.**

\_\_call\_\_()还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个Callable对象，比如函数和我们上面定义的带有\_\_call\_\_()的类实例：

**>>>** callable**(**Student**())**

**True**

**>>>** callable**(**max**)**

**True**

**>>>** callable**([**1**,** 2**,** 3**])**

**False**

**>>>** callable**(None)**

**False**

**>>>** callable**(**'str'**)**

**False**

通过callable()函数，我们就可以判断一个对象是否是“可调用”对象。

## 使用枚举类

当我们需要定义常量时，一个办法是用大写变量通过整数来定义，例如月份：

JAN **=** 1

FEB **=** 2

MAR **=** 3

**...**

NOV **=** 11

DEC **=** 12

好处是简单，缺点是类型是int，并且仍然是变量。

更好的方法是为这样的枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例。Python提供了Enum类来实现这个功能：

**from** enum **import** Enum

Month **=** Enum**(**'Month'**,** **(**'Jan'**,** 'Feb'**,** 'Mar'**,** 'Apr'**,** 'May'**,** 'Jun'**,** 'Jul'**,** 'Aug'**,** 'Sep'**,** 'Oct'**,** 'Nov'**,** 'Dec'**))**

这样我们就获得了Month类型的枚举类，可以直接使用Month.Jan来引用一个常量，或者枚举它的所有成员：

**for** name**,** member **in** Month**.**\_\_members\_\_**.**items**():**

**print(**name**,** '=>'**,** member**,** ','**,** member**.**value**)**

**输出：**

Jan **=>** Month**.**Jan **,** 1

Feb **=>** Month**.**Feb **,** 2

Mar **=>** Month**.**Mar **,** 3

Apr **=>** Month**.**Apr **,** 4

May **=>** Month**.**May **,** 5

Jun **=>** Month**.**Jun **,** 6

Jul **=>** Month**.**Jul **,** 7

Aug **=>** Month**.**Aug **,** 8

Sep **=>** Month**.**Sep **,** 9

Oct **=>** Month**.**Oct **,** 10

Nov **=>** Month**.**Nov **,** 11

Dec **=>** Month**.**Dec **,** 12

value属性则是自动赋给成员的int常量，默认从1开始计数。

如果需要更精确地控制枚举类型，可以从Enum派生出自定义类：

# 错误、调试和测试

## 错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套try...except...finally...的错误处理机制，Python也不例外。

**try**:

**print**('try...')

r = 10 / int('2')

**print**('result:', r)

**except** ValueError **as** e:

**print**('ValueError:', e)

**except** ZeroDivisionError **as** e:

**print**('ZeroDivisionError:', e)

**else**: #如果没有错误发生，可以在except语句块后面加一个else，当没有错误发生时，会自动执行else语句

**print**('no error!')

**finally**:

**print**('finally...')

**print**('END')

Python的错误其实也是class，所有的错误类型都继承自BaseException，所以在使用except时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。

Python所有的错误都是从BaseException类派生的，常见的错误类型和继承关系看这里：

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

使用try...except捕获错误还有一个巨大的好处，就是可以跨越多层调用，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写try...except...finally的麻烦。

## 调试

### print()

第一种方法简单直接粗暴有效，就是用print()把可能有问题的变量打印出来看看。

### 断言

凡是用print()来辅助查看的地方，都可以用断言（assert）来替代：

**def** **foo**(s):

n = int(s)

**assert** n != 0, 'n is zero!'

**return** 10 / n

**def** **main**():

foo('0')

assert的意思是，表达式n != 0应该是True，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，assert语句本身就会抛出AssertionError. 程序中如果到处充斥着assert，和print()相比也好不到哪去。不过，启动Python解释器时可以用-O参数来关闭assert：

$ python -O err.py

关闭后，你可以把所有的assert语句当成pass来看。

### logging

把print()替换为logging是第3种方式，和assert比，logging不会抛出错误，而且可以输出到文件：

**import** logging

logging.basicConfig(level=logging.INFO)

s = '0'

n = int(s)

logging.info('n = %d' % n)

**print**(10 / n)

这就是logging的好处，它允许你指定记录信息的级别，有debug，info，warning，error等几个级别，logging的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

### pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。

python -m pdb err.py

以参数-m pdb启动后，pdb定位到下一步要执行的代码-> s = '0'。输入命令l来查看代码：

(Pdb) l

1 # err.py

2 -> s = '0'

3 n = int(s)

4 **print**(10 / n)

输入命令n可以单步执行代码：

(Pdb) n

> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<module>()

-> n = int(s)

(Pdb) n

> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<module>()

-> **print**(10 / n)

任何时候都可以输入命令p 变量名来查看变量：

(Pdb) p s

'0'

(Pdb) p n

0

输入命令q结束调试，退出程序。这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

### pdb.set\_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要import pdb，然后，在可能出错的地方放一个pdb.set\_trace()，就可以设置一个断点：

# err.py

**import** pdb

s = '0'

n = int(s)

pdb.set\_trace() # 运行到这里会自动暂停

**print**(10 / n)

运行代码，程序会自动在pdb.set\_trace()暂停并进入pdb调试环境，可以用命令p查看变量，或者用命令c继续运行：

$ python err.py

> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<module>()

-> **print**(10 / n)

(Pdb) p n

0

(Pdb) c

Traceback (most recent call last):

File "err.py", line 7, **in** <module>

**print**(10 / n)

ZeroDivisionError: division by zero

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

### IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有PyCharm：

<http://www.jetbrains.com/pycharm/>

另外，[Eclipse](http://eclipse.org/)加上[pydev](http://pydev.org/)插件也可以调试Python程序。

# 网络数据采集

## 使用BeautifulSoup

我的系统为Ubuntu14.04，安装了python2.7.4和python3.4.4，系统默认的为python2，如果直接使用如下命令安装beautifulsoup，则默认是安装在python2.7.4上面：

sudo apt-get install python-bs4

如果要安装在python3.4.4上面，需要如下方式：

sudo apt-get install python3-pip

sudo pip3 install beautifulsoup4

然后验证一下是否安装成功：

python3

> **from** bs4 **import** BeautifulSoup

使用BeautifulSoup处理html，并检查与处理相应的错误：

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

**from** urllib.request **import** urlopen

**from** urllib.error **import** HTTPError

**from** bs4 **import** BeautifulSoup

**def** **getTitle**(url):

**try**:

html = urlopen(url)

**except** HTTPError **as** e:#server not found or URL can not open

**return** **None**

**try**:

bsObj = BeautifulSoup(html.read())

title = bsObj.body.h1

**except** AttributeError **as** e:#tag is None

**return** **None**

**return** title

title = getTitle("http://www.pythonscraping.com/pages/page1.html")

**if** title == **None**:

**print**("Title cound not be found")

**else**:

**print**(title)

## 复杂 HTML 解析

抽取网页内容遵循以下方法：

* 寻找“打印此页”的链接， 或者看看网站有没有 HTML 样式更友好的移动版。
* 寻找隐藏在 JavaScript 文件里的信息。 要实现这一点， 你可能需要查看网页加载的 JavaScript 文件。 我曾经要把一个网站上的街道地址（以经度和纬度呈现的） 整理成格式整洁的数组时， 查看过内嵌谷歌地图的 JavaScript 文件， 里面有每个地址的标记点。
* 虽然网页标题经常会用到， 但是这个信息也许可以从网页的 URL 链接里获取。
* 如果你要找的信息只存在于一个网站上， 别处没有， 那你确实是运气不佳。 如果不只限于这个网站， 那么你可以找找其他数据源。 有没有其他网站也显示了同样的数据？ 网站上显示的数据是不是从其他网站上抓取后攒出来的？

### 正则表达式

正则表达式常用符号

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | 含义 | 例子 | 匹配结果 |
| \* | 匹配前面的字符、 子表达式或括号里的字符 0 次或多次 | a\*b\* | aaaaaaaa， aaabbbbb， bbbbbb |
| + | 匹配前面的字符、 子表达式或括号里的字符至少 1 次 | a+b+ | aaaaaaab， aaabbbbb， abbbbbb |
| [] | 匹配任意一个字符（相当于“任选一个”） | [A-Z] \* | APPLE， CAPITALS， QWERTY |
| () | 表达式编组（在正则表达式的规则里编组会优先运行） | (a\*b) \* | aaabaab， abaaab， ababaaaaab |
| { m, n} | 匹配前面的字符、 子表达式或括号里的字符 m到 n 次（包含 m或 n） | a{ 2, 3} b{ 2, 3} | aabbb， aaabbb， aabb |
| [^] | 匹配任意一个不在中括号里的字符 | [^A-Z] \* | apple， lowercase， qwerty |
| | | 匹配任意一个由竖线分割的字符、 子表达式（注意是竖线， 不是大字字母I） | b(a| i| e) d | bad， bid， bed |
| . | 匹配任意单个字符（包括符号、 数字和空格等） | b. d | bad， bzd， b$d， b d |
| ^ | 指字符串开始位置的字符或子表达式 | ^a | apple， asdf， a |
| \ | 转义字符（把有特殊含义的字符转换成字面形式） | \. \ | \\ | . | \ |
| $ | 经常用在正则表达式的末尾， 表示“从字符串的末端匹配”。 如果不用它， 每个正则表达式实际都带着“. \* ”模式， 只会从字符串开头进行匹配。 这个符号可以看成是 ^ 符号的反义词 | [A-Z] \*[a-z]\*S | ABCabc， zzzyx， Bob |
| ?! | “不包含”。 这个奇怪的组合通常放在字符或正则表达式前面， 表示字符不能出现在目 标字符串里。 这个符号比较难用， 字符通常会在字符串的不同部位出现。 如果要在整个字符串中全部 排除某个字符， 就加上 ^ 和 $ 符号 ^((?! [AZ] ) . ) \* | 排除某个字符， 就加上 ^ 和 $ 符号 ^((?! [AZ] ) . ) \*$ | no-caps-here， $ymb0ls a4e f!ne |

## 开始采集

### 遍历单个域名

获取维基百科网站的任何页面并提取页面链接：

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

#from urllib.error import HTTPError

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

html = urlopen("http://en.wikipedia.org/wiki/Kevin\_Bacon")

bsObj = BeautifulSoup(html, 'html.parser')

**for** link **in** bsObj.findAll("a"):

**if** 'href' **in** link.attrs:

**print**(link.attrs['href'])

结果中有词条链接，也有一些我们不需要的链接，因为维基百科的每个页面都充满了侧边栏、 页眉、 页脚链接， 以及连接到分类页面、 对话页面和其他不包含词条的页面的链接。如果你仔细观察那些指向词条页面（不是指向其他内容页面） 的链接，会发现它们都有三个共同点：

* 它们都在 id 是 bodyContent 的 div 标签里
* URL 链接不包含冒号
* URL 链接都以 /wiki/ 开头

我们可以利用这些规则稍微调整一下代码来获取词条链接：

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

#from urllib.error import HTTPError

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

**import** re

html = urlopen("http://en.wikipedia.org/wiki/Kevin\_Bacon")

bsObj = BeautifulSoup(html, 'html.parser')

**for** link **in** bsObj.find("div", {"id":"bodyContent"}).findAll("a", href=re.compile("^(/wiki/)((?!:).)\*$")):

**if** 'href' **in** link.attrs:

**print**(link.attrs['href'])

说明：

bsObj.find("div", {"id":"bodyContent"}) 提取标签<div id="bodyContent" class="mw-body-content">内容。

^(/wiki/)：以”/wiki/”开始的字符串。

(?!:)：不包含”:”。

((?!:).)\*:匹配不包含”:”的任意单个字符，0次或多次。

$：经常用在正则表达式的末尾

当然， 写程序来找出这个静态的维基百科词条里所有的词条链接很有趣， 不过没什么实际用处。 我们需要让这段程序更像下面的形式。

* 一个函数 getLinks ， 可以用维基百科词条 /wiki/< 词条名称 > 形式的 URL 链接作为参数， 然后以同样的形式返回一个列表， 里面包含所有的词条 URL 链接。
* 一个主函数， 以某个起始词条为参数调用 getLinks ， 再从返回的 URL 列表里随机选择一个词条链接， 再调用 getLinks ， 直到我们主动停止， 或者在新的页面上没有词条链接了， 程序才停止运行。

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

#from urllib.error import HTTPError

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

**import** datetime

**import** random

**import** re

random.seed(datetime.datetime.now())

**def** **getLinks**(articleUrl):

html = urlopen("http://en.wikipedia.org"+articleUrl)

bsObj = BeautifulSoup(html, 'html.parser')

**return** bsObj.find("div", {"id":"bodyContent"}).findAll("a", href=re.compile("^(/wiki/)((?!:).)\*$"))

links = getLinks("/wiki/Kevin\_Bacon")

**while** len(links) > 0:

newArticle = links[random.randint(0, len(links)-1)].attrs["href"]

**print**(newArticle)

links = getLinks(newArticle)

我们定义 getLinks 函数， 其参数是维基百科词条页面中 /wiki/< 词条名称 > 形式的 URL 链接， 前面加上维基百科的域名， http: //en. wikipedia. org ， 再用域名中的网页获得一个BeautifulSoup 对象。 之后用前面介绍过的参数抽取一列词条链接所在的标签 a 并返回它们。

程序的主函数首先把起始页面 https://en.wikipedia.org/wiki/Kevin\_Bacon里的词条链接列表（links 变量） 设置成链接列表。 然后用一个循环， 从页面中随机找一个词条链接标签并抽取 href 属性， 打印这个页面链接， 再把这个链接传入 getLinks 函数， 重新获取新的链接列表。

当然， 这里只是简单地构建一个从一个页面到另一个页面的爬虫， 要解决“维基百科六度分隔理论”问题还有一点儿工作得做。 我们还应该存储 URL 链接数据并分析数据。

### 采集整个网站

如果你需要系统地把整个网站按目 录分类， 或者要搜索网站上的每一个页面，那就得采集整个网站。那是一种非常耗费内存资源的过程， 尤其是处理大型网站时， 最合适的工具就是用一个数据库来储存采集的资源。

为了避免一个页面被采集两次， 需要对链接去重：

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

#from urllib.error import HTTPError

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

**import** re

pages = set()

**def** **getLinks**(pageUrl):

**global** pages

html = urlopen("http://en.wikipedia.org"+pageUrl)

bsObj = BeautifulSoup(html, 'html.parser')

**for** link **in** bsObj.findAll("a", href=re.compile("^(/wiki/)")):

**if** 'href' **in** link.attrs:

**if** link.attrs['href'] **not** **in** pages:

#new page

newPage = link.attrs['href']

**print**(newPage)

pages.add(newPage)

getLinks(newPage)

links = getLinks("")

不再限制爬虫采集的页面范围， 只要遇到页面就查找所有以 /wiki/ 开头的链接， 也不考虑链接是不是包含分号。 （提示： 词条链接不包含冒号， 而文档上传页面、 讨论页面之类的页面 URL 链接都包含冒号。 ）

警告：如果递归运行的次数非常多， 前面的递归程序就很可能崩溃。

Python默认的递归限制（程序递归地自我调用次数） 是 1000 次。 因为维基百科的网络链接浩如烟海， 所以这个程序达到递归限制后就会停止， 除非你设置一个较大的递归计数器， 或用其他手段不让它停止。

创建一个爬虫来收集页面标题、 正文的第一个段落， 以及编辑页面的链接（如果有的话） 这些信息。

第一步就是观察页面格式，拟定一个采集模式：

* 所有的标题（所有页面上， 不论是词条页面、 编辑历史页面还是其他页面） 都是在 h1 → span 标签里， 而且页面上只有一个 h1 标签。
* 所有的正文文字都在 div#bodyContent 标签里。 但是， 如果我们想更进一步获取第一段文字， 可能用 div#mw-content-text → p 更好（只选择第一段的标签） 。 这个规则对所有页面都适用， 除了文件页面（例如， https://en.wikipedia.org/wiki/File:Orbit\_of\_274301\_Wikipedia.svg ） ， 页面不包含内容文字（content text ） 的部分内容。
* 编辑链接只出现在词条页面上。 如果有编辑链接， 都位于 li#ca-edit 标签的 li#ca-edit → span → a 里面。

# -\*- coding: utf-8 -\*-

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

**import** re

pages = set()

**def** **getLinks**(pageUrl):

**global** pages

html = urlopen("http://en.wikipedia.org"+pageUrl)

bsObj = BeautifulSoup(html, 'html.parser')

**try**:

**print**(bsObj.h1.get\_text())

**print**(bsObj.find(id="mw-content-text").findAll("p")[0])

**print**(bsObj.find(id="ca-edit").find("span").find("a").attrs['href'])

**except** AttributeError:

**print**("页面缺少一些属性！ 不过不用担心！ ")

**for** link **in** bsObj.findAll("a", href=re.compile("^(/wiki/)")):

**if** 'href' **in** link.attrs:

**if** link.attrs['href'] **not** **in** pages:

#new page

newPage = link.attrs['href']

**print**("---------------------------------\n" + newPage)

pages.add(newPage)

getLinks(newPage)

links = getLinks("")

因为我们不可能确保每一页上都有所有类型的数据， 所以每个打印语句都是按照数据在页面上出现的可能性从高到低排列的。 也就是说， <h1> 标题标签会出现在每一页上（只要能识别， 无论哪一页都有） ， 所以我们首先试着获取它的数据。 正文内容会出现在大多数页面上（除了文件页面） ， 因此是第二个获取的数据。 “编辑”按钮只出现在标题和正文内容都已经获取的页面上， 但不是所有这类页面上都有， 所以我们最后打印这类数据。

### 通过互联网采集

之前是在一个网站内，顺着链接从一个页面跳到另一个页面，但是这一次，不再忽略外链， 而是跟着外链跳转。这就意味着我们必须在要寻找的信息以及查找方式上都极具灵活性。

# -\*- coding: utf-8 -\*-

**from** urllib.request **import** urlopen

**from** urllib.error **import** URLError, HTTPError

**from** urllib.parse **import** urlparse

**from** bs4 **import** BeautifulSoup

**import** re

**import** datetime

**import** random

pages = set()

random.seed(datetime.datetime.now())

#获取页面所有内链的列表

**def** **getInternalLinks**(bsObj, includeUrl):

includeUrl = urlparse(includeUrl).scheme + "://" + urlparse(includeUrl).netloc

internalLinks = []

#找出所有以'/'开头的链接

**for** link **in** bsObj.findAll("a", href=re.compile("^(/|.\*"+includeUrl+")")):

**if** link.attrs['href'] **is** **not** **None**:

**if** link.attrs['href'] **not** **in** internalLinks:

internalLinks.append(link.attrs['href'])

**else**:

internalLinks.append(link.attrs['href'])

**return** internalLinks

**def** **splitAddress**(address):

addressParts = address.replace('http://','').split('/')

**return**(addressParts)

#获取页面所有外链的列表

**def** **getExternalLinks**(bsObj, excludeUrl):

externalLinks = []

#找出所有以"http"或"www"开头且不包含当前URL的链接

**for** link **in** bsObj.findAll("a", href=re.compile("^(http|www)((?!" + excludeUrl + ").)\*$")):

**if** link.attrs['href'] **is** **not** **None**:

**if** link.attrs['href'] **not** **in** externalLinks:

externalLinks.append(link.attrs['href'])

**return** externalLinks

**def** **getRandomExternalLink**(startingPage):

**try**:

html = urlopen(startingPage)

**except** HTTPError **as** e:

**print**(e)

**return** **None**

**except** URLError **as** e:

**print**(e)

**return** **None**

**else**:

bsObj = BeautifulSoup(html, 'html.parser', from\_encoding="utf8")

externalLinks = getExternalLinks(bsObj, splitAddress(startingPage)[0])

**if** len(externalLinks) == 0:

**print**("No external links, looking around the site for one")

domain = urlparse(startingPage).scheme+"://"+urlparse(startingPage).netloc

internalLinks = getInternalLinks(bsObj, domain)

#internalLinks = getInternalLinks(startingPage)

**return** getRandomExternalLink(internalLinks[random.randint(0, len(internalLinks)-1)])

**else**:

**return** externalLinks[random.randint(0, len(externalLinks)-1)]

**def** **followExternalOnly**(startingSite):

externalLink = getRandomExternalLink(startingSite)

**if** externalLink **is** **not** **None**:

**print**("Random external link is: "+externalLink)

followExternalOnly(externalLink)

**else**:

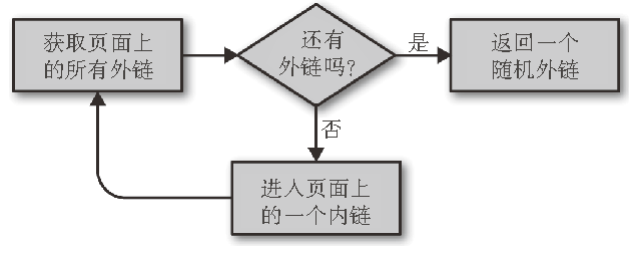
followExternalOnly(startingSite)

#followExternalOnly("http://www.sina.com.cn")

followExternalOnly("http://oreilly.com")

上面这个程序从 http://oreilly.com开始， 然后随机地从一个外链跳到另一个外链。

网站首页上并不能保证一直能发现外链。 这时为了能够发现外链， 就需要用一种类似前面案例中使用的采集方法， 即递归地深入一个网站直到找到一个外链才停止。



把任务分解成像“获取页面上所有外链”这样的小函数是不错的做法， 以后可以方便地修改代码以满足另一个采集任务的需求。 例如， 如果我们的目 标是采集一个网站所有的外链， 并且记录每一个外链，我们可以增加下面的函数：

#收集网站上发现的所有外链列表

allExtLinks = set()

allIntLinks = set()

**def** **getAllExternalLinks**(siteUrl):

html = urlopen(siteUrl)

bsObj = BeautifulSoup(html, 'html.parser')

internalLinks = getInternalLinks(bsObj, splitAddress(siteUrl)[0])

**print**(internalLinks)

externalLinks = getExternalLinks(bsObj, splitAddress(siteUrl)[0])

**for** link **in** externalLinks:

**if** link **not** **in** allExtLinks:

allExtLinks.add(link)

**print**(link)

**for** link **in** internalLinks:

**if** link **not** **in** allIntLinks:

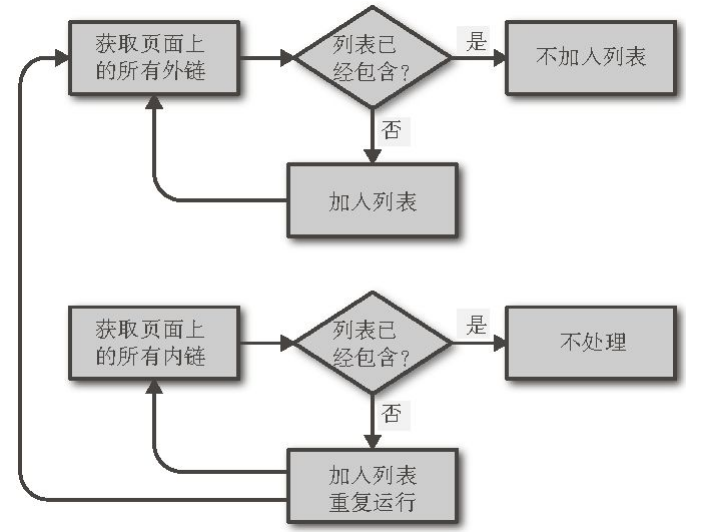
**print**("即将获取链接的URL是： "+link)

allIntLinks.add(link)

getAllExternalLinks(link)

getAllExternalLinks("http://oreilly.com")

这段代码可以看出两个循环——一个是收集内链， 一个是收集外链——然后彼此连接起来工作， 程序的流程如图 3-2 所示。



### 用 Scrapy采集

Scrapy就是一个帮你大幅度降低网页链接查找和识别工作复杂度的 Python库， 它可以让你轻松地采集一个或多个域名的信息。

创建scrapy工程：

$scrapy startproject wikiSpider

wikiSpider\items.py

# -\*- coding: utf-8 -\*-

# Define here the models for your scraped items

#

# See documentation in:

# http://doc.scrapy.org/en/latest/topics/items.html

**from** scrapy **import** Item, Field

**class** **Article**(Item):

# define the fields for your item here like:

# name = scrapy.Field()

title = Field()

wikiSpider\spiders\articleSpider.py

# -\*- coding: utf-8 -\*-

**from** scrapy.selector **import** Selector

**from** scrapy **import** Spider

**from** wikiSpider.items **import** Article

**from** scrapy.contrib.spiders **import** CrawlSpider, Rule

**from** scrapy.contrib.linkextractors.sgml **import** SgmlLinkExtractor

**class** **ArticleSpider**(Spider):

name = "article"

allowed\_domains = ["en.wikipedia.org"]

start\_urls = ["http://en.wikipedia.org/wiki/Main\_Page", "http://en.wikipedia.org/wiki/Python\_%28programming\_language%29"]

rules = [Rule(SgmlLinkExtractor(allow=('(/wiki/) ((?!:).)\*$'),), callback="parse\_item", follow=**True**)]

**def** **parse\_item**(self, response):

item = Article()

title = response.xpath('//h1/text() ')[0].extract()

**print**("Title is: "+title)

item['title'] = title

**return** item

运行：

$ scrapy crawl article

## 使用API

API 之所以叫 API 而不是叫网站的原因， 其实是首先 API 请求使用非常严谨的语法， 其次 API用 JSON 或 XML 格式表示数据， 而不是 HTML 格式。

### API通用规则

利用 HTTP 从网络服务获取信息有四种方式：

* GET

GET 就是你在浏览器中输入网址浏览网站所做的事情。 当你访问 http://freegeoip.net/json/50.78.253.58 时， 就会使用 GET 方法。 可以想象成 GET 在说： “喂，网络服务器，请按照这个网址给我信息。 ”

* POST

POST 基本就是当你填写表单或提交信息到网络服务器的后端程序时所做的事情。 每次当你登录网站的时候， 就是通过用户名和（有可能加密的） 密码发起一个 POST 请求。 如果你用 API 发起一个 POST请求， 相当于说“请把信息保存到你的数据库里”。

* PUT

PUT 在网站交互过程中不常用， 但是在 API 里面有时会用到。 PUT 请求用来更新一个对象或信息。 例如， API 可能会要求用 POST 请求创建新用户， 但是如果你要更新老用户的邮箱地址， 就要用 PUT 请求了。

* DELETE

DELETE 用于删除一个对象。 例如， 如果我们向 http://myapi.com/user/23 发出一个 DELETE 请求， 就会删除 ID 号是 23 的用户。 DELETE 方法在公共 API 里面不常用， 它们主要用于创建信息， 不能随便让一个用户去删掉数据库的信息。 但是， 和 PUT 方法一样， DELETE 方法也值得了解一下。

虽然有些 API 不需要验证操作（就是说任何人都可以使用 API，不需要注册），但是很多新式 API 在使用之前都要求客户验证。

有些 API 要求客户验证是为了计算 API 调用的费用， 或者是提供了包月 的服务。 有些验证是为了“限制”用户使用 API（限制每秒钟、 每小时或每天 API 调用的次数） ， 或者是限制一部分用户对某种信息或某类 API 的访问。 还有一些 API 可能不要求验证， 但是可能会为了市场营销而跟踪用户的使用行为。令牌除了在 URL 链接中传递， 还会通过请求头里的 cookie 把用户信息传递给服务器。

### 服务器响应

API 有一个重要的特征是它们会反馈格式友好的数据。 大多数反馈的数据格式都是 XML 和 JSON。JSON 比 XML 更受欢迎， 主要有两个原因。 首先， JSON 文件比完整的 XML 格式小。另一个原因是网络技术的改变。

不同 API 的调用语法大不相同， 但是有几条共同准则。 当使用 GET 请求获取数据时， 用 URL 路径描述你要获取的数据范围， 查询参数可以作为过滤器或附加请求使用。

例如， 下面这个虚拟的 API， 可以获取 ID 是 1234 的用户在 2014 年 8 月 份发表的所有博文：

http://socialmediasite.com/users/1234/posts?from=08012014&to=08312014

## 存储数据

### 媒体文件

下面的程序会把 http://pythonscraping.com主页上所有 src 属性的文件都下载下来：

# -\*- coding: utf-8 -\*-

**import** os

**from** urllib.request **import** urlretrieve, urlopen

**from** bs4 **import** BeautifulSoup

downloadDirectory = "downloaded"

baseUrl = "http://pythonscraping.com"

#--------------------------------------------------------

**def** **getAbsoluteURL**(baseUrl, source):

**if** source.startswith("http://www."):

url = "http://"+source[11:]

**elif** source.startswith("http://"):

url = source

**elif** source.startswith("www."):

url = "http://"+source[4:]

**else**:

url = baseUrl+"/"+source

**if** baseUrl **not** **in** url:

**return** **None**

**return** url

#--------------------------------------------------------

**def** **getDownloadPath**(baseUrl, absoluteUrl, downloadDirectory):

path = absoluteUrl.replace("www.", "")

path = path.replace(baseUrl, "")

path = downloadDirectory+path

directory = os.path.dirname(path)

**if** **not** os.path.exists(directory):

os.makedirs(directory)

**return** path

#--------------------------------------------------------

html = urlopen("http://www.pythonscraping.com")

bsObj = BeautifulSoup(html, 'html.parser')

downloadList = bsObj.findAll(src=**True**)

**for** download **in** downloadList:

fileUrl = getAbsoluteURL(baseUrl, download["src"])

**if** fileUrl **is** **not** **None**:

**print**(fileUrl)

urlretrieve(fileUrl, getDownloadPath(baseUrl, fileUrl, downloadDirectory))

#--------------------------------------------------------

这个程序只是为了演示； 请不要随意运行它， 因为这里没有对所有下载文件的类型进行检查， 也不应该用管理员权限运行它。 记得经常备份重要的文件， 不要在硬盘上存储敏感信息， 小心驶得万年船。

### 媒体文件

### 把数据存储到CSV

Python的 csv库可以非常简单地修改 CSV 文件， 甚至从零开始创建一个 CSV 文件：

**import** csv

csvFile = open("../files/test.csv", 'w+')

**try**:

writer = csv.writer(csvFile)

writer.writerow(('number' , 'number plus 2' , 'number times 2'))

**for** i **in** range(10):

writer.writerow((i, i+2, i\*2) )

**finally**:

csvFile.close()

网络数据采集的一个常用功能就是获取 HTML 表格并写入 CSV 文件。维基百科的文本编辑器对比词条（ https://en.wikipedia.org/wiki/Comparison\_of\_text\_editors ） 中用了许多复杂的 HTML 表格， 用到了颜色、 链接、 排序， 以及其他在写入 CSV 文件之前需要忽略的 HTML 元素。 用 BeautifulSoup 和 get\_text( ) 函数， 你可以用十几行代码完成这件事：

**import** csv

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

html = urlopen("http://en.wikipedia.org/wiki/Comparison\_of\_text\_editors")

bsObj = BeautifulSoup(html)

# 主对比表格是当前页面上的第一个表格

table = bsObj.findAll("table", {"class": "wikitable"})[0]

rows = table.findAll("tr")

csvFile = open("../files/editors.csv", 'wt', newline=' ', encoding='utf-8')

writer = csv.writer(csvFile)

**try**:

**for** row **in** rows:

csvRow = []

**for** cell **in** row.findAll(['td' , 'th']):

csvRow.append(cell.get\_text())

writer.writerow(csvRow)

**finally**:

csvFile.close()

### MySQL

Ubuntu安装：

**sudo** apt-get install mysql-server

**sudo** apt-get isntall mysql-client

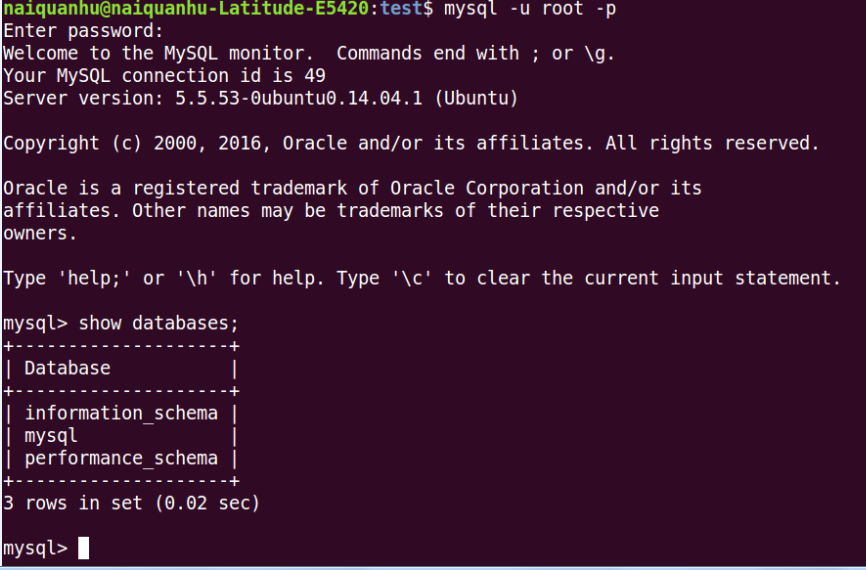
**sudo** apt-get install libmysqlclient-dev

登陆mysql数据库：

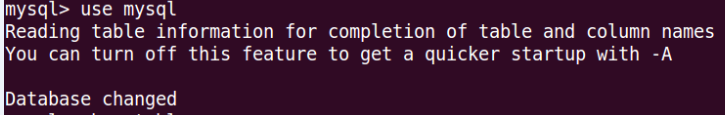
mysql -u root -p

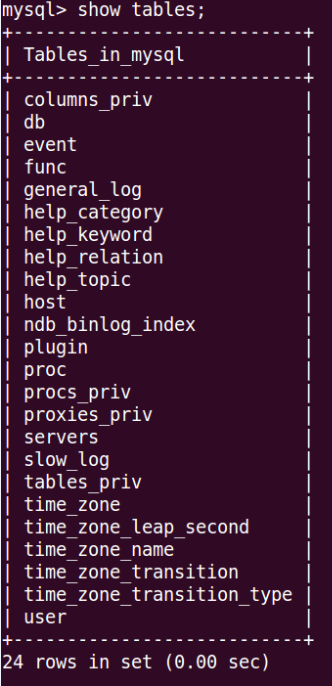
u 表示选择登陆的用户名， -p 表示登陆的用户密码

然后通过 show databases; 就可以查看当前的数据库。



我们选择 mysql数据库就行下一步操作，使用use mysql 命令，显示当前数据库的表单：show tables





创建数据库scraping：

>**CREATE** DATABASE scraping;

在数据库里创建一个表来存储采集的网页：

>**USE** scraping;

>**CREATE** **TABLE** pages (id **BIGINT**(7) **NOT** **NULL** AUTO\_INCREMENT, title **VARCHAR**(200) ,

content **VARCHAR**(10000) , created **TIMESTAMP** **DEFAULT** **CURRENT\_TIMESTAMP**, **PRIMARY** **KEY**

(id) ) ;

语句执行之后， 你可以用 DESCRIBE 查看数据表的结构：

mysql> **DESCRIBE** pages;

+---------+----------------+------+-----+-------------------+----------------+

| Field | **Type** | **Null** | **Key** | **Default** | Extra |

+---------+----------------+------+-----+-------------------+----------------+

| id | **bigint**(7) | **NO** | PRI | **NULL** | auto\_increment |

| title | **varchar**(200) | YES | | **NULL** | |

| content | **varchar**(10000) | YES | | **NULL** | |

| created | **timestamp** | **NO** | | **CURRENT\_TIMESTAMP** | |

+---------+----------------+------+-----+-------------------+----------------+

4 **rows** **in** **set** (0.00 sec)

可以在 pages 表里插入一些测试数据， 如下所示：

> **INSERT** **INTO** pages (title, content) **VALUES** ("Test page title", "This is some test page content. It can be up to 10, 000 characters long. ") ;

#### 与Python整合

使用PyMySQL:

**sudo** pip install PyMySQL

查询：

# -\*- coding: utf-8 -\*-

**import** pymysql

conn = pymysql.connect(host='127.0.0.1',

#unix\_socket='/tmp/mysql.sock',

port=3306,

user='root',

passwd='robin',

db='scraping',

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor)

cur = conn.cursor()

#cur.execute("USE scraping")

cur.execute("SELECT \* FROM pages")

**print**(cur.fetchone())

cur.close()

conn.close()

让数据库支持 Unicode：

mysql> **ALTER** DATABASE scraping **CHARACTER** **SET** = utf8mb4 **COLLATE** = utf8mb4\_unicode\_ci;

mysql> **ALTER** **TABLE** pages **CONVERT** **TO** **CHARACTER** **SET** utf8mb4 **COLLATE** utf8mb4\_unicode\_ci;

mysql> **ALTER** **TABLE** pages CHANGE title title **VARCHAR**(200) **CHARACTER** **SET** utf8mb4 **COLLATE** utf8mb4\_unicode\_ci;

mysql> **ALTER** **TABLE** pages CHANGE content content **VARCHAR**(10000) **CHARACTER** **SET** utf8mb4 **COLLATE** utf8mb4\_unicode\_ci;

这四行语句改变的内容有： 数据库、 数据表， 以及两个字段的默认编码都从 utf8mb4 （严格说来也属于 Unicode， 但是对大多数 Unicode 字符的支持都非常不好） 转变成了 utf8mb4 \_unicode\_ci 。

现在数据库已经准备好接收维基百科的各种信息了， 你可以用下面的程序来存储数据：

# -\*- coding: utf-8 -\*-

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

**import** re

**import** datetime

**import** random

**import** pymysql

conn = pymysql.connect(host='127.0.0.1',

#unix\_socket='/tmp/mysql.sock',

port=3306,

user='root',

passwd='robin',

db='scraping',

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor)

cur = conn.cursor()

#cur.execute("USE scraping")

random.seed(datetime.datetime.now())

**def** **store**(title, content):

cur.execute("INSERT INTO pages (title, content) VALUES (\"%s\",\"%s\")", (title, content) )

cur.connection.commit()

**def** **getLinks**(articleUrl):

html = urlopen("http://en.wikipedia.org"+articleUrl)

bsObj = BeautifulSoup(html, 'html.parser')

title = bsObj.find("h1").get\_text()

content = bsObj.find("div", {"id": "mw-content-text"}).find("p").get\_text()

store(title, content)

**return** bsObj. find("div", {"id": "bodyContent"}).findAll("a", href=re.compile("^(/wiki/) ((?!:).)\*$"))

links = getLinks("/wiki/Kevin\_Bacon")

**try**:

**while** len(links) > 0:

newArticle = links[random.randint(0, len(links)-1)].attrs["href"]

**print**(newArticle)

links = getLinks(newArticle)

**finally**:

cur.close()

conn.close()

为了使数据库更高效，要注意一下几点：

1. 给每个数据表都增加一个 id 字段。
2. 用智能索引 。
3. 最后一点是关于数据查询时间和数据库空间的问题。

### Email

略

## 读取文档

略

## 数据清洗

本章将介绍一些工具和技术， 通过改变代码的编写方式， 帮你从源头控制数据零乱的问题， 并且对已经进入数据库的数据进行清洗。清洗内容见代码中的注释：

# -\*- coding: utf-8 -\*-

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

**import** re

**import** string

**def** **cleanInput**(input):#数据清洗

input = re.sub('\n+' , " ", input) #把内容中的换行符（或者多个换行符） 替换成空格

input = re.sub('\[[0-9]\*\]' , "", input) #剔除维基百科的引 用标记（方括号包裹的数字， 如 [1]）

input = re.sub(' +' , " ", input) #把连续的多个空格替换成一个空格

input = bytes(input, "UTF-8") #把内容转换成 UTF-8 格式以消除转义字符

input = input.decode("ascii", "ignore")

cleanInput = []

input = input.split(' ')

**for** item **in** input: #

item = item.strip(string.punctuation) #剔除标点符号

**if** len(item) > 1 **or** (item.lower() == 'a' **or** item.lower() == 'i' ): #剔除单字符的“单词”， 除非这个字符是“i”或“a”；

cleanInput.append(item)

**return** cleanInput

**def** **ngrams**(input, n):

input = cleanInput(input)

output = []

**for** i **in** range(len(input)-n+1):

output.append(input[i:i+n])

**return** output

html = urlopen("http://en.wikipedia.org/wiki/Python\_(programming\_language)")

bsObj = BeautifulSoup(html, 'html.parser')

content = bsObj.find("div", {"id": "mw-content-text"}).get\_text()

ngrams = ngrams(content, 2)

**print**(ngrams)

**print**("2-grams count is: "+str(len(ngrams)))

数据清洗可以自己写代码来做，也可以使用第三方的工具。

**OpenRefine：**是 Metaweb 公司 2009 年发布的一个开源软件。

如何安装与使用请自行搜索。

## 自然语言处理

我们用 Python的 operator 模块对 2-gram序列的频率字典进行排序：

# -\*- coding: utf-8 -\*-

**from** urllib.request **import** urlopen

**from** bs4 **import** BeautifulSoup

**import** re

**import** string

**import** operator

**def** **cleanInput**(input):#数据清洗

input = re.sub('\n+' , " ", input) #把内容中的换行符（或者多个换行符） 替换成空格

input = re.sub('\[[0-9]\*\]' , "", input) #剔除维基百科的引 用标记（方括号包裹的数字， 如 [1]）

input = re.sub(' +' , " ", input) #把连续的多个空格替换成一个空格

input = bytes(input, "UTF-8") #把内容转换成 UTF-8 格式以消除转义字符

input = input.decode("ascii", "ignore")

cleanInput = []

input = input.split(' ')

**for** item **in** input: #

item = item.strip(string.punctuation) #剔除标点符号

**if** len(item) > 1 **or** (item.lower() == 'a' **or** item.lower() == 'i' ): #剔除单字符的“单词”， 除非这个字符是“i”或“a”；

cleanInput.append(item)

**return** cleanInput

**def** **ngrams**(input, n):

input = cleanInput(input)

output = {}

**for** i **in** range(len(input)-n+1):

ngramTemp = " ".join(input[i: i+n])

**if** ngramTemp **not** **in** output:

output[ngramTemp] = 0

output[ngramTemp] += 1

**return** output

content = str(

urlopen("http://pythonscraping.com/files/inaugurationSpeech.txt").read(),

'utf-8')

ngrams = ngrams(content, 2)

sortedNGrams = sorted(ngrams.items(), key=operator.itemgetter(1), reverse=**True**)

**print**(sortedNGrams)

## 穿越网页表单与登录窗口进行采集

”今天， 网络正在朝着页面交互、 社交媒体、 用户产生内容的趋势不断地演进。 表单和登录窗口是许多网站中不可或缺的组成部分。 不过， 它们还是比较容易处理的。到目 前为止， 我们示例中的网络爬虫在和大多数网站的服务器进行数据交互时， 都是用 HTTP 协议的 GET 方法去请求信息。 这一章， 我们将重点介绍 POST 方法， 即把信息推送给网络服务器进行存储和分析。

### Python Requests库

Requests 库（ http://www.python-requests.org/ ） 就是这样一个擅长处理那些复杂的 HTTP 请求、 cookie、 header（响应头和请求头） 等内容的 Python第三方库。Python的标准库 urllib2 为你提供了大多数 HTTP 功能， 但是它的 API 非常差劲。和任何 Python第三方库一样， Requests 库也可以用其他第三方 Python库管理器安装， 比如 pip， 或者直接下载源代码（ https://github.com/kennethreitz/requests/tarball/master ） 安装

# 项目练习

## 代码管理——Git&Github使用

### Git安装及使用

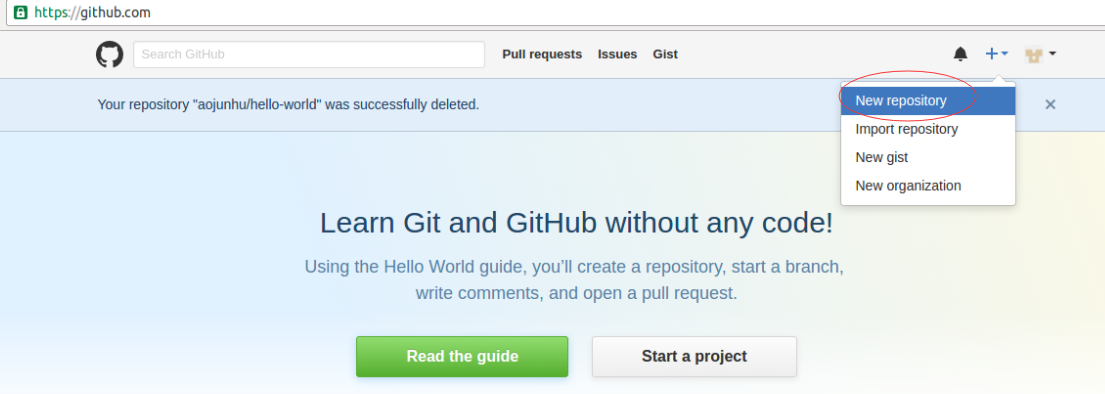
略

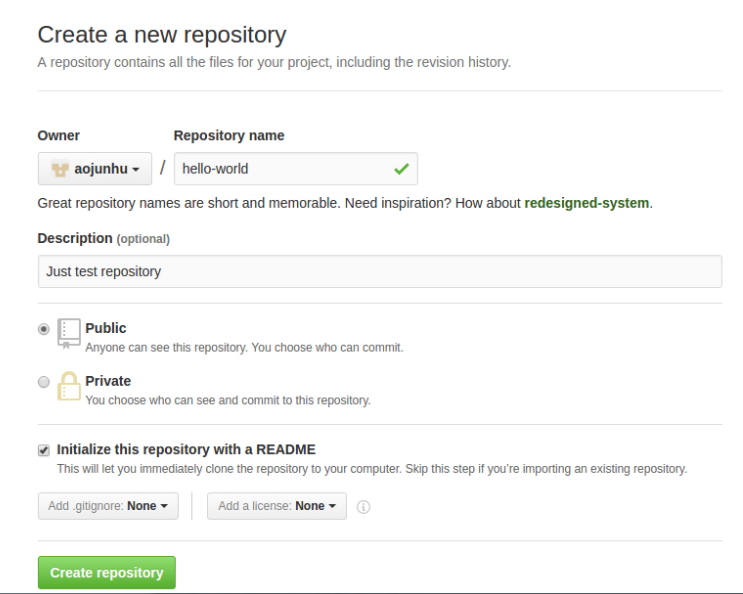
### GitHub使用

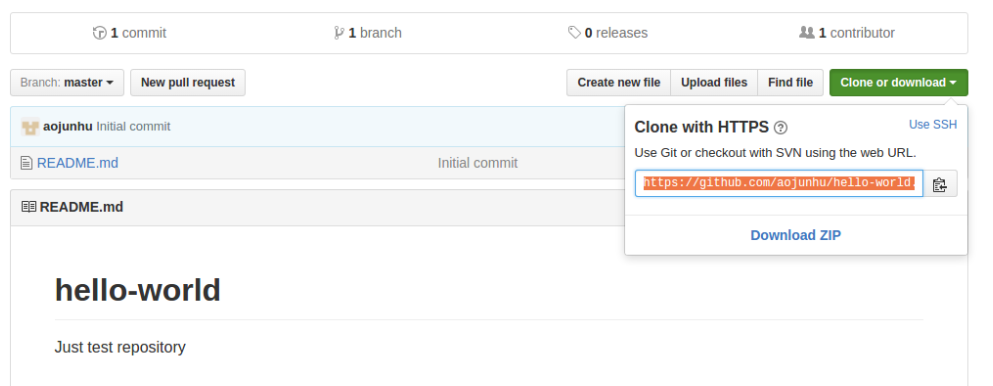
1.注册GitHub账户并登录

<https://github.com>

2.创建仓库



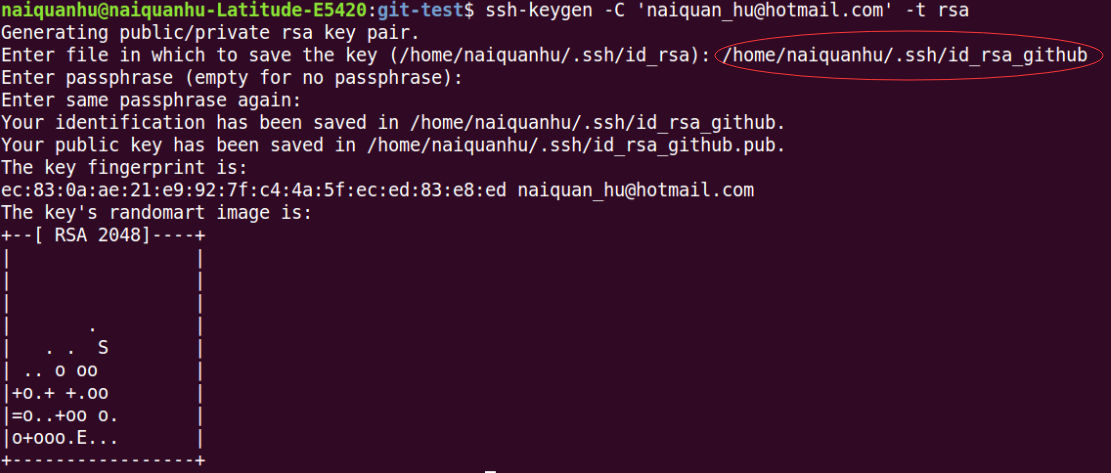




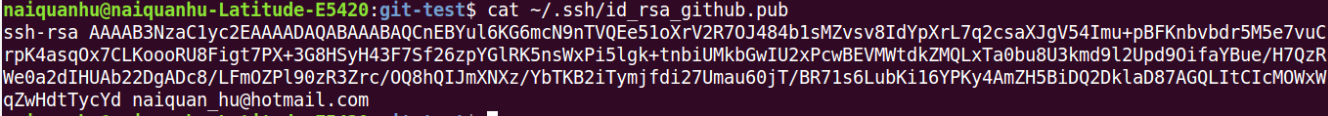
3.check out代码库（Ubuntu为例）

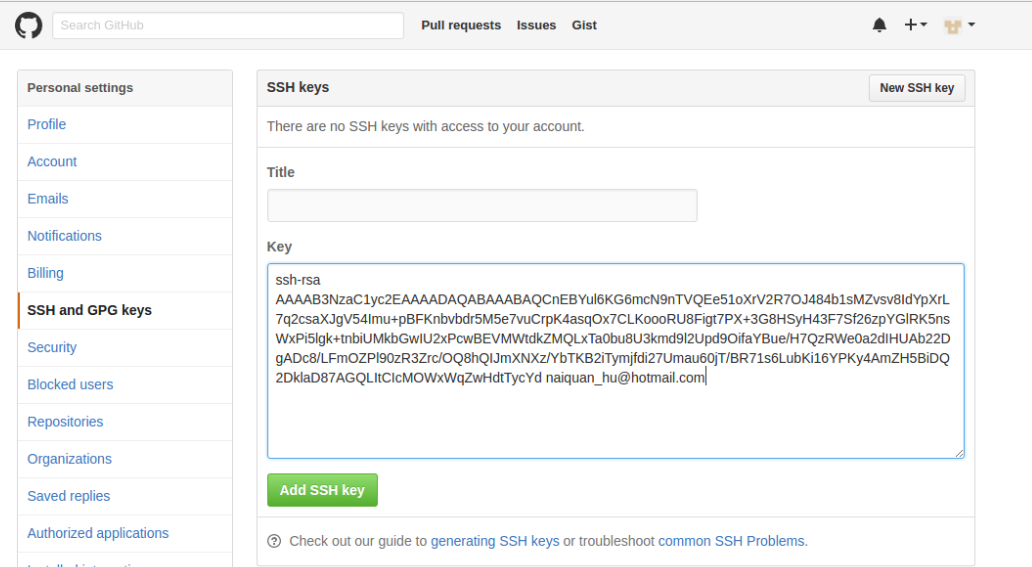
首先，生成SSH public key，并加入GitHub：

$ ssh-keygen -C 'your@email.address' -t



然后，把SSH public key加入GitHub：





之后，check out 代码：

ssh-add ~/.ssh/id\_rsa\_github

**git** clone https://github.com/aojunhu/hello-world.git git-test

之后对代码修改后可以用git add; git commi; git push提交，在git push时要输入username和password

## 图片转字符画

ascii.py

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

**from** PIL **import** Image

**import** argparse

#命令行输入参数处理

parser **=** argparse**.**ArgumentParser**()**

parser**.**add\_argument**(**'file'**)** #输入文件

parser**.**add\_argument**(**'-o'**,** '--output'**)** #输出文件

parser**.**add\_argument**(**'--width'**,** type **=** int**,** default **=** 80**)** #输出字符画宽

parser**.**add\_argument**(**'--height'**,** type **=** int**,** default **=** 80**)** #输出字符画高

#获取参数

args **=** parser**.**parse\_args**()**

IMG **=** args**.**file #输入文件

WIDTH **=** args**.**width #输出字符画宽

HEIGHT **=** args**.**height #输出字符画高

OUTPUT **=** args**.**output #输出文件

ascii\_char **=** list**(**"$@B%8&WM#\*oahkbdpqwmZO0QLCJUYXzcvunxrjft/\|()1{}[]?-\_+~<>i!lI;:,\"^`'. "**)**

# 将256灰度映射到70个字符上

**def** get\_char**(**r**,** g**,** b**,** alpha **=** 256**):**

**if** alpha **==** 0**:**

**return** ' '

length **=** len**(**ascii\_char**)**

gray **=** int**(**0.2126 **\*** r **+** 0.7152 **\*** g **+** 0.0722 **\*** b**)**

unit **=** **(**256.0 **+** 1**)/**length

**return** ascii\_char**[**int**(**gray**/**unit**)]**

'''

表示执行的是此代码所在的文件。 如果这个文件是作为模块被其他文件调用，不会执行这里面的代码。

只有执行这个文件时， if 里面的语句才会被执行。 这个功能经常可以用于进行测试。

'''

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:**

im **=** Image**.**open**(**IMG**)**

im **=** im**.**resize**((**WIDTH**,**HEIGHT**),** Image**.**NEAREST**)**

txt **=** ""

**for** i **in** range**(**HEIGHT**):**

**for** j **in** range**(**WIDTH**):**

txt **+=** get\_char**(\***im**.**getpixel**((**j**,**i**)))**#在list或tuple前面加一个\*号，把list或tuple的元素变成可变参数传进去，getpixel((j, i))返回值为(R, G, B, A)=(237, 28, 36, 255)

txt **+=** '\n'

**print(**txt**)**

#字符画输出到文件

**if** OUTPUT**:**

**with** open**(**OUTPUT**,**'w'**)** **as** f**:**

f**.**write**(**txt**)**

**else:**

**with** open**(**"output.txt"**,**'w'**)** **as** f**:**

f**.**write**(**txt**)**

## 200行Python代码实现2048

### 用户行为

所有的有效输入都可以转换为"上，下，左，右，游戏重置，退出"这六种行为，用 actions 表示

actions **=** **[**'Up'**,** 'Left'**,** 'Down'**,** 'Right'**,** 'Restart'**,** 'Exit'**]**

有效输入键是最常见的 W（上），A（左），S（下），D（右），R（重置），Q（退出），这里要考虑到大写键开启的情况，获得有效键值列表：

letter\_codes **=** **[**ord**(**ch**)** **for** ch **in** 'WASDRQwasdrq'**]**

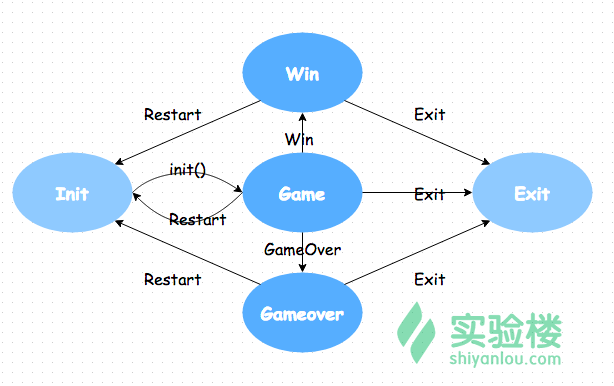
将输入与行为进行关联：

actions\_dict = dict(zip(letter\_codes, actions \* 2))

### 状态机

处理游戏主逻辑的时候我们会用到一种十分常用的技术：状态机，或者更准确的说是有限状态机（FSM）

你会发现 2048 游戏很容易就能分解成几种状态的转换。



状态机会不断循环，直到达到 Exit 终结状态结束程序。

### 用户输入处理

阻塞＋循环，直到获得用户有效输入才返回对应行为：

**def** **get\_user\_action**(keyboard):

char = "N"

**while** char **not** **in** actions\_dict:

char = keyboard.getch()

**return** actions\_dict[char]

### 矩阵转置与矩阵逆转

加入这两个操作可以大大节省我们的代码量，减少重复劳动，看到后面就知道了。

矩阵转置：将矩阵的行变为列，将列变为行。

zip函数接受任意多个（包括0个和1个）序列作为参数，返回一个tuple列表。实现的其实是矩阵转置功能。

**def** **transpose**(field):

**return** [list(row) **for** row **in** zip(\*field)]

矩阵逆转（不是逆矩阵）：

比如A=[[1, 2], [3, 4] -🡪 B=[[2, 1], [4, 3]]

**def** **invert**(field):

**return** [row[::-1] **for** row **in** field]

### 创建棋盘

初始化棋盘的参数，可以指定棋盘的高和宽以及游戏胜利条件，默认是最经典的 4x4～2048。

**class** **GameField**(object):

**def** **\_\_init\_\_**(self, height=4, width=4, win=2048):

self.height = height #高

self.width = width #宽

self.win\_value = 2048 #过关分数

self.score = 0 #当前分数

self.highscore = 0 #最高分

self.reset() #棋盘重置

### 棋盘操作

随机生成一个 2 或者 4

**def** **spawn**(self):

new\_element = 4 **if** randrange(100) > 89 **else** 2

(i,j) = choice([(i,j) **for** i **in** range(self.width) **for** j **in** range(self.height) **if** self.field[i][j] == 0])

self.field[i][j] = new\_element

### 重置棋盘

**def** **reset**(self):

**if** self.score > self.highscore:

self.highscore = self.score

self.score = 0

self.field = [[0 **for** i **in** range(self.width)] **for** j **in** range(self.height)]

self.spawn()

self.spawn()

### 一行向左合并

(注：这一操作是在 move 内定义的，拆出来是为了方便阅读)

**def** **move\_row\_left**(row):

**def** **tighten**(row): # 把零散的非零单元挤到一块

new\_row = [i **for** i **in** row **if** i != 0]

new\_row += [0 **for** i **in** range(len(row) - len(new\_row))]

**return** new\_row

**def** **merge**(row): # 对邻近元素进行合并

pair = **False**

new\_row = []

**for** i **in** range(len(row)):

**if** pair:

new\_row.append(2 \* row[i])

self.score += 2 \* row[i]

pair = **False**

**else**:

**if** i + 1 < len(row) **and** row[i] == row[i + 1]:

pair = **True**

new\_row.append(0)

**else**:

new\_row.append(row[i])

**assert** len(new\_row) == len(row)

**return** new\_row

#先挤到一块再合并再挤到一块

**return** tighten(merge(tighten(row)))

### 棋盘走一步

通过对矩阵进行转置与逆转，可以直接从左移得到其余三个方向的移动操作

**def** **move**(self, direction):

**def** **move\_row\_left**(row):

#一行向左合并

moves = {}

moves['Left'] = **lambda** field: [move\_row\_left(row) **for** row **in** field]

moves['Right'] = **lambda** field: invert(moves['Left'](invert(field)))

moves['Up'] = **lambda** field: transpose(moves['Left'](transpose(field)))

moves['Down'] = **lambda** field: transpose(moves['Right'](transpose(field)))

**if** direction **in** moves:

**if** self.move\_is\_possible(direction):

self.field = moves[direction](self.field)

self.spawn()

**return** **True**

**else**:

**return** **False**

### 判断输赢

**def** **is\_win**(self):

**return** any(any(i >= self.win\_value **for** i **in** row) **for** row **in** self.field)

**def** **is\_gameover**(self):

**return** **not** any(self.move\_is\_possible(move) **for** move **in** actions)

### 判断能否移动

**def** **move\_is\_possible**(self, direction):

**def** **row\_is\_left\_movable**(row):

**def** **change**(i):

**if** row[i] == 0 **and** row[i + 1] != 0: # 可以移动

**return** **True**

**if** row[i] != 0 **and** row[i + 1] == row[i]: # 可以合并

**return** **True**

**return** **False**

**return** any(change(i) **for** i **in** range(len(row) - 1))

check = {}

check['Left'] = **lambda** field: any(row\_is\_left\_movable(row) **for** row **in** field)

check['Right'] = **lambda** field: check['Left'](invert(field))

check['Up'] = **lambda** field: check['Left'](transpose(field))

check['Down'] = **lambda** field: check['Right'](transpose(field))

**if** direction **in** check:

**return** check[direction](self.field)

**else**:

**return** **False**

### 绘制游戏界面

**def** **draw**(self, screen):

help\_string1 = '(W)Up (S)Down (A)Left (D)Right'

help\_string2 = ' (R)Restart (Q)Exit'

gameover\_string = ' GAME OVER'

win\_string = ' YOU WIN!'

**def** **cast**(string):

screen.addstr(string + '\n')

#绘制水平分割线

**def** **draw\_hor\_separator**():

line = '+' + ('+------' \* self.width + '+')[1:]

separator = defaultdict(**lambda**: line)

**if** **not** hasattr(draw\_hor\_separator, "counter"):

draw\_hor\_separator.counter = 0

cast(separator[draw\_hor\_separator.counter])

draw\_hor\_separator.counter += 1

**def** **draw\_row**(row):

cast(''.join('|{: ^5} '.format(num) **if** num > 0 **else** '| ' **for** num **in** row) + '|')

screen.clear()

cast('SCORE: ' + str(self.score))

**if** 0 != self.highscore:

cast('HGHSCORE: ' + str(self.highscore))

**for** row **in** self.field:

draw\_hor\_separator()

draw\_row(row)

draw\_hor\_separator()

**if** self.is\_win():

cast(win\_string)

**else**:

**if** self.is\_gameover():

cast(gameover\_string)

**else**:

cast(help\_string1)

cast(help\_string2)

### 全部代码

#!/usr/bin/env python3

#-\*- coding:utf-8 -\*-

**import** curses

**from** random **import** randrange, choice # generate and place new tile

**from** collections **import** defaultdict

letter\_codes = [ord(ch) **for** ch **in** 'WASDRQwasdrq']

actions = ['Up', 'Left', 'Down', 'Right', 'Restart', 'Exit']

actions\_dict = dict(zip(letter\_codes, actions \* 2))

**def** **get\_user\_action**(keyboard):

char = "N"

**while** char **not** **in** actions\_dict:

char = keyboard.getch()

**return** actions\_dict[char]

#矩阵转置

**def** **transpose**(field):

**return** [list(row) **for** row **in** zip(\*field)]

#矩阵逆转，前后列互换

**def** **invert**(field):

**return** [row[::-1] **for** row **in** field]

**class** **GameField**(object):

**def** **\_\_init\_\_**(self, height=4, width=4, win=2048):

self.height = height

self.width = width

self.win\_value = win

self.score = 0

self.highscore = 0

self.reset()

**def** **reset**(self):

**if** self.score > self.highscore:

self.highscore = self.score

self.score = 0

self.field = [[0 **for** i **in** range(self.width)] **for** j **in** range(self.height)]

self.spawn()

self.spawn()

**def** **move**(self, direction):

**def** **move\_row\_left**(row):

**def** **tighten**(row): # squeese non-zero elements together

new\_row = [i **for** i **in** row **if** i != 0]

new\_row += [0 **for** i **in** range(len(row) - len(new\_row))]

**return** new\_row

**def** **merge**(row):

pair = **False**

new\_row = []

**for** i **in** range(len(row)):

**if** pair:

new\_row.append(2 \* row[i])

self.score += 2 \* row[i]

pair = **False**

**else**:

**if** i + 1 < len(row) **and** row[i] == row[i + 1]:

pair = **True**

new\_row.append(0)

**else**:

new\_row.append(row[i])

**assert** len(new\_row) == len(row)

**return** new\_row

**return** tighten(merge(tighten(row)))

moves = {}

moves['Left'] = **lambda** field: \

[move\_row\_left(row) **for** row **in** field]

moves['Right'] = **lambda** field: \

invert(moves['Left'](invert(field)))

moves['Up'] = **lambda** field: \

transpose(moves['Left'](transpose(field)))

moves['Down'] = **lambda** field: \

transpose(moves['Right'](transpose(field)))

**if** direction **in** moves:

**if** self.move\_is\_possible(direction):

self.field = moves[direction](self.field)

self.spawn()

**return** **True**

**else**:

**return** **False**

**def** **is\_win**(self):

**return** any(any(i >= self.win\_value **for** i **in** row) **for** row **in** self.field)

**def** **is\_gameover**(self):

**return** **not** any(self.move\_is\_possible(move) **for** move **in** actions)

**def** **draw**(self, screen):

help\_string1 = '(W)Up (S)Down (A)Left (D)Right'

help\_string2 = ' (R)Restart (Q)Exit'

gameover\_string = ' GAME OVER'

win\_string = ' YOU WIN!'

**def** **cast**(string):

screen.addstr(string + '\n')

**def** **draw\_hor\_separator**():

line = '+' + ('+------' \* self.width + '+')[1:]

separator = defaultdict(**lambda**: line)

**if** **not** hasattr(draw\_hor\_separator, "counter"):

draw\_hor\_separator.counter = 0

cast(separator[draw\_hor\_separator.counter])

draw\_hor\_separator.counter += 1

**def** **draw\_row**(row):

cast(''.join('|{: ^5} '.format(num) **if** num > 0 **else** '| ' **for** num **in** row) + '|')

screen.clear()

cast('SCORE: ' + str(self.score))

**if** 0 != self.highscore:

cast('HGHSCORE: ' + str(self.highscore))

**for** row **in** self.field:

draw\_hor\_separator()

draw\_row(row)

draw\_hor\_separator()

**if** self.is\_win():

cast(win\_string)

**else**:

**if** self.is\_gameover():

cast(gameover\_string)

**else**:

cast(help\_string1)

cast(help\_string2)

**def** **spawn**(self):

new\_element = 4 **if** randrange(100) > 89 **else** 2

(i,j) = choice([(i,j) **for** i **in** range(self.width) **for** j **in** range(self.height) **if** self.field[i][j] == 0])

self.field[i][j] = new\_element

**def** **move\_is\_possible**(self, direction):

**def** **row\_is\_left\_movable**(row):

**def** **change**(i): # true if there'll be change in i-th tile

**if** row[i] == 0 **and** row[i + 1] != 0: # Move

**return** **True**

**if** row[i] != 0 **and** row[i + 1] == row[i]: # Merge

**return** **True**

**return** **False**

**return** any(change(i) **for** i **in** range(len(row) - 1))

check = {}

check['Left'] = **lambda** field: \

any(row\_is\_left\_movable(row) **for** row **in** field)

check['Right'] = **lambda** field: \

check['Left'](invert(field))

check['Up'] = **lambda** field: \

check['Left'](transpose(field))

check['Down'] = **lambda** field: \

check['Right'](transpose(field))

**if** direction **in** check:

**return** check[direction](self.field)

**else**:

**return** **False**

**def** **main**(stdscr):

**def** **init**():

#重置游戏棋盘

game\_field.reset()

**return** 'Game'

**def** **not\_game**(state):

#画出 GameOver 或者 Win 的界面

game\_field.draw(stdscr)

#读取用户输入得到action，判断是重启游戏还是结束游戏

action = get\_user\_action(stdscr)

responses = defaultdict(**lambda**: state) #默认是当前状态，没有行为就会一直在当前界面循环

responses['Restart'], responses['Exit'] = 'Init', 'Exit' #对应不同的行为转换到不同的状态

**return** responses[action]

**def** **game**():

#画出当前棋盘状态

game\_field.draw(stdscr)

#读取用户输入得到action

action = get\_user\_action(stdscr)

**if** action == 'Restart':

**return** 'Init'

**if** action == 'Exit':

**return** 'Exit'

**if** game\_field.move(action): # move successful

**if** game\_field.is\_win():

**return** 'Win'

**if** game\_field.is\_gameover():

**return** 'Gameover'

**return** 'Game'

state\_actions = {

'Init': init,

'Win': **lambda**: not\_game('Win'),

'Gameover': **lambda**: not\_game('Gameover'),

'Game': game

}

curses.use\_default\_colors()

game\_field = GameField(win=32)

state = 'Init'

#状态机开始循环

**while** state != 'Exit':

state = state\_actions[state]()

curses.wrapper(main)

## pygame 开发打飞机游戏

注意：当把系统python版本设置为3.4.3时，安装pygame失败，所以该项目运行与python2.7.

使用 Python 及 pygame 快速开发小游戏的方式。需要安装 pygame 库来支持本实验所需的代码运行。

### 游戏角色

角色包括：玩家飞机、敌机及子弹。

用户可以通过键盘移动玩家飞机在屏幕上的位置来打击不同位置的敌机。因此设计以下 Player，Enemy 和 Bullet 三个类对应三种游戏角色。

对于 Player，需要的操作有射击和移动两种，移动又分为上下左右 4 种情况。

对于 Enemy，则比较简单，只需要移动即可，从屏幕上方出现并移动到屏幕下方。

对于 Bullet，与飞机相同，仅需要以一定速度移动即可。

### 游戏功能

相信玩过微信打飞机的朋友都熟悉，这里将游戏做了简化。飞机的速度固定，子弹的速度固定，基本操作是移动玩家飞机，目标飞机随机从屏幕上方出现并匀速落到下方，子弹从玩家飞机发出，碰到目标飞机会击毁，如果目标飞机碰到玩家飞机，则 Game Over 并显示分数。

### 代码实现

#### 界面显示

代码实现所需的 resources 图片文件都可以通过下述命令获取：

$git clone http://git.shiyanlou.com/shiyanlou/PythonShootGame

**详细步骤**

1. 初始化 pygame
2. 设置游戏界面大小、背景图片及标题
3. 游戏主循环内需要处理游戏界面的初始化、更新及退出
4. 显示玩家飞机（代码中使用的 resources/image/shoot.png 图里包含多种飞机，只需要使用 pygame.image 的 subsurface API 根据位置截取 shoot.png 中所需的图片）

**示例代码**

# 初始化PyGame

pygame.init()

# 设置游戏界面大小、背景图片及标题

# 游戏界面像素大小

screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))

# 游戏界面标题

pygame.display.set\_caption('飞机大战')

# 背景图

background = pygame.image.load('resources/image/background.png').convert()

# Game Over的背景图

game\_over = pygame.image.load('resources/image/gameover.png')

# 飞机及子弹图片集合

plane\_img = pygame.image.load('resources/image/shoot.png')

# 设置玩家飞机不同状态的图片列表，多张图片展示为动画效果

player\_rect = []

player\_rect.append(pygame.Rect(0, 99, 102, 126)) # 玩家飞机图片

player\_rect.append(pygame.Rect(165, 360, 102, 126))

player\_rect.append(pygame.Rect(165, 234, 102, 126)) # 玩家爆炸图片

player\_rect.append(pygame.Rect(330, 624, 102, 126))

player\_rect.append(pygame.Rect(330, 498, 102, 126))

player\_rect.append(pygame.Rect(432, 624, 102, 126))

player\_pos = [200, 600]

player = Player(plane\_img, player\_rect, player\_pos)

# 子弹图片

bullet\_rect = pygame.Rect(1004, 987, 9, 21)

bullet\_img = plane\_img.subsurface(bullet\_rect)

# 敌机不同状态的图片列表，多张图片展示为动画效果

enemy1\_rect = pygame.Rect(534, 612, 57, 43)

enemy1\_img = plane\_img.subsurface(enemy1\_rect)

enemy1\_down\_imgs = []

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(267, 347, 57, 43)))

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(873, 697, 57, 43)))

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(267, 296, 57, 43)))

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(930, 697, 57, 43)))

enemies1 = pygame.sprite.Group()

# 存储被击毁的飞机，用来渲染击毁动画

enemies\_down = pygame.sprite.Group()

# 初始化射击及敌机移动频率

shoot\_frequency = 0

enemy\_frequency = 0

# 玩家飞机被击中后的效果处理

player\_down\_index = 16

# 初始化分数

score = 0

# 游戏循环帧率设置

clock = pygame.time.Clock()

# 判断游戏循环退出的参数

running = **True**

#### 事件处理

1. 获取键盘事件（上下左右按键）
2. 处理键盘事件（移动飞机的位置）
3. 将上述步骤代码放入游戏主循环中

# 获取键盘事件（上下左右按键）

key\_pressed = pygame.key.get\_pressed()

# 处理键盘事件（移动飞机的位置）

**if** key\_pressed[K\_w] **or** key\_pressed[K\_UP]:

player.moveUp()

**if** key\_pressed[K\_s] **or** key\_pressed[K\_DOWN]:

player.moveDown()

**if** key\_pressed[K\_a] **or** key\_pressed[K\_LEFT]:

player.moveLeft()

**if** key\_pressed[K\_d] **or** key\_pressed[K\_RIGHT]:

player.moveRight()

#### 子弹处理

子弹由玩家飞机发出，并以一定速度向界面上方移动。

**详细步骤**

1. 生成子弹，需要控制发射频率
2. 以固定速度移动子弹
3. 移动出屏幕后删除子弹
4. 敌机被子弹击中效果处理（下一节处理）

# 生成子弹，需要控制发射频率

# 首先判断玩家飞机没有被击中

**if** **not** player.is\_hit:

**if** shoot\_frequency % 15 == 0:

player.shoot(bullet\_img)

shoot\_frequency += 1

**if** shoot\_frequency >= 15:

shoot\_frequency = 0

# 生成敌机，需要控制生成频率

**if** enemy\_frequency % 50 == 0:

enemy1\_pos = [random.randint(0, SCREEN\_WIDTH - enemy1\_rect.width), 0]

enemy1 = Enemy(enemy1\_img, enemy1\_down\_imgs, enemy1\_pos)

enemies1.add(enemy1)

enemy\_frequency += 1

**if** enemy\_frequency >= 100:

enemy\_frequency = 0

**for** bullet **in** player.bullets:

# 以固定速度移动子弹

bullet.move()

# 移动出屏幕后删除子弹

**if** bullet.rect.bottom < 0:

player.bullets.remove(bullet)

**for** enemy **in** enemies1:

#2. 移动敌机

enemy.move()

#3. 敌机与玩家飞机碰撞效果处理

**if** pygame.sprite.collide\_circle(enemy, player):

enemies\_down.add(enemy)

enemies1.remove(enemy)

player.is\_hit = **True**

**break**

#4. 移动出屏幕后删除飞机

**if** enemy.rect.top < 0:

enemies1.remove(enemy)

#敌机被子弹击中效果处理

# 将被击中的敌机对象添加到击毁敌机Group中，用来渲染击毁动画

enemies1\_down = pygame.sprite.groupcollide(enemies1, player.bullets, 1, 1)

**for** enemy\_down **in** enemies1\_down:

enemies\_down.add(enemy\_down)

# 绘制背景

screen.fill(0)

screen.blit(background, (0, 0))

# 绘制玩家飞机

**if** **not** player.is\_hit:

screen.blit(player.image[player.img\_index], player.rect)

# 更换图片索引使飞机有动画效果

player.img\_index = shoot\_frequency / 8

**else**:

# 玩家飞机被击中后的效果处理

player.img\_index = player\_down\_index / 8

screen.blit(player.image[player.img\_index], player.rect)

player\_down\_index += 1

**if** player\_down\_index > 47:

# 击中效果处理完成后游戏结束

running = **False**

# 敌机被子弹击中效果显示

**for** enemy\_down **in** enemies\_down:

**if** enemy\_down.down\_index == 0:

**pass**

**if** enemy\_down.down\_index > 7:

enemies\_down.remove(enemy\_down)

score += 1000

**continue**

screen.blit(enemy\_down.down\_imgs[enemy\_down.down\_index / 2], enemy\_down.rect)

enemy\_down.down\_index += 1

# 显示子弹

player.bullets.draw(screen)

# 显示敌机

enemies1.draw(screen)

#### 敌机处理

敌机需要随机在界面上方产生，并以一定速度向下移动。

**详细步骤**

1. 生成敌机，需要控制生成频率
2. 移动敌机
3. 敌机与玩家飞机碰撞效果处理
4. 移动出屏幕后删除敌机
5. 敌机被子弹击中效果处理

…..

#### 完整代码参考

# -\*- coding: utf-8 -\*-

**import** pygame

**from** sys **import** exit

**from** pygame.locals **import** \*

**import** random

# 设置游戏屏幕大小

SCREEN\_WIDTH = 480

SCREEN\_HEIGHT = 800

# 子弹类

**class** **Bullet**(pygame.sprite.Sprite):

**def** **\_\_init\_\_**(self, bullet\_img, init\_pos):

pygame.sprite.Sprite.\_\_init\_\_(self)

self.image = bullet\_img

self.rect = self.image.get\_rect()

self.rect.midbottom = init\_pos

self.speed = 10

**def** **move**(self):

self.rect.top -= self.speed

# 玩家飞机类

**class** **Player**(pygame.sprite.Sprite):

**def** **\_\_init\_\_**(self, plane\_img, player\_rect, init\_pos):

pygame.sprite.Sprite.\_\_init\_\_(self)

self.image = [] # 用来存储玩家飞机图片的列表

**for** i **in** range(len(player\_rect)):

self.image.append(plane\_img.subsurface(player\_rect[i]).convert\_alpha())

self.rect = player\_rect[0] # 初始化图片所在的矩形

self.rect.topleft = init\_pos # 初始化矩形的左上角坐标

self.speed = 8 # 初始化玩家飞机速度，这里是一个确定的值

self.bullets = pygame.sprite.Group() # 玩家飞机所发射的子弹的集合

self.img\_index = 0 # 玩家飞机图片索引

self.is\_hit = **False** # 玩家是否被击中

# 发射子弹

**def** **shoot**(self, bullet\_img):

bullet = Bullet(bullet\_img, self.rect.midtop)

self.bullets.add(bullet)

# 向上移动，需要判断边界

**def** **moveUp**(self):

**if** self.rect.top <= 0:

self.rect.top = 0

**else**:

self.rect.top -= self.speed

# 向下移动，需要判断边界

**def** **moveDown**(self):

**if** self.rect.top >= SCREEN\_HEIGHT - self.rect.height:

self.rect.top = SCREEN\_HEIGHT - self.rect.height

**else**:

self.rect.top += self.speed

# 向左移动，需要判断边界

**def** **moveLeft**(self):

**if** self.rect.left <= 0:

self.rect.left = 0

**else**:

self.rect.left -= self.speed

# 向右移动，需要判断边界

**def** **moveRight**(self):

**if** self.rect.left >= SCREEN\_WIDTH - self.rect.width:

self.rect.left = SCREEN\_WIDTH - self.rect.width

**else**:

self.rect.left += self.speed

# 敌机类

**class** **Enemy**(pygame.sprite.Sprite):

**def** **\_\_init\_\_**(self, enemy\_img, enemy\_down\_imgs, init\_pos):

pygame.sprite.Sprite.\_\_init\_\_(self)

self.image = enemy\_img

self.rect = self.image.get\_rect()

self.rect.topleft = init\_pos

self.down\_imgs = enemy\_down\_imgs

self.speed = 2

self.down\_index = 0

# 敌机移动，边界判断及删除在游戏主循环里处理

**def** **move**(self):

self.rect.top += self.speed

# 初始化PyGame

pygame.init()

# 设置游戏界面大小、背景图片及标题

# 游戏界面像素大小

screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))

# 游戏界面标题

pygame.display.set\_caption('飞机大战')

# 背景图

background = pygame.image.load('resources/image/background.png').convert()

# Game Over的背景图

game\_over = pygame.image.load('resources/image/gameover.png')

# 飞机及子弹图片集合

plane\_img = pygame.image.load('resources/image/shoot.png')

# 设置玩家飞机不同状态的图片列表，多张图片展示为动画效果

player\_rect = []

player\_rect.append(pygame.Rect(0, 99, 102, 126)) # 玩家飞机图片

player\_rect.append(pygame.Rect(165, 360, 102, 126))

player\_rect.append(pygame.Rect(165, 234, 102, 126)) # 玩家爆炸图片

player\_rect.append(pygame.Rect(330, 624, 102, 126))

player\_rect.append(pygame.Rect(330, 498, 102, 126))

player\_rect.append(pygame.Rect(432, 624, 102, 126))

player\_pos = [200, 600]

player = Player(plane\_img, player\_rect, player\_pos)

# 子弹图片

bullet\_rect = pygame.Rect(1004, 987, 9, 21)

bullet\_img = plane\_img.subsurface(bullet\_rect)

# 敌机不同状态的图片列表，多张图片展示为动画效果

enemy1\_rect = pygame.Rect(534, 612, 57, 43)

enemy1\_img = plane\_img.subsurface(enemy1\_rect)

enemy1\_down\_imgs = []

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(267, 347, 57, 43)))

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(873, 697, 57, 43)))

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(267, 296, 57, 43)))

enemy1\_down\_imgs.append(plane\_img.subsurface(pygame.Rect(930, 697, 57, 43)))

enemies1 = pygame.sprite.Group()

# 存储被击毁的飞机，用来渲染击毁动画

enemies\_down = pygame.sprite.Group()

# 初始化射击及敌机移动频率

shoot\_frequency = 0

enemy\_frequency = 0

# 玩家飞机被击中后的效果处理

player\_down\_index = 16

# 初始化分数

score = 0

# 游戏循环帧率设置

clock = pygame.time.Clock()

# 判断游戏循环退出的参数

running = **True**

# 游戏主循环

**while** running:

# 控制游戏最大帧率为60

clock.tick(60)

# 生成子弹，需要控制发射频率

# 首先判断玩家飞机没有被击中

**if** **not** player.is\_hit:

**if** shoot\_frequency % 15 == 0:

player.shoot(bullet\_img)

shoot\_frequency += 1

**if** shoot\_frequency >= 15:

shoot\_frequency = 0

# 生成敌机，需要控制生成频率

**if** enemy\_frequency % 50 == 0:

enemy1\_pos = [random.randint(0, SCREEN\_WIDTH - enemy1\_rect.width), 0]

enemy1 = Enemy(enemy1\_img, enemy1\_down\_imgs, enemy1\_pos)

enemies1.add(enemy1)

enemy\_frequency += 1

**if** enemy\_frequency >= 100:

enemy\_frequency = 0

**for** bullet **in** player.bullets:

# 以固定速度移动子弹

bullet.move()

# 移动出屏幕后删除子弹

**if** bullet.rect.bottom < 0:

player.bullets.remove(bullet)

**for** enemy **in** enemies1:

#2. 移动敌机

enemy.move()

#3. 敌机与玩家飞机碰撞效果处理

**if** pygame.sprite.collide\_circle(enemy, player):

enemies\_down.add(enemy)

enemies1.remove(enemy)

player.is\_hit = **True**

**break**

#4. 移动出屏幕后删除飞机

**if** enemy.rect.top < 0:

enemies1.remove(enemy)

#敌机被子弹击中效果处理

# 将被击中的敌机对象添加到击毁敌机Group中，用来渲染击毁动画

enemies1\_down = pygame.sprite.groupcollide(enemies1, player.bullets, 1, 1)

**for** enemy\_down **in** enemies1\_down:

enemies\_down.add(enemy\_down)

# 绘制背景

screen.fill(0)

screen.blit(background, (0, 0))

# 绘制玩家飞机

**if** **not** player.is\_hit:

screen.blit(player.image[player.img\_index], player.rect)

# 更换图片索引使飞机有动画效果

player.img\_index = shoot\_frequency / 8

**else**:

# 玩家飞机被击中后的效果处理

player.img\_index = player\_down\_index / 8

screen.blit(player.image[player.img\_index], player.rect)

player\_down\_index += 1

**if** player\_down\_index > 47:

# 击中效果处理完成后游戏结束

running = **False**

# 敌机被子弹击中效果显示

**for** enemy\_down **in** enemies\_down:

**if** enemy\_down.down\_index == 0:

**pass**

**if** enemy\_down.down\_index > 7:

enemies\_down.remove(enemy\_down)

score += 1000

**continue**

screen.blit(enemy\_down.down\_imgs[enemy\_down.down\_index / 2], enemy\_down.rect)

enemy\_down.down\_index += 1

# 显示子弹

player.bullets.draw(screen)

# 显示敌机

enemies1.draw(screen)

# 绘制得分

score\_font = pygame.font.Font(**None**, 36)

score\_text = score\_font.render(str(score), **True**, (128, 128, 128))

text\_rect = score\_text.get\_rect()

text\_rect.topleft = [10, 10]

screen.blit(score\_text, text\_rect)

# 更新屏幕

pygame.display.update()

# 处理游戏退出

**for** event **in** pygame.event.get():

**if** event.type == pygame.QUIT:

pygame.quit()

exit()

# 获取键盘事件（上下左右按键）

key\_pressed = pygame.key.get\_pressed()

# 处理键盘事件（移动飞机的位置）

**if** key\_pressed[K\_w] **or** key\_pressed[K\_UP]:

player.moveUp()

**if** key\_pressed[K\_s] **or** key\_pressed[K\_DOWN]:

player.moveDown()

**if** key\_pressed[K\_a] **or** key\_pressed[K\_LEFT]:

player.moveLeft()

**if** key\_pressed[K\_d] **or** key\_pressed[K\_RIGHT]:

player.moveRight()

# 游戏Game Over后显示最终得分

font = pygame.font.Font(**None**, 48)

text = font.render('Score: '+ str(score), **True**, (255, 0, 0))

text\_rect = text.get\_rect()

text\_rect.centerx = screen.get\_rect().centerx

text\_rect.centery = screen.get\_rect().centery + 24

screen.blit(game\_over, (0, 0))

screen.blit(text, text\_rect)

# 显示得分并处理游戏退出

**while** 1:

**for** event **in** pygame.event.get():

**if** event.type == pygame.QUIT:

pygame.quit()

exit()

pygame.display.update()

## 生成汉字图片字库

本文主要介绍pygame模块渲染字体。

### 环境搭建

**安装pygame和PIL模块**

sudo apt-get install python-pygame python-imaging

安装完成后，继续在命令行输入python进入python的交互式环境，输入以下命令检查pygame模块是否已经安装成功：

# 导入pygame模块

**import** pygame,PIL

# 退出交互式环境

exit()

**常用汉字**

根据中国大陆常用字标准，常用汉字包含：[现代汉语常用字表](https://zh.wikipedia.org/wiki/%E7%8E%B0%E4%BB%A3%E6%B1%89%E8%AF%AD%E5%B8%B8%E7%94%A8%E5%AD%97%E8%A1%A8)常用部分为2500字和[通用规范汉字表](https://zh.wikipedia.org/wiki/%E9%80%9A%E7%94%A8%E8%A7%84%E8%8C%83%E6%B1%89%E5%AD%97%E8%A1%A8)常用字集3500字，我们可以通过网上搜索到这些文字，本次项目课中已经提前将这些字保存到电子文档word.txt。下面是下载该字表文档：

# 创建一个文件夹，然后进入该文件夹下载汉字文件

mkdir chinese

cd chinese

mkdir word # 创建一个文件夹存放汉字图片库

wget http://labfile.oss.aliyuncs.com/courses/382/word.txt

### 使用pygame渲染字体

python中常用的图像处理库是PIL模块，但是如果直接使用PIL将中文文字渲染到图片上，那么文字将不能正常显示，也就说可能是PIL目前的版本中不能正确的处理非ASCII字符的点阵字符的渲染。因此，使用pygame渲染[点阵字体（位图字体）](http://baike.baidu.com/view/545607.htm)。

先查看系统拥有的中文字体：

fc-list :lang=zh-cn

结果：

/usr/share/fonts/truetype/arphic/uming.ttc: AR PL UMing TW MBE:style=Light

/usr/share/fonts/truetype/arphic/ukai.ttc: AR PL UKai CN:style=Book

/usr/share/fonts/truetype/arphic/ukai.ttc: AR PL UKai HK:style=Book

/usr/share/fonts/truetype/arphic/ukai.ttc: AR PL UKai TW:style=Book

/usr/share/fonts/truetype/wqy/wqy-microhei.ttc: WenQuanYi Micro Hei,文泉驛微米黑,文泉驿微米黑:style=Regular

/usr/share/fonts/truetype/droid/DroidSansFallbackFull.ttf: Droid Sans Fallback:style=Regular

/usr/share/fonts/truetype/arphic/ukai.ttc: AR PL UKai TW MBE:style=Book

/usr/share/fonts/truetype/arphic/uming.ttc: AR PL UMing TW:style=Light

/usr/share/fonts/truetype/arphic/uming.ttc: AR PL UMing CN:style=Light

/usr/share/fonts/truetype/arphic/uming.ttc: AR PL UMing HK:style=Light

/usr/share/fonts/truetype/wqy/wqy-microhei.ttc: WenQuanYi Micro Hei Mono,文泉驛等寬微米黑,文泉驿等宽微米黑:style=Regular

有11种中文字体，这里选择的字体的时候直接输入该字体的完整路径即可：

/usr/share/fonts/truetype/wqy/wqy-microhei.ttc

下面是具体的代码：

# -\*- coding: utf-8 -\*-

# 导入必要的模块

**import** pygame

**import** Image

**import** StringIO,os

# 打开文件，读取文件内容

f = open('word.txt','r')

words = f.readlines()[0].strip()#使用readlines()函数读取文件后再添加[0]表示选取列表中的字符串，还有记得在后面加上strip()去除字符串的首尾中的空白符

f.close()

**def** **pasteWord**(words):

'''定义一个渲染文字的函数'''

# 初始化pygame，并加载字体

pygame.init()

os.chdir('./word')

font\_path = '/usr/share/fonts/truetype/wqy/wqy-microhei.ttc'

font = pygame.font.Font(font\_path,22)

# 以空格分割字符串，遍历每一个汉字，对没有渲染的文字进行处理

text\_list = words.split(' ')

length = len(text\_list)

**for** i **in** range(length):

text = text\_list[i].decode('utf-8','ignore')

imgName = text\_list[i] + '.png'

**if** os.path.isfile(imgName):

**continue**

**else**:

paste(text,font,imgName)

**def** **paste**(text,font,imgName,area=(5,3)):

'''创建新的白色图片文件，并对文字进行渲染，最后使用Image模块令其生成图片'''

im = Image.new('RGB', (30,30), (255,255,255))

rtext = font.render(text, **True**, (0,0,0), (255,255,255))

# 将渲染结果rtext保存到StringIO对象中，再使用Image模块读取该对象

sio = StringIO.StringIO()

pygame.image.save(rtext, sio)

sio.seek(0)

line = Image.open(sio)

im.paste(line,area)

im.save(imgName)

# 运行该文件的时候执行pasteWord函数

**if** \_\_name\_\_ == '\_\_main\_\_':

pasteWord(words)

## 基于 Python 的文件备份

#!/usr/bin/python

# -\*- coding: utf-8 -\*-

#Filename : backUp.py

#导入模板

**import** Tkinter

**import** os

**import** time

#定义函数，这里才是精髓所在

**def** **backup**():

**global** entry\_source

**global** entry\_target

source = entry\_source.get()

target\_dir = entry\_target.get()

today\_dir = target\_dir + time.strftime('%Y%m%d')

time\_dir = time.strftime("%H%M%S")

touch = today\_dir + os.sep + time\_dir + '.zip'

command\_touch = "zip -qr " + touch +' '+ source

**print** command\_touch

**print** source

**print** target\_dir

**if** os.path.exists(today\_dir)==0:

os.mkdir(today\_dir)

**if** os.system(command\_touch)==0:

**print** 'Success backup Up'

**else**:

**print** 'Failed backup'

#从这里开始呢，则是开始界面的编写及布局

root = Tkinter.Tk()

root.title('BackUp')

root.geometry("200x200")

#第一行的两个控件

lbl\_source = Tkinter.Label(root, text='Source')

lbl\_source.grid(row=0, column=0)

entry\_source = Tkinter.Entry(root)

entry\_source.grid(row=0,column=1)

#第二行的两个控件

lbl\_target = Tkinter.Label(root, text='Target')

lbl\_target.grid(row=1, column=0)

entry\_target = Tkinter.Entry(root)

entry\_target.grid(row=1,column=1)

#第三行的一个按钮控件

but\_back = Tkinter.Button(root,text='BackUp')

but\_back.grid(row=3, column=0)

but\_back["command"] = backup

#界面的开始

root.mainloop()

## Kaggle入门：泰坦尼克号幸存者项目

Kaggle是一个线上数据科学竞赛类网站。本课程将使用python科学计算工具对Kaggle提供的泰坦尼克号上的人员数据进行分析建模。

## 基于scrapy爬虫的天气数据采集

### 安装所需软件

# 安装依赖的包

$ sudo apt-get update

$ sudo apt-get install python-lxml python-dev libffi-dev libevent-dev python-pip

### 安装Scrapy

sudo pip install Scrapy

### 创建scrapy项目

创建一个scrapy的项目:

scrapy startproject weather

生成的目录结构：

weather

├── scrapy.cfg #项目的配置文件

└── weather

├── \_\_init\_\_.py

├── items.py #要提取的元素，相当于一个容器

├── pipelines.py #存文件时或者发送到其他地方可用其编写

├── settings.py #项目的设置文件

└── spiders #放置spider代码的目录

└── \_\_init\_\_.py

2 directories, 6 files

### 实现爬虫

Item是保存爬取到的数据的容器；其使用方法和[**Python**](http://lib.csdn.net/base/11)字典类似，并且提供了额外保护机制来避免拼写错误导致的未定义字段错误。

weather\weather\items.py

# -\*- coding: utf-8 -\*-

# Define here the models for your scraped items

#

# See documentation in:

# http://doc.scrapy.org/en/latest/topics/items.html

**import** scrapy

**class** **WeatherItem**(scrapy.Item):

# define the fields for your item here like:

# name = scrapy.Field()

city = scrapy.Field()

date = scrapy.Field()

dayDesc = scrapy.Field()

dayTemp = scrapy.Field()

**pass**

下面实现爬虫，使用xpath分解html中的标签，为了创建一个Spider，您必须继承scrapy.Spider类， 且定义以下三个属性:

1.name:用于区别Spider。该名字必须是唯一的，您不可以为不同的Spider设定相同的名字。

2.start\_urls:包含了Spider在启动时进行爬取的url列表。因此，第一个被获取到的页面将是其中之一。后续的URL则从初始的URL获取到的数据中提取。

3.parse()是spider的一个方法。被调用时，每个初始URL完成下载后生成的Response对象将会作为唯一的参数传递给该函数。该方法负责解析返回的数据(responsedata)，提取数据(生成item)以及生成需要进一步处理的URL的Request对象。

weather\spiders\weather\_spider1.py

# -\*- coding: utf-8 -\*-

**import** scrapy

**from** weather.items **import** WeatherItem

**class** **WeatherSpider**(scrapy.Spider):

name = 'weather\_spider1'

allowed\_domains = ['sina.com.cn']

start\_urls = ['http://weather.sina.com.cn/beijing']

**def** **parse**(self, response):

item = WeatherItem()

item['city'] = response.xpath('//\*[@id="slider\_ct\_name"]/text()').extract()

tenDay = response.xpath('//\*[@id="blk\_fc\_c0\_scroll"]')

item['date'] = tenDay.css('p.wt\_fc\_c0\_i\_date::text').extract()

item['dayDesc'] = tenDay.css('img.icons0\_wt::attr(title)').extract()

item['dayTemp'] = tenDay.css('p.wt\_fc\_c0\_i\_temp::text').extract()

**return** item

Scrapy使用了一种基于XPath和CSS表达式机制:Scrapy Selectors。

这里给出XPath表达式的例子及对应的含义:

/html/head/title:选择HTML文档中<head>标签内的<title>元素

/html/head/title/text():选择上面提到的<title>元素的文字

//td:选择所有的<td>元素

//div[@class="mine"]:选择所有具有class="mine"属性的div元素

上边仅仅是几个简单的XPath例子，XPath实际上要比这远远强大的多。

为了配合XPath，Scrapy除了提供了Selector之外，还提供了方法来避免每次从response中提取数据时生成selector的麻烦。

Selector有四个基本的方法(点击相应的方法可以看到详细的API文档):

xpath():传入xpath表达式，返回该表达式所对应的所有节点的selectorlist列表 。

css():传入CSS表达式，返回该表达式所对应的所有节点的selectorlist列表.

extract():序列化该节点为unicode字符串并返回list。

re():根据传入的正则表达式对数据进行提取，返回unicode字符串list列表。

运行爬虫：

scrapy crawl weather\_spider1 -o weather.json

### 中文处理

#### 针对交互输出

由于extract()序列化该节点为unicode字符串并返回list，所以中文不能正常显示。将其转换为utf-8在输出即可：

**for** str **in** item['city']:

**print** (str.encode('utf-8'))

encode()只针对str数据结构，如果不是，可以先转换为str。

#### 针对存储

见下一节。

### 保存结果

Weather\ pipelines.py

# -\*- coding: utf-8 -\*-

# Define your item pipelines here

#

# Don't forget to add your pipeline to the ITEM\_PIPELINES setting

# See: http://doc.scrapy.org/en/latest/topics/item-pipeline.html

**class** **WeatherPipeline**(object):

**def** **\_\_init\_\_**(self):

self.file = open('wea.txt', 'w+')

**def** **process\_item**(self, item, spider):

city = item['city'][0].encode('utf-8')

self.file.write('city: '+str(city)+'\n\n')

date = item['date']

desc = item['dayDesc']

dayDesc = desc[1::2]

nightDesc = desc[0::2]

dayTemp = item['dayTemp']

weaitem = zip(date, dayDesc, nightDesc, dayTemp)

**for** i **in** range(len(weaitem)):

item = weaitem[i]

d = item[0]

dd = item[1]

nd = item[2]

ta = item[3].split('/')

dt = ta[0]

nt = ta[1]

txt = 'data: {0} \t\t day:{1}({2}) \t\t night:{3}({4}) \n\n'.format(

d,

dd.encode('utf-8'),

dt.encode('utf-8'),

nd.encode('utf-8'),

nt.encode('utf-8'),

)

self.file.write(txt)

**return** item

## Python文本解析器

使用 Python 来解析纯文本生成一个 HTML 页面的小程序。从中我们学习到Python基本语法，HTML标记语言，以及如何用 Python 将纯文本分成一个一个的文本块，并对它对解析。

### 文本块生成器

首先我们需要有一个文本块生成器把纯文本分成一个一个的文本块，以便接下来对每一个文本快进行解析，util.py 代码如下：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**def** **lines**(file):

'''

生成器，在文本最后加一空行

'''

**for** line **in** file:

**yield** line

**yield** '\n'

**def** **blocks**(file):

'''

生成器，生成单独的文本块

'''

block = []

**for** line **in** lines(file):

**if** line.strip():

block.append(line)

**elif** block:

**yield** ''.join(block).strip()

block = []

### 处理程序

通过文本生成器我们得到了一个一个的文本块，然后需要有处理程序对不同的文本块加相应的 HTML 标记，handlers.py 代码如下：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**class** **Handler**:

'''

处理程序父类

'''

**def** **callback**(self, prefix, name, \*args):

method = getattr(self, prefix + name, **None**)

**if** callable(method): **return** method(\*args)

**def** **start**(self, name):

self.callback('start\_', name)

**def** **end**(self, name):

self.callback('end\_', name)

**def** **sub**(self, name):

**def** **substitution**(match):

result = self.callback('sub\_', name, match)

**if** result **is** **None**: result = match.group(0)

**return** result

**return** substitution

**class** **HTMLRenderer**(Handler):

'''

HTML处理程序，给文本块加相应的HTML标记

'''

**def** **start\_document**(self):

**print**('<html><head><title>ShiYanLou</title></head><body>')

**def** **end\_document**(self):

**print**('</body></html>')

**def** **start\_paragraph**(self):

**print**('<p style="color: #444;">')

**def** **start\_heading**(self):

**print** '<h2 style="color: #68BE5D;">'

**def** **end\_heading**(self):

**print** '</h2>'

**def** **start\_list**(self):

**print** '<ul style="color: #363736;">'

**def** **end\_list**(self):

**print** '</ul>'

**def** **start\_listitem**(self):

**print** '<li>'

**def** **end\_listitem**(self):

**print** '</li>'

**def** **start\_title**(self):

**print** '<h1 style="color: #1ABC9C;">'

**def** **end\_title**(self):

**print** '</h1>'

**def** **sub\_emphasis**(self, match):

**return** '<em>%s</em>' % match.group(1)

**def** **sub\_url**(self, match):

**return** '<a target="\_blank" style="text-decoration: none;color: #BC1A4B;" href="%s">%s</a>' % (match.group(1), match.group(1))

**def** **sub\_mail**(self, match):

**return** '<a style="text-decoration: none;color: #BC1A4B;" href="mailto:%s">%s</a>' % (match.group(1), match.group(1))

**def** **feed**(self, data):

**print** data

### 规则

有了处理程序和文本块生成器，接下来就需要一定的规则来判断每个文本块交给处理程序将要加什么标记，rules.py 代码如下：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**class** **Rule**:

"""

规则父类

"""

**def** **action**(self, block, handler):

"""

加标记

"""

handler.start(self.type)

handler.feed(block)

handler.end(self.type)

**return** **True**

**class** **HeadingRule**(Rule):

"""

一号标题规则

"""

type = 'heading'

**def** **condition**(self, block):

"""

判断文本块是否符合规则

"""

**return** **not** '\n' **in** block **and** len(block) <= 70 **and** **not** block[-1] == ':'

**class** **TitleRule**(HeadingRule):

"""

二号标题规则

"""

type = 'title'

first = **True**

**def** **condition**(self, block):

**if** **not** self.first: **return** **False**

self.first = **False**

**return** HeadingRule.condition(self, block);

**class** **ListItemRule**(Rule):

"""

列表项规则

"""

type = 'listitem'

**def** **condition**(self, block):

**return** block[0] == '-'

**def** **action**(self, block, handler):

handler.start(self.type)

handler.feed(block[1:].strip())

handler.end(self.type)

**return** **True**

**class** **ListRule**(ListItemRule):

"""

列表规则

"""

type = 'list'

inside = **False**

**def** **condition**(self, block):

**return** **True**

**def** **action**(self, block, handler):

**if** **not** self.inside **and** ListItemRule.condition(self, block):

handler.start(self.type)

self.inside = **True**

**elif** self.inside **and** **not** ListItemRule.condition(self, block):

handler.end(self.type)

self.inside = **False**

**return** **False**

**class** **ParagraphRule**(Rule):

"""

段落规则

"""

type = 'paragraph'

**def** **condition**(self, block):

**return** **True**

### 解析

最后我们就可以进行解析了，markup.py 代码如下：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**import** sys, re

**from** handlers **import** \*

**from** util **import** \*

**from** rules **import** \*

**class** **Parser**:

"""

解析器父类

"""

**def** **\_\_init\_\_**(self, handler):

self.handler = handler

self.rules = []

self.filters = []

**def** **addRule**(self, rule):

"""

添加规则

"""

self.rules.append(rule)

**def** **addFilter**(self, pattern, name):

"""

添加过滤器

"""

**def** **filter**(block, handler):

**return** re.sub(pattern, handler.sub(name), block)

self.filters.append(filter)

**def** **parse**(self, file):

"""

解析

"""

self.handler.start('document')

**for** block **in** blocks(file):

**for** filter **in** self.filters:

block = filter(block, self.handler)

**for** rule **in** self.rules:

**if** rule.condition(block):

last = rule.action(block, self.handler)

**if** last: **break**

self.handler.end('document')

**class** **BasicTextParser**(Parser):

"""

纯文本解析器

"""

**def** **\_\_init\_\_**(self, handler):

Parser.\_\_init\_\_(self, handler)

self.addRule(ListRule())

self.addRule(ListItemRule())

self.addRule(TitleRule())

self.addRule(HeadingRule())

self.addRule(ParagraphRule())

self.addFilter(r'\\*(.+?)\\*', 'emphasis')

self.addFilter(r'(http://[\.a-zA-Z/]+)', 'url')

self.addFilter(r'([\.a-zA-Z]+@[\.a-zA-Z]+[a-zA-Z]+)', 'mail')

"""

运行程序

"""

handler = HTMLRenderer()

parser = BasicTextParser(handler)

parser.parse(sys.stdin)

### 运行

首先创建测试文件test.txt：

Welcome to ShiYanLou

ShiYanLou is the first experiment with IT as the core of online education platform.\*Our aim is to do the experiment, easy to learn IT\*.

Course

-Basic Course

-Project Course

-Evaluation Course

Contact us

-Web:http://www.shiyanlou.com

-QQ Group:241818371

-E-mail:support@shiyanlou.com

运行：

python markup.py < test.txt > test.html

### 程序说明

**def** **parse**(self, file):

"""

解析

"""

self.handler.start('document')

**for** block **in** blocks(file):#针对每个block处理

**for** filter **in** self.filters:

'''

针对block,首先处理 emphasis, url, mail标记

'''

block = filter(block, self.handler)

**for** rule **in** self.rules:

**if** rule.condition(block):

last = rule.action(block, self.handler)

**if** last: **break**

**if** last: **break**

self.handler.end('document')

针对每个block，首先执行：

block = filter(block, self.handler)

实际是：

**return** re.sub(pattern, handler.sub(name), block)

re.sub的功能：**对于输入的一个字符串，利用正则表达式（的强大的字符串处理功能），去实现（相对复杂的）字符串替换处理，然后返回被替换后的字符串**

pattern：表示正则中的模式字符串

repl：可以是字符串，也可以是函数，表示被替换成的字符串或者是被函数处理以后的字符串。

String：被处理的对象。

比如下面内容：

ShiYanLou is the first experiment with IT as the core of online education platform.\*Our aim is to do the experiment, easy to learn IT\*.

Filter如下：

self.addFilter(r'\\*(.+?)\\*', 'emphasis')

则对应的re.sub参数为：

re.sub(r'\\*(.+?)\\*', handler.sub('emphasis'), block)

执行class Handler中的sub(self, name)，实际执行的是**substitution**(match)

**def** **sub**(self, name):

**def** **substitution**(match):

result = self.callback('sub\_', name, match)

**if** result **is** **None**: result = match.group(0)

**return** result

**return** substitution

其中match.group(0)为：

\*Our aim is to do the experiment, easy to learn IT\*

match.group(1)为：

Our aim is to do the experiment, easy to learn IT

接着执行：

**def** **sub\_emphasis**(self, match):

**return** '<em>%s</em>' % match.group(1)

返回值为：

'<em> Our aim is to do the experiment, easy to learn IT</em>'

替换后的内容为：

ShiYanLou is the first experiment with IT as the core of online education platform.<em>Our aim is to do the experiment, easy to learn IT</em>.

之后交给rulers处理，ParagraphRule. Condition为真：

**def** **action**(self, block, handler):

"""

加标记

"""

handler.start(self.type)

handler.feed(block)

handler.end(self.type)

**return** **True**

**def** **start\_paragraph**(self):

**print**('<p style="color: #444;">')

**def** **feed**(self, data):

**print**(data)

结果为：

<p style="color: #444;">

ShiYanLou is the first experiment with IT as the core of online education platform.<em>Our aim is to do the experiment, easy to learn IT</em>.

## 破解验证码

使用的验证码captcha.png

Z:\python\projects\crack-verification-code\captcha.png

### 提取文本图片

新建 crack.py 文件，进行编辑。

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**from** PIL **import** Image

im = Image.open("captcha.png")

#将图片转换为8位像素模式

im = im.convert("P")

#打印颜色直方图

**print**(im.histogram())

输出：

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 2, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 0, 1, 2, 0, 1, 0, 0, 1, 0, 2, 0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 3, 1, 3, 3, 0, 0, 0, 0, 0, 0, 1, 0, 3, 2, 132, 1, 1, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 15, 0, 1, 0, 1, 0, 0, 8, 1, 0, 0, 0, 0, 1, 6, 0, 2, 0, 0, 0, 0, 18, 1, 1, 1, 1, 1, 2, 365, 115, 0, 1, 0, 0, 0, 135, 186, 0, 0, 1, 0, 0, 0, 116, 3, 0, 0, 0, 0, 0, 21, 1, 1, 0, 0, 0, 2, 10, 2, 0, 0, 0, 0, 2, 10, 0, 0, 0, 0, 1, 0, 625]

颜色直方图的每一位数字都代表了在图片中含有对应位的颜色的像素的数量。

每个像素点可表现256种颜色，你会发现白点是最多（白色序号255的位置，也就是最后一位，可以看到，有625个白色像素）。红像素在序号200左右，我们可以通过排序，得到有用的颜色。

his = im.histogram()

values = {}#dict, 使用键-值（key-value）存储

**for** i **in** range(256):

values[i] = his[i]

'''

sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序

key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。

要进行反向排序，不必改动key函数，可以传入第三个参数reverse=True

'''

**for** j,k **in** sorted(values.items(),key=**lambda** x:x[1],reverse = **True**)[:10]:

**print** j,k

输出：

255 625

212 365

220 186

219 135

169 132

227 116

213 115

234 21

205 18

184 15

我们得到了图片中最多的10种颜色，其中 220 与 227 才是我们需要的红色和灰色，可以通过这一讯息构造一种黑白二值图片。

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**from** PIL **import** Image

im = Image.open("captcha.png")

#将图片转换为8位像素模式

im = im.convert("P")

im2 = Image.new("P",im.size,255)

**for** x **in** range(im.size[1]):

**for** y **in** range(im.size[0]):

pix = im.getpixel((y,x))

**if** pix == 220 **or** pix == 227: # these are the numbers to get

im2.putpixel((y,x),0)

im2.show()

得到的结果：

此处输入图片的描述

### 提取单个字符图片

接下来的工作是要得到单个字符的像素集合，由于例子比较简单，我们对其进行纵向切割：

inletter = **False**

foundletter=**False**

start = 0

end = 0

letters = []

**for** y **in** range(im2.size[0]):

**for** x **in** range(im2.size[1]):

pix = im2.getpixel((y,x))

**if** pix != 255:

inletter = **True**

**if** foundletter == **False** **and** inletter == **True**:

foundletter = **True**

start = y

**if** foundletter == **True** **and** inletter == **False**:

foundletter = **False**

end = y

letters.append((start,end))

inletter=**False**

**print** letters

输出：

[(6, 14), (15, 25), (27, 35), (37, 46), (48, 56), (57, 67)]

得到每个字符开始和结束的列序号。

count = 0

**for** letter **in** letters:

m = hashlib.md5()

im3 = im2.crop(( letter[0] , 0, letter[1],im2.size[1] ))

m.update("%s%s"%(time.time(),count))

im3.save("./%s.gif"%(m.hexdigest()))

count += 1

对图片进行切割，得到每个字符所在的那部分图片。

### AI 与向量空间图像识别

在这里我们使用向量空间搜索引擎来做字符识别，它具有很多优点：

* 不需要大量的训练迭代
* 不会训练过度
* 你可以随时加入／移除错误的数据查看效果
* 很容易理解和编写成代码
* 提供分级结果，你可以查看最接近的多个匹配
* 对于无法识别的东西只要加入到搜索引擎中，马上就能识别了。

当然它也有缺点，例如分类的速度比神经网络慢很多，它不能找到自己的方法解决问题等等。

关于向量空间搜索引擎的原理可以参考这篇文章：<http://ondoc.logand.com/d/2697/pdf>

Don't panic。向量空间搜索引擎名字听上去很高大上其实原理很简单。拿文章里的例子来说：

你有 3 篇文档，我们要怎么计算它们之间的相似度呢？2 篇文档所使用的相同的单词越多，那这两篇文章就越相似！但是这单词太多怎么办，就由我们来选择几个关键单词，选择的单词又被称作特征，每一个特征就好比空间中的一个维度（x，y，z 等），一组特征就是一个矢量，每一个文档我们都能得到这么一个矢量，只要计算矢量之间的夹角就能得到文章的相似度了。

全部代码：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

**from** PIL **import** Image

**import** hashlib

**import** time

**import** os

**import** math

**class** **VectorCompare**:

**def** **magnitude**(self,concordance):

total = 0

**for** word,count **in** concordance.iteritems():

total += count \*\* 2

**return** math.sqrt(total)

**def** **relation**(self,concordance1, concordance2):

relevance = 0

topvalue = 0

**for** word, count **in** concordance1.iteritems():

**if** concordance2.has\_key(word):

topvalue += count \* concordance2[word]

**return** topvalue / (self.magnitude(concordance1) \* self.magnitude(concordance2))

**def** **buildvector**(im):

d1 = {}

count = 0

**for** i **in** im.getdata():

d1[count] = i

count += 1

**return** d1

v = VectorCompare()

iconset = ['0','1','2','3','4','5','6','7','8','9','0','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']

imageset = []

**for** letter **in** iconset:

**for** img **in** os.listdir('./iconset/%s/'%(letter)):

temp = []

**if** img != "Thumbs.db" **and** img != ".DS\_Store": # windows check...

temp.append(buildvector(Image.open("./iconset/%s/%s"%(letter,img))))

imageset.append({letter:temp})

im = Image.open("captcha.png")

im2 = Image.new("P",im.size,255)

#将图片转换为8位像素模式

im = im.convert("P")

temp = {}

**for** x **in** range(im.size[1]):

**for** y **in** range(im.size[0]):

pix = im.getpixel((y,x))

temp[pix] = pix

**if** pix == 220 **or** pix == 227: # these are the numbers to get

im2.putpixel((y,x),0)

inletter = **False**

foundletter=**False**

start = 0

end = 0

letters = []

**for** y **in** range(im2.size[0]): # slice across

**for** x **in** range(im2.size[1]): # slice down

pix = im2.getpixel((y,x))

**if** pix != 255:

inletter = **True**

**if** foundletter == **False** **and** inletter == **True**:

foundletter = **True**

start = y

**if** foundletter == **True** **and** inletter == **False**:

foundletter = **False**

end = y

letters.append((start,end))

inletter=**False**

count = 0

**for** letter **in** letters:

m = hashlib.md5()

im3 = im2.crop(( letter[0] , 0, letter[1],im2.size[1] ))

guess = []

**for** image **in** imageset:

**for** x,y **in** image.iteritems():

**if** len(y) != 0:

guess.append( ( v.relation(y[0],buildvector(im3)),x) )

guess.sort(reverse=**True**)

**print** "",guess[0]

count += 1

## 端口扫描

#!/usr/bin/python

# -\*- coding:utf8 -\*-

**import** socket, time, threading, sys, os

socket.setdefaulttimeout(3)

**def** **socket\_port**(ip,port):

"""

输入IP和端口号，扫描判断端口是否开放

"""

**try**:

**if** port>=65535:

**print** '端口扫描结束'

os.exit(1)#一般来说os.\_exit() 用于在线程中退出, sys.exit() 用于在主线程中退出。

s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

result=s.connect\_ex((ip,port))

**if** result==0:

lock.acquire()

**print** ip,':',port,'端口开放' #在多线程中，print时要用锁

lock.release()

s.close()

**except**:

lock.acquire()

**print** '端口扫描异常'

lock.release()

**def** **ip\_scan**(ip):

"""

输入IP，扫描IP的0-65534端口情况

"""

**try**:

**print** '开始扫描 %s' % ip

start\_time=time.time()

**for** i **in** range(0,65534):

thread = threading.Thread(target=socket\_port,args=(ip,int(i),)) #for循环创建线程，每个端口开一个线程

thread.setDaemon(**True**) #将线程声明为守护线程,使其可快速退出

thread.start()

**print** '扫描端口完成，总共用时 ：%.2f' %(time.time()-start\_time)

**except**:

**print** '扫描ip出错'

**def** **get\_remote\_machine\_info**(Domain):

**try**:

**return** socket.gethostbyname(Domain)

**except** socket.error,e:

**print** '%s: %s'%(Domain,e)

**return** 0

**if** \_\_name\_\_=='\_\_main\_\_':

url=sys.argv[1]

ip = get\_remote\_machine\_info(url)

lock=threading.Lock()

ip\_scan(ip)

运行：python scan.py www.xxx.com

## Python实现简单的Web服务器

本课程项目完成过程中，我们将学习：

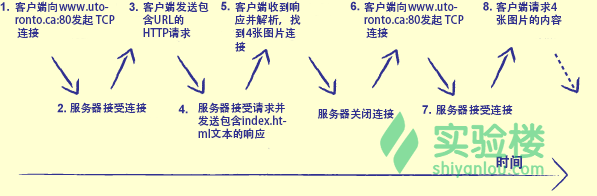
1. HTTP 协议基本原理
2. 简单的 Web 服务器框架
3. Python 语言的网络开发
4. Web 服务请求，响应及错误处理的实现
5. CGI 协议的 Python 实现
6. 使用 Python 面向对象思想重构代码

本实验使用httpie代替浏览器发送请求并在终端打印响应信息。

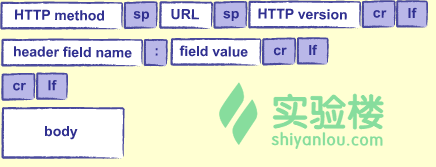
$ sudo apt-get install httpie

一般我们的web程序都运行在 TCP/IP 协议上，程序之间使用 socket(套接字) 进行通信，它能够让计算机之间的通信就像写文件和读文件一样简单。 一个 tcp socket 由一个IP地址和端口号组成。

超文本传输协议（HTTP）描述了一种程序之间交换数据的方法，它非常简单易用，在一个socket连接上，客户端首先发送请求说明它需要什么，然后服务器发送响应，并在响应中包含客户端的数据。响应数据也许是从本地磁盘上复制来的，也许是程序动态生成的。传输过程如图：

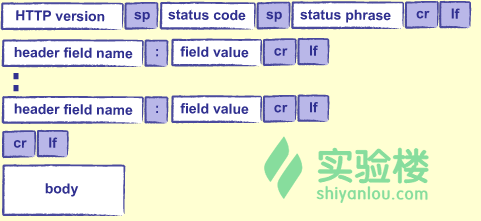


HTTP请求就是一段文本，任何程序都能生成一个http请求，就像生成文本一样简单。这段文本需要包含以下这些部分：



* HTTP method：HTTP请求方法。最常用的就是 GET(抓取数据)与POST(更新数据或者上传文件)
* URL：通常是客户端请求的文件的路径，比如 /research/experiments.html， 但是是否响应文件都是由服务器决定的。
* HTTP version：HTTP版本。通常是 HTTP/1.0 或 HTTP/1.1
* header field：HTTP头内的键值对，做一些基本设置，就像下面这样。
* #客户端接受的数据类型
* Accept: text/html
* #客户端接受的语言
* Accept-Language: en, fr
* If-Modified-Since: 16-May-2005
* body： 一些与请求有关的负载数据了。比如在一个网站登陆的时候提交登陆表单，那负载数据就是你的账号与密码信息了。

HTTP响应的结构类似于请求：



* status code：状态码。请求成功响应200，请求的文件找不到则响应404。
* status phrase：对状态码的描述。

### 你好, web

现在就来写我们第一个web服务器吧， 基本概念非常简单：

1. 等待某个人连接我们的服务器并向我们发送一个HTTP请求
2. 解析该请求
3. 了解该请求希望请求的内容
4. 服务器根据请求抓取需要的数据（从服务器本地文件中读或者程序动态生成）
5. 将数据格式化为请求需要的格式
6. 送回HTTP响应

步骤1，2，6的操作对所有web应用都是一样的，这部分内容Python标准库中的 BaseHTTPServer 模块可以帮助我们处理。我们只需要关注步骤3～5。

首先在工作目录下创建 server.py 文件

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

**import** BaseHTTPServer

**class** **RequestHandler**(BaseHTTPServer.BaseHTTPRequestHandler):

'''处理请求并返回页面'''

# 页面模板

Page = '''\

<html>

<body>

<p>Hello, web!</p>

</body>

</html>

'''

# 处理一个GET请求

**def** **do\_GET**(self):

self.send\_response(200)

self.send\_header("Content-Type", "text/html")

self.send\_header("Content-Length", str(len(self.Page)))

self.end\_headers()

self.wfile.write(self.Page)

#----------------------------------------------------------------------

**if** \_\_name\_\_ == '\_\_main\_\_':

serverAddress = ('', 8080)

server = BaseHTTPServer.HTTPServer(serverAddress, RequestHandler)

server.serve\_forever()

模块的 BaseHTTPRequestHandler 类会帮我们处理对请求的解析，并通过确定请求的方法来调用其对应的函数，比如方法是 GET ,该类就会调用名为 do\_GET 的方法。RequestHandler 继承了 BaseHTTPRequestHandler 并重写了 do\_GET 方法，其效果如代码所示是返回 Page 的内容。 Content-Type 告诉了客户端要以处理html文件的方式处理返回的内容。end\_headers 方法会插入一个空白行，如之前的request结构图所示。

运行我们的第一个 web服务器

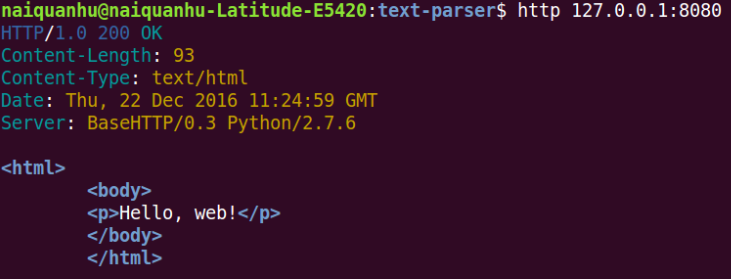
$ python server.py

可以在浏览器地址输入 127.0.0.1:8080 或http://ip:8080/进行查看



方便起见，还是让我们新开一个终端窗口，使用httpie来查看输出(之后都使用httpie来查看输出)

$ http 127.0.0.1:8080



httpie很贴心地显示了响应报文的全部内容

### 显示请求的信息

修改之前的代码来显示请求的信息，同时重新整理一下代码：

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

**import** BaseHTTPServer

**class** **RequestHandler**(BaseHTTPServer.BaseHTTPRequestHandler):

'''处理请求并返回页面'''

# ...页面模板...

Page = '''\

<html>

<body>

<table>

<tr> <td>Header</td> <td>Value</td> </tr>

<tr> <td>Date and time</td> <td>{date\_time}</td> </tr>

<tr> <td>Client host</td> <td>{client\_host}</td> </tr>

<tr> <td>Client port</td> <td>{client\_port}</td> </tr>

<tr> <td>Command</td> <td>{command}</td> </tr>

<tr> <td>Path</td> <td>{path}</td> </tr>

</table>

</body>

</html>

'''

# 处理一个GET请求

**def** **do\_GET**(self):

page = self.create\_page()

self.send\_content(page)

**def** **create\_page**(self):

values = {

'date\_time' : self.date\_time\_string(),

'client\_host' : self.client\_address[0],

'client\_port' : self.client\_address[1],

'command' : self.command,

'path' : self.path

}

page = self.Page.format(\*\*values)

**return** page

**def** **send\_content**(self, page):

self.send\_response(200)

self.send\_header("Content-type", "text/html")

self.send\_header("Content-Length", str(len(page)))

self.end\_headers()

self.wfile.write(page)

#----------------------------------------------------------------------

**if** \_\_name\_\_ == '\_\_main\_\_':

serverAddress = ('', 8080)

server = BaseHTTPServer.HTTPServer(serverAddress, RequestHandler)

server.serve\_forever()

$ http 127.0.0.1:8080/something.html



注意到它仍旧返回了200 OK而不是404 Not Found，即使 something.html 文件并不存在。那是因为我们现在的web服务器还没有实现找不到文件就返回404错误的功能。反过来说，只要我们想，可以通过编程实现任何我们想要的效果，像是随机返回一个维基百科的页面或是帮老王家订一个披萨(并不会)。

怎么解决返回404的问题呢，首先得有返回文件的功能吧。

### 响应静态页面

所以这一步就该处理静态页面了，处理静态页面就是根据请求的页面名得到磁盘上的页面文件并返回。

在当前目录下创建新文件 plain.html，这是我们测试用的静态页面

**<html>**

**<head>**

**<title>**Plain Page**</title>**

**</head>**

**<body>**

**<h1>**Plain Page**</h1>**

**<p>**Nothin' but HTML.**</p>**

**</body>**

**</html>**

至今为止的代码：

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

**import** sys, os, BaseHTTPServer

**class** **ServerException**(Exception):

'''服务器内部错误'''

**pass**

**class** **RequestHandler**(BaseHTTPServer.BaseHTTPRequestHandler):

'''处理请求并返回页面'''

# ...页面模板...

Page = '''

<html>

<body>

<table>

<tr> <td>Header</td> <td>Value</td> </tr>

<tr> <td>Date and time</td> <td>{date\_time}</td> </tr>

<tr> <td>Client host</td> <td>{client\_host}</td> </tr>

<tr> <td>Client port</td> <td>{client\_port}</td> </tr>

<tr> <td>Command</td> <td>{command}</td> </tr>

<tr> <td>Path</td> <td>{path}</td> </tr>

</table>

</body>

</html>

'''

Error\_Page = """\

<html>

<body>

<h1>Error accessing {path}</h1>

<p>{msg}</p>

</body>

</html>

"""

# 处理一个GET请求

**def** **do\_GET**(self):

**try**:

# 文件完整路径

full\_path = os.getcwd() + self.path

# 如果该路径不存在...

**if** **not** os.path.exists(full\_path):

#抛出异常：文件未找到

**raise** ServerException("'{0}' not found".format(self.path))

# 如果该路径是一个文件

**elif** os.path.isfile(full\_path):

#调用 handle\_file 处理该文件

self.handle\_file(full\_path)

# 如果该路径不是一个文件

**else**:

#抛出异常：该路径为不知名对象

**raise** ServerException("Unknown object '{0}'".format(self.path))

# 处理异常

**except** Exception **as** msg:

self.handle\_error(msg)

**def** **handle\_file**(self, full\_path):

**try**:

**with** open(full\_path, 'rb') **as** reader:

content = reader.read()

self.send\_content(content)

**except** IOError **as** msg:

msg = "'{0}' cannot be read: {1}".format(self.path, msg)

self.handle\_error(msg)

**def** **handle\_error**(self, msg):

content = self.Error\_Page.format(path=self.path, msg=msg)

self.send\_content(content, 404)

**def** **create\_page**(self):

values = {

'date\_time' : self.date\_time\_string(),

'client\_host' : self.client\_address[0],

'client\_port' : self.client\_address[1],

'command' : self.command,

'path' : self.path

}

page = self.Page.format(\*\*values)

**return** page

**def** **send\_content**(self, content, status=200):

self.send\_response(status)

self.send\_header("Content-type", "text/html")

self.send\_header("Content-Length", str(len(content)))

self.end\_headers()

self.wfile.write(content)

#----------------------------------------------------------------------

**if** \_\_name\_\_ == '\_\_main\_\_':

serverAddress = ('', 8080)

server = BaseHTTPServer.HTTPServer(serverAddress, RequestHandler)

server.serve\_forever()

### 在根url显示首页内容

大部分时候我们都希望能够直接在http://127.0.0.1:8080/ 显示主页内容。要怎么做呢，也许我们可以在do\_GET那冗长的if-elif-else判断里再加一个判断请求地址是不是根地址的分支，也许我们可以找到一个更加聪明的方法。

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

**import** sys, os, BaseHTTPServer

**class** **ServerException**(Exception):

'''服务器内部错误'''

**pass**

**class** **case\_no\_file**(object):

'''该路径不存在'''

**def** **test**(self, handler):

**return** **not** os.path.exists(handler.full\_path)

**def** **act**(self, handler):

**raise** ServerException("'{0}' not found".format(handler.path))

**class** **case\_existing\_file**(object):

'''该路径是文件'''

**def** **test**(self, handler):

**return** os.path.isfile(handler.full\_path)

**def** **act**(self, handler):

handler.handle\_file(handler.full\_path)

**class** **case\_always\_fail**(object):

'''所有情况都不符合时的默认处理类'''

**def** **test**(self, handler):

**return** **True**

**def** **act**(self, handler):

**raise** ServerException("Unknown object '{0}'".format(handler.path))

**class** **case\_directory\_index\_file**(object):

**def** **index\_path**(self, handler):

**return** os.path.join(handler.full\_path, 'index.html')

#判断目标路径是否是目录&&目录下是否有index.html

**def** **test**(self, handler):

**return** os.path.isdir(handler.full\_path) **and** os.path.isfile(self.index\_path(handler))

#响应index.html的内容

**def** **act**(self, handler):

handler.handle\_file(self.index\_path(handler))

**class** **RequestHandler**(BaseHTTPServer.BaseHTTPRequestHandler):

'''处理请求并返回页面'''

# ...页面模板...

Page = '''

<html>

<body>

<table>

<tr> <td>Header</td> <td>Value</td> </tr>

<tr> <td>Date and time</td> <td>{date\_time}</td> </tr>

<tr> <td>Client host</td> <td>{client\_host}</td> </tr>

<tr> <td>Client port</td> <td>{client\_port}</td> </tr>

<tr> <td>Command</td> <td>{command}</td> </tr>

<tr> <td>Path</td> <td>{path}</td> </tr>

</table>

</body>

</html>

'''

Error\_Page = """\

<html>

<body>

<h1>Error accessing {path}</h1>

<p>{msg}</p>

</body>

</html>

"""

# 所有可能的情况

Cases = [case\_no\_file(),

case\_existing\_file(),

case\_directory\_index\_file(),

case\_always\_fail()]

# 处理一个GET请求

**def** **do\_GET**(self):

**try**:

# 文件完整路径

self.full\_path = os.getcwd() + self.path#注意，一定要给self.full\_path赋值

#遍历所有可能的情况

**for** case **in** self.Cases:

handler = case

#如果满足该类情况

**if** handler.test(self):

#调用相应的act函数

handler.act(self)

**break**

# 处理异常

**except** Exception **as** msg:

self.handle\_error(msg)

**def** **handle\_file**(self, full\_path):

**try**:

**with** open(full\_path, 'rb') **as** reader:

content = reader.read()

self.send\_content(content)

**except** IOError **as** msg:

msg = "'{0}' cannot be read: {1}".format(self.path, msg)

self.handle\_error(msg)

**def** **handle\_error**(self, msg):

content = self.Error\_Page.format(path=self.path, msg=msg)

self.send\_content(content, 404)

**def** **create\_page**(self):

values = {

'date\_time' : self.date\_time\_string(),

'client\_host' : self.client\_address[0],

'client\_port' : self.client\_address[1],

'command' : self.command,

'path' : self.path

}

page = self.Page.format(\*\*values)

**return** page

**def** **send\_content**(self, content, status=200):

self.send\_response(status)

self.send\_header("Content-type", "text/html")

self.send\_header("Content-Length", str(len(content)))

self.end\_headers()

self.wfile.write(content)

#----------------------------------------------------------------------

**if** \_\_name\_\_ == '\_\_main\_\_':

serverAddress = ('', 8080)

server = BaseHTTPServer.HTTPServer(serverAddress, RequestHandler)

server.serve\_forever()

在工作目录下添加index.html文件

**<html>**

**<head>**

**<title>**Index Page**</title>**

**</head>**

**<body>**

**<h1>**Index Page**</h1>**

**<p>**Welcome to my home.**</p>**

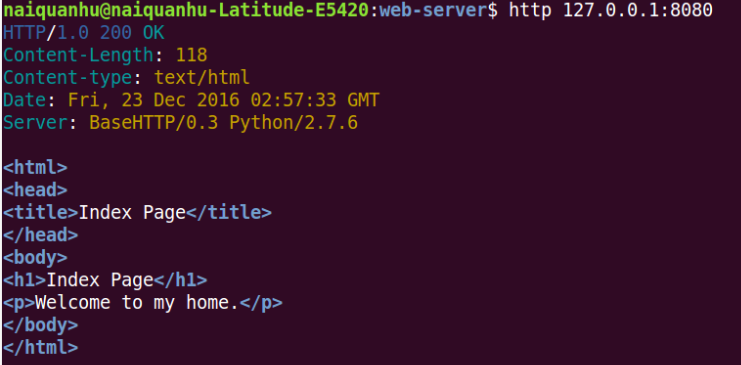
**</body>**

**</html>**

测试一下

$ http 127.0.0.1:8080

效果图



### CGI协议

当然，大部分人都不希望每次给服务器加新功能都要到服务器的源代码里进行修改。如果程序能独立在另一个脚本文件里运行那就再好不过了。[什么是CGI？](http://www.biaodianfu.com/cgi-fastcgi-wsgi.html) 本小节会实现CGI的效果。

接下来的例子中，我们会在html页面上显示当地时间。

创建新文件 time.py

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

**from** datetime **import** datetime

**print** '''\

<html>

<body>

<p>Generated {0}</p>

</body>

</html>'''.format(datetime.now())

新建一个处理脚本文件的条件类：

**class** **case\_cgi\_file**(object):

'''脚本文件处理'''

**def** **test**(self, handler):

**return** os.path.isfile(handler.full\_path) **and** \

handler.full\_path.endswith('.py')

**def** **act**(self, handler):

##运行脚本文件

handler.run\_cgi(handler.full\_path)

实现运行脚本文件的函数

**def** **run\_cgi**(self, full\_path):

data = subprocess.check\_output(["python", full\_path])

self.send\_content(data)

不要忘了加到Cases中去

Cases = [case\_no\_file(),

case\_cgi\_file(),

case\_existing\_file(),

case\_directory\_index\_file(),

case\_always\_fail()]

整个代码：

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

**import** sys, os, BaseHTTPServer

**import** subprocess

**class** **ServerException**(Exception):

'''服务器内部错误'''

**pass**

**class** **case\_no\_file**(object):

'''该路径不存在'''

**def** **test**(self, handler):

**return** **not** os.path.exists(handler.full\_path)

**def** **act**(self, handler):

**raise** ServerException("'{0}' not found".format(handler.path))

**class** **case\_existing\_file**(object):

'''该路径是文件'''

**def** **test**(self, handler):

**return** os.path.isfile(handler.full\_path)

**def** **act**(self, handler):

handler.handle\_file(handler.full\_path)

**class** **case\_always\_fail**(object):

'''所有情况都不符合时的默认处理类'''

**def** **test**(self, handler):

**return** **True**

**def** **act**(self, handler):

**raise** ServerException("Unknown object '{0}'".format(handler.path))

**class** **case\_directory\_index\_file**(object):

**def** **index\_path**(self, handler):

**return** os.path.join(handler.full\_path, 'index.html')

#判断目标路径是否是目录&&目录下是否有index.html

**def** **test**(self, handler):

**return** os.path.isdir(handler.full\_path) **and** os.path.isfile(self.index\_path(handler))

#响应index.html的内容

**def** **act**(self, handler):

handler.handle\_file(self.index\_path(handler))

**class** **case\_cgi\_file**(object):

'''脚本文件处理'''

**def** **test**(self, handler):

**return** os.path.isfile(handler.full\_path) **and** \

handler.full\_path.endswith('.py')

**def** **act**(self, handler):

##运行脚本文件

handler.run\_cgi(handler.full\_path)

**class** **RequestHandler**(BaseHTTPServer.BaseHTTPRequestHandler):

'''处理请求并返回页面'''

# ...页面模板...

Page = '''\

<html>

<body>

<table>

<tr> <td>Header</td> <td>Value</td> </tr>

<tr> <td>Date and time</td> <td>{date\_time}</td> </tr>

<tr> <td>Client host</td> <td>{client\_host}</td> </tr>

<tr> <td>Client port</td> <td>{client\_port}</td> </tr>

<tr> <td>Command</td> <td>{command}</td> </tr>

<tr> <td>Path</td> <td>{path}</td> </tr>

</table>

</body>

</html>

'''

Error\_Page = """\

<html>

<body>

<h1>Error accessing {path}</h1>

<p>{msg}</p>

</body>

</html>

"""

# 所有可能的情况

Cases = [case\_no\_file(),

case\_cgi\_file(),

case\_existing\_file(),

case\_directory\_index\_file(),

case\_always\_fail()]

# 处理一个GET请求

**def** **do\_GET**(self):

**try**:

# 文件完整路径

self.full\_path = os.getcwd() + self.path#注意，一定要给self.full\_path赋值

#遍历所有可能的情况

**for** case **in** self.Cases:

handler = case

#如果满足该类情况

**if** handler.test(self):

#调用相应的act函数

handler.act(self)

**break**

# 处理异常

**except** Exception **as** msg:

self.handle\_error(msg)

**def** **handle\_file**(self, full\_path):

**try**:

**with** open(full\_path, 'rb') **as** reader:

content = reader.read()

self.send\_content(content)

**except** IOError **as** msg:

msg = "'{0}' cannot be read: {1}".format(self.path, msg)

self.handle\_error(msg)

**def** **handle\_error**(self, msg):

content = self.Error\_Page.format(path=self.path, msg=msg)

self.send\_content(content, 404)

**def** **create\_page**(self):

values = {

'date\_time' : self.date\_time\_string(),

'client\_host' : self.client\_address[0],

'client\_port' : self.client\_address[1],

'command' : self.command,

'path' : self.path

}

page = self.Page.format(\*\*values)

**return** page

**def** **send\_content**(self, content, status=200):

self.send\_response(status)

self.send\_header("Content-type", "text/html")

self.send\_header("Content-Length", str(len(content)))

self.end\_headers()

self.wfile.write(content)

**def** **run\_cgi**(self, full\_path):

data = subprocess.check\_output(["python", full\_path])

self.send\_content(data)

#----------------------------------------------------------------------

**if** \_\_name\_\_ == '\_\_main\_\_':

serverAddress = ('', 8080)

server = BaseHTTPServer.HTTPServer(serverAddress, RequestHandler)

server.serve\_forever()

查看效果

$ http 127.0.0.1:8080/time.py



### 代码重构

回头看看我们的代码，注意到一个新的问题了吗？虽然条件判断已经被我们整理到几个类中去了，但是像run\_cgi只有在路径为py文件的条件下才使用的函数是放在 RequestHandler下的，那以后再加几个新功能，但是这类函数都放到 RequestHandler下的话可想而知RequestHandler 会变的臃肿不堪。当然你会想这算什么问题嘛，把它放到各自的条件类下不就好了噢。

各自的代码归各自是个好办法，但有时候不同的条件类内可能会有功能相同的函数，这时候我们都知道重复相同的代码是软件开发里很忌讳的一件事情，那么怎么处理重复的代码呢？

可以抽象出一个基类嘛，遇到重复的内容就放在基类的下面，所有的条件类都继承这个基类。

子类继承基类，依此类推进行修改，重构后的全部代码：

#!/usr/bin/env python

#-\*- coding:utf-8 -\*-

**import** sys, os, BaseHTTPServer, subprocess

#-------------------------------------------------------------------------------

**class** **ServerException**(Exception):

'''服务器内部错误'''

**pass**

#-------------------------------------------------------------------------------

**class** **base\_case**(object):

'''条件处理基类'''

**def** **handle\_file**(self, handler, full\_path):

**try**:

**with** open(full\_path, 'rb') **as** reader:

content = reader.read()

handler.send\_content(content)

**except** IOError **as** msg:

msg = "'{0}' cannot be read: {1}".format(full\_path, msg)

handler.handle\_error(msg)

**def** **index\_path**(self, handler):

**return** os.path.join(handler.full\_path, 'index.html')

**def** **test**(self, handler):

**assert** **False**, 'Not implemented.'

**def** **act**(self, handler):

**assert** **False**, 'Not implemented.'

#-------------------------------------------------------------------------------

**class** **case\_no\_file**(base\_case):

'''文件或目录不存在'''

**def** **test**(self, handler):

**return** **not** os.path.exists(handler.full\_path)

**def** **act**(self, handler):

**raise** ServerException("'{0}' not found".format(handler.path))

#-------------------------------------------------------------------------------

**class** **case\_existing\_file**(base\_case):

'''文件存在的情况'''

**def** **test**(self, handler):

**return** os.path.isfile(handler.full\_path)

**def** **act**(self, handler):

self.handle\_file(handler, handler.full\_path)

#-------------------------------------------------------------------------------

**class** **case\_always\_fail**(base\_case):

'''所有情况都不符合时的默认处理类'''

**def** **test**(self, handler):

**return** **True**

**def** **act**(self, handler):

**raise** ServerException("Unknown object '{0}'".format(handler.path))

#-------------------------------------------------------------------------------

**class** **case\_directory\_index\_file**(base\_case):

'''在根路径下返回主页文件'''

#判断目标路径是否是目录&&目录下是否有index.html

**def** **test**(self, handler):

**return** os.path.isdir(handler.full\_path) **and** os.path.isfile(self.index\_path(handler))

#响应index.html的内容

**def** **act**(self, handler):

self.handle\_file(handler, self.index\_path(handler))

#-------------------------------------------------------------------------------

**class** **case\_cgi\_file**(base\_case):

'''脚本文件处理'''

**def** **run\_cgi**(self, handler):

data = subprocess.check\_output(["python", handler.full\_path])

handler.send\_content(data)

**def** **test**(self, handler):

**return** os.path.isfile(handler.full\_path) **and** \

handler.full\_path.endswith('.py')

**def** **act**(self, handler):

#运行脚本文件

self.run\_cgi(handler)

#-------------------------------------------------------------------------------

**class** **RequestHandler**(BaseHTTPServer.BaseHTTPRequestHandler):

'''

请求路径合法则返回相应处理

否则返回错误页面

'''

Error\_Page = """\

<html>

<body>

<h1>Error accessing {path}</h1>

<p>{msg}</p>

</body>

</html>

"""

# 所有可能的情况

Cases = [case\_no\_file(),

case\_cgi\_file(),

case\_existing\_file(),

case\_directory\_index\_file(),

case\_always\_fail()]

# 处理一个GET请求

**def** **do\_GET**(self):

**try**:

# 文件完整路径

self.full\_path = os.getcwd() + self.path#注意，一定要给self.full\_path赋值

#遍历所有可能的情况

**for** case **in** self.Cases:

#如果满足该类情况

**if** case.test(self):

#调用相应的act函数

case.act(self)

**break**

# 处理异常

**except** Exception **as** msg:

self.handle\_error(msg)

**def** **handle\_error**(self, msg):

content = self.Error\_Page.format(path=self.path, msg=msg)

self.send\_content(content, 404)

**def** **send\_content**(self, content, status=200):

self.send\_response(status)

self.send\_header("Content-type", "text/html")

self.send\_header("Content-Length", str(len(content)))

self.end\_headers()

self.wfile.write(content)

#----------------------------------------------------------------------

**if** \_\_name\_\_ == '\_\_main\_\_':

serverAddress = ('', 8080)

server = BaseHTTPServer.HTTPServer(serverAddress, RequestHandler)

server.serve\_forever()

通过重构我们发现，真正实施行为(Action)的代码逻辑可以抽出来进行封装(封装成各种条件处理类)，而 RequestHandler类 或是 basecase类 提供了供条件处理类使用的接口，它们可以看作是一系列服务(Service)，在软件设计中我们常常会把业务代码进行分层，将行为与服务分开，降低耦合，更有利于我们开发维护代码。

通过统一接口，以及cgi程序，我们的代码功能扩展变的更加容易，可以专心于编写功能代码，而不用去关心其他部分。case 的添加虽然仍在server代码中，但我们也可以把它放到配置文件中，由server读取配置文件。

## Python自动填问卷星

本课程讲授如何使用 Python 自动填写表单，学习使用 tshark 抓包，根据 HTTP 协议分析浏览器是怎样将数据提交到服务器的，然后使用 urllib 和 urllib2 以相同的方式把表单数据发送到服务器。

### 抓包

首先安装 tshark

$ sudo apt-get install tshark

## 新浪微博爬虫