Contents

[Contents 1](#_Toc482099493)

[<最佳实践> 3](#_Toc482099494)

[模块与包 3](#_Toc482099495)

[基本概念 4](#_Toc482099496)

[字符串: 5](#_Toc482099497)

[列表 （可变，异构，原地修改） 5](#_Toc482099498)

[元组（不可变，常用于函数或方法返回多个值打包成元组） 6](#_Toc482099499)

[字典（键不可变，如整形、浮点型、字符串、元组） 6](#_Toc482099500)

[集合 7](#_Toc482099501)

[生成器和迭代器 7](#_Toc482099502)

[文件 8](#_Toc482099503)

[面向函数编程 8](#_Toc482099504)

[函数 8](#_Toc482099505)

[装饰器 8](#_Toc482099506)

[函数curry化 9](#_Toc482099507)

[filter, map and reduce 9](#_Toc482099508)

[面向对象编程 10](#_Toc482099509)

[技巧 11](#_Toc482099510)

[序列解包和字典解包 11](#_Toc482099511)

[性能优化 11](#_Toc482099512)

[Python包使用 12](#_Toc482099513)

[包管理器pip 12](#_Toc482099514)

[模块查询 13](#_Toc482099515)

[virtual environment 13](#_Toc482099516)

[Requirements files 14](#_Toc482099517)

[Python脚本编程和运行方式 15](#_Toc482099518)

[Interactive Mode Programming 15](#_Toc482099519)

[Script Mode Programming 15](#_Toc482099520)

[like shell 15](#_Toc482099521)

[语法 15](#_Toc482099522)

[Standard Data Types 15](#_Toc482099523)

[Types of Operator 15](#_Toc482099524)

[函数 16](#_Toc482099525)

[模块 17](#_Toc482099526)

[Packages in Python 18](#_Toc482099527)

[输入输出 18](#_Toc482099528)

[惯用法 20](#_Toc482099529)

[函数式编程 20](#_Toc482099530)

[常用模块 21](#_Toc482099531)

[$import copy, sys, os, glob, tempfile, shutil, base64, logging, smptplib, optparse 21](#_Toc482099532)

[日期和时间 23](#_Toc482099533)

[数学 24](#_Toc482099534)

[迭代器函数 25](#_Toc482099535)

[$import re 26](#_Toc482099536)

[文件解析 27](#_Toc482099537)

[import pprint 27](#_Toc482099538)

[copy and paste 28](#_Toc482099539)

[$import json 28](#_Toc482099540)

[$import csv 28](#_Toc482099541)

[Beautiful Soup --HTML and XML 解析 29](#_Toc482099542)

[PDF解析 29](#_Toc482099543)

[Excel解析 30](#_Toc482099544)

[Word解析 31](#_Toc482099545)

[多线程，定时任务和GUI自动化 31](#_Toc482099546)

[多线程 31](#_Toc482099547)

[启动进程 32](#_Toc482099548)

[定时任务 32](#_Toc482099549)

[GUI自动化 33](#_Toc482099550)

[数据库 34](#_Toc482099551)

[Sqlite3 34](#_Toc482099552)

[网络数据采集 35](#_Toc482099553)

[网络请求和响应 --requests 35](#_Toc482099554)

[解析HTML and XML --Beautiful Soup 37](#_Toc482099555)

[模拟浏览器操作 40](#_Toc482099556)

[web framework 41](#_Toc482099557)

[build-in web server to setup a web server 41](#_Toc482099558)

[Flask 41](#_Toc482099559)

[表单自动填写 41](#_Toc482099560)

[<python基础教程> 42](#_Toc482099561)

[模块 42](#_Toc482099562)

[包 43](#_Toc482099563)

[python基本类型 43](#_Toc482099564)

[序列 43](#_Toc482099565)

[通用操作 43](#_Toc482099566)

[列表 44](#_Toc482099567)

[元组 44](#_Toc482099568)

[字符串 44](#_Toc482099569)

[集合:通过列表或元组定义 44](#_Toc482099570)

[字典 45](#_Toc482099571)

[常用操作 45](#_Toc482099572)

[不常用的命令： 46](#_Toc482099573)

[性能比较： 46](#_Toc482099574)

[函数 46](#_Toc482099575)

[异常 47](#_Toc482099576)

[类 47](#_Toc482099577)

[class 继承 48](#_Toc482099578)

[迭代器 49](#_Toc482099579)

[生成器 50](#_Toc482099580)

[文件 51](#_Toc482099581)

[<Python in Practice> 51](#_Toc482099582)

[1. sequence unpacking and map unpacking 51](#_Toc482099583)

[2. locals() & glocal() 52](#_Toc482099584)

[3. @classmethod and @staticmethod的区别: 52](#_Toc482099585)

[5. \_\_slots\_\_ 53](#_Toc482099586)

[6. \_\_init\_\_, \_\_new\_\_, and \_\_call\_\_区别 54](#_Toc482099587)

[7. Exec and eval 54](#_Toc482099588)

[8. Type使用 54](#_Toc482099589)

[9. 面向函数编程：提供的函数 54](#_Toc482099590)

[10. \_\_call\_\_(self) 55](#_Toc482099591)

[<Python编程快速上手—让繁琐工作自动化> 56](#_Toc482099592)

[基础知识 56](#_Toc482099593)

[<Flask Web开发> 57](#_Toc482099594)

[tutorial 57](#_Toc482099595)

[视图 59](#_Toc482099596)

[模板 59](#_Toc482099597)

[集成Twitter Bootstrap 60](#_Toc482099598)

[自定义错误页面 61](#_Toc482099599)

[链接 61](#_Toc482099600)

[静态文件 61](#_Toc482099601)

[使用Flask-Moment本地化日期和时间 62](#_Toc482099602)

[Web表单 62](#_Toc482099603)

[Flash消息 （相当于toast） 63](#_Toc482099604)

[数据库 63](#_Toc482099605)

[电子邮件 65](#_Toc482099606)

[大型程序结构 66](#_Toc482099607)

# <最佳实践>

## 模块与包

模块放在可以被解释器找到的路径中

import sys, pprint

**pprint.pprint(sys.path) #查看Python解释器的搜索目录**

**help(‘模块’) #查看模块的帮助信息** help(‘模块.函数’) #查看模块内函数的具体信息

dir(‘模块’) #列出模块的类，函数，变量等

help(对象) #查看对象对应的类的API

**if \_\_name\_\_ == '\_\_main\_\_': 模块main入口**

包是文件夹

drawing/ #包目录

\_\_init\_\_.py #包代码(drawing模块)

colors.py #包中colors模块

如何引用？

import drawing #import the drawing package

import drawing.colors #import the colors module,使用时需用drawing.colors.fun()

from drawing import shapes #import the shapes model, 使用时只需用shapes.fun()

包管理器pip

**$apt-get -y install python-pip**

$pip --help

$pip list #列出已安装包 $pip install #安装包 $pip install --upgrade #升级包

$pip uninstall 卸载包 $pip show显示包详细信息 $pip search 搜索包

虚拟环境

**$sudo pip install virtualenv**

$ cd ~/code/myproject/

$ virtualenv env

**$source env/bin/activate 激活虚拟环境**

(env)$ pip install flask 在虚拟环境下工作

**(env)$ deactivate 退出虚拟环境**

(env)$pip freeze > requirements.txt # output installed packages in requirements format

(env)$pip install -r requirements.txt # 安装清单中的包

常用模块

NLTK(文本处理), Pandas(数据处理), sys(系统环境), os+shutil+tempfile+glob(文件和目录), base64, logging, smptplib, time+datetime+calendar, decimal+fractions+math+random, re, threading+subprocess, json+csv+pypdf2+openpyxl+python-docs (文件), requests+beautifulsoup4

## 基本概念

条件测试

false none 0 “” () [] {} set()被解释为假.

“==”判定同一性(别名)，”is”判定相等性(值相等)

all([True, 1, {3}]) 每个元素都为真时

any([True, 0, []) 至少一个元素为真时

None是NoneType数据类型的唯一值（其他编程语言称为null, nil或undefined

在变量必须保存可变数据类型的值时，例如列表或字典，Python使用引用。对于不可变的数据类型的值，例如字符串、整形或元组，Python变量就保存值本身

|  |  |  |
| --- | --- | --- |
| argument is being passed by reference  the change reflects back in the calling function | | |
| def changeme( mylist ):  mylist.append([1,2,3,4]);  print "Values inside the function: ", mylist  return | mylist = [10,20,30];  changeme( mylist );  print "Values outside the function: ", mylist | Values inside the function: [10, 20, 30, [1, 2, 3, 4]]  Values outside the function: [10, 20, 30, [1, 2, 3, 4]] |
| the reference is being overwritten inside the called function | | |
| def changeme( mylist ):  # This would assig new reference in mylist  mylist = [1,2,3,4];  print "Values inside the function: ", mylist  return | mylist = [10,20,30];  changeme( mylist );  print "Values outside the function: ", mylist | Values inside the function: [1, 2, 3, 4]  Values outside the function: [10, 20, 30] |

### 字符串:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 单引号 | 双引号 | 跨多行-三引号 | 跨多行 | 原始字符串 | unicode字符串 |
| ‘hello world’ | ”hello world” | ’’’hello  Qzlin’’’ | ‘Hello \  Qzlin’ | r’c:\Program Files\fnord’ | u’林其忠’ |

格式化

u'&titles=Category:{0}'.format(category)

方法：find, join, lower, replace, split, strip, translate,

’/’.join((‘’, ‘usr’, ‘bin’, ‘env’)) => ‘/usr/bin/env’

### 列表 （可变，异构，原地修改）

索引: arr[i]

切片: arr[0:4]

连接(记住：同类型才能相加): [1, 2, 3] + [4]

复制多次: ‘python’\*5, [42]\*10

初始值: []空列表, [0]\*10长度为10，值为0的列表，[none]\*10预留长度为10，没有值的列表

成员资格: ‘low’ in arr

删除元素：arr[i:j] = [] or del arr[i]

对象的方法: append, count, extend, index, insert, pop, remove, reverse, sort,

for (i, v) in enumerate(arr): #遍历

values[i] = 2\*v ;

聚合函数：len, min, max, sum

**列表推导式** （利用其他列表创建新列表，字典或集合）

list1 = [(x,y) for x in range(3) for y in range(3)]

**list2 = [x\*x for x in range(10) if x%3==0]**

**squre\_dict = {x: x\*x for x in range(5)} #{0:0, 1:1, 2:4, 3:9, 4:16}**

**square\_set = {x\*x for x in [1, -1]} #{ 1 }**

**square\_tuple = tuple(x\*x for x in range(5))**

x = [1, 2, 3]

x.extend([4, 5, 6]) == [1, 2, 3, 4, 5, 6]

x.append(7) == [1, 2, 3, 4, 5, 6, 7]

### 元组（不可变，常用于函数或方法返回多个值打包成元组）

def sum\_and\_product(x, y):

return (x+y), (x\*y)

(value\_sum, value\_product) = sum\_and\_product(x, y)

支持操作：in, not in, 比较串联切片和索引, min(), max()

(1, 2, 3) + (4, 5, 6) = (1, 2, 3, 4, 5, 6)

为了提高程序的可读性，可以给元组的每个元素取一个名字，并通过名字来访问这些元素

from collections import namedtuple

vector = namedtuple(“Dimension”, “x y z”)

voel = vector(1, 2, 3)

voel.x, voel.y, voel.z

### 字典（键不可变，如整形、浮点型、字符串、元组）

x = {}

x[42] = ‘ff’ #字典可以，自动添加

注意：

x = []

x[42] = ‘ff’ #出错，列表不可以，需要预留空间x = [none] \* 43

方法：clear, copy, fromkeys, get, has\_key, items, iteritems, keys, iterkeys, popitem, setdefault, update,values, itervalues

for (key, value) in scientists.iteritems(): #遍历

…

for k in dict.keys():

for v in dict.values():

for k, v in dict.items():

key in dict #检查字典是否含键

dict.get(key, defaultValue) 获取键值，若不存在则返回初始值

dict.setdefault(key, defaultValue) 检查键，若不存在则设置初始值

一个defaultdict相当于标准的字典，除了当你查找一个没有包含在内的键时，它用一个你提供的零参数函数建立一个新的键

from collections import defaultdict

word\_counts = defaultdict(int) #int()生成0

word\_counts[word] += 1

dd\_list = defaultdict(list) #list()生成一个空列表

dd\_list[2].append(1) #dd\_list = {2: [1]}

dd\_dcit = defaultdict(dict) #dict()产生一个新字典

dd\_dict["Joel"]["City"] = "Seattle"

#结果：

dd\_list = {

"Joel": {

"City": "Seattle"

}}

### 集合

item\_set = set(item\_list)

item in item\_set

cmn\_set = set1.intersection(set2)

unq\_set = set1.union(set2)

### 生成器和迭代器

元组推导得到生成器generator： 迭代按需延迟产生

SimpleCounter = (x\*\*2 for x in range(10\*\*10)

for val in SimpleCounter:

…

yeild语句创建生成器

def lazy\_range(n):

i = 0

while i < n :

yield i

i += 1

实际范例：

**range(10)产生[0, ... , 9]列表**

**xrange(10) 产生[0, ..., 9]生成器**

zip(range(10), range(10)　产生元组列表[(0, 0), (1, 1), …, (9, 9)]

from itertools import izip 　元组生成器

for val in izip(range(10), range(10):

…

from itertools import combinations

a = [1, 2, 3, 4]

aa = combinations(a, 2) 所有(a[i], a[j])的组合，即[(1,2), (1,3), ...(3,4)]

### 文件

#文件内容如果可以很小，一次性读取所有内容:

f = open(filename)

for line in f.readlines():

process(line)

f.close()

#文件内容很大，流式处理,一行一行读取

import fileinput

for line in fileinput.input(filename):

process(line)

**在python里，一个文件对象就是一个迭代器，每次只处理一行数据，而不是将全部文件加载到内存**

with open("data.txt’") as file:

for line in file:

print len(line.strip())

## 面向函数编程

### 函数

python函数是一等公民，可以赋给变量，或函数参数

函数可以有默认参数

def my\_print(message="my default message"):

print message

### 装饰器

使用装饰器改变函数行为　(AOP编程）

关键点：函数作为参数，函数内嵌函数，@修饰表示函数执行之前，插入AOP

import time

def log\_wrapper(func):

def wrapper(intext, \*args, \*\*kwargs):

print("logging: {0}({1})".format(func.\_\_name\_\_, intext))

return func(intext)

return wrapper

@log\_wrapper

def process(text):

print text

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

process("qzlin")

### 函数curry化

from functools import partial

def foo(a,b,c):

return a + b + c

foo23 = partial(foo, b=23)

foo23(a = 1, c = 3) # => 27

def fun\_wapper(cases, function):

def norm(y):

y\_max, y\_min = np.max(y), np.min(y)

y = (y - y\_min) / (y\_max - y\_min)

return y

y = function(cases)

y = norm(y)

return y

cases = np.array([[val] for val in np.linspace(0, 2\*np.pi, 30)])

partial(fun\_wapper, function=lambda cases: np.sin(cases)+np.cos(cases))(cases)

### filter, map and reduce

filter(function, sequence)

对sequence中的item依次执行function(item)，将执行结果为True的item组成一个List/String/Tuple（取决于sequence的类型）返回

如：filter(lambda x: x!='a', 'abcdef')

map(function, sequence)

对sequence中的item依次执行function(item)，将执行结果组成一个List返回

如：map(lambda x: x+x, 'abcdef')

另外map也支持多个sequence，这就要求function也支持相应数量的参数输入

如：map(lambda x,y: x+y, range(8), range(8))

map(math.pow, [10, 20, 30], [1, 2, 3])

reduce(function, sequence, starting\_value)

对sequence中的item顺序迭代调用function，如果有starting\_value，还可以作为初始值调用

如：reduce(lambda x,y: x+y, range(1,100))

employees = [(‘joe’, 1, 53), (‘qzlin’, 2, 34), … ]

sorted(employees, key=lambda emp: emp[0]);

## 面向对象编程

class Organism(object):

#所有数据通过构造器

def \_\_init\_\_(self, name, x, y):

self.name = name

self.x = x

self.y = y

self.atoms = []

#类似于Java toString()

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

return ‘(%s, [%s, %s])’ % self.name, self.x, self.y)

#公有方法

def move(self):

…

#私有方法

def \_\_inaccessible(self):

…

#静态方法

@staticmethod

def smeth():

…

python并没有真正的私有化支持，解释器会对上面翻译成

Organism. move(self)

Organism.\_Organism\_\_inaccessible(self)

所以在实际中，

o = Organism()

b.move() #公有方法访问

b.\_Organism\_\_inaccessible() #私有方法访问

# 继承于Organism

class Arthropod(Organism):

def \_\_init\_\_(self, name, x, y, legs):

#老方法

Organism.\_\_init\_\_(self, name, x, y)

#新方法

super(Arthropod, self).\_\_init\_\_(name, x, y)

self.legs = legs

#以下子类与父类同名函数，实现多态性

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

…

def move(self):

..

## 技巧

### 序列解包和字典解包

first, second, \*rest = sequence # 序列解包，可以用来传递任意数量的位置参数

等同于first =sequence[0], second=sequence[1], rest=sequence[2:]

args = (600, 900)

kwargs = dict(copies=2, collate=False)

def print\_args(**\*args**, \*\*kwargs): 序列解包和映射解包

**\*args 是传入位置参数的tuple**

**\*\*kwargs是传入关键字参数的dict**

**for arg in args:**

**…**

**for key in kwargs:**

**kwargs[key]**

locals()：囊括了函数执行到该时间点时所定义的一切变量, 返回的字典对所有局部变量的名称与值进行映射.常配合mapping unpacking，用来实例化字符串模板

str.format(**\*\*locals()**). #映射解包，可以用来传递任意数量的关键字参数

字符串模板形参需要的实参从locals()返回的dict传入

### 性能优化

$pip install numba

from numba import jit

@jit

def scale\_image(imageData, newRange = (0, 255)):

slice, height, width = len(imageData), len(imageData[0]), len(imageData[0][0])

seq = np.zeros((slice, height, width))

(min, max) = get\_range(imageData)

for z in xrange(slice):

for y in xrange(height):

for x in xrange(width):

val = imageData[z, y, x]

　　nMin = newRange[0]

nMax = newRange[1]

newVal = (val - min) \* (nMax - nMin) / (max - min) + nMin

seq[z, y, x] = newVal

return seq

ts = time.clock()

traverseImageData(imageData)

print time.clock() - ts, "seconds process time"

$pip install joblib

from joblib import Parallel, delayed

def find\_amicable\_numbers(low, high):

for i in xrange(low, high):

for j in xrange(i + 1, high):

i\_factors\_sum = calc\_primefactors\_sum(i)

j\_factors\_sum = calc\_primefactors\_sum(j)

if i\_factors\_sum == j and i == j\_factors\_sum:

print (i, j)

a = np.arange(1, 10\*\*5+2, 1000)

Parallel(n\_jobs=-1)(delayed(find\_amicable\_numbers)(i, j) for (i, j) in zip(a[:-1], a[1:]))

# Python包使用

## 包管理器pip

pip类似RedHat的yum, Ubuntu的apt-get, Ruby的gem

PyPI is a repository for open-source third-party Python packages. It's similar to RubyGems in the Ruby world, PHP's Packagist, CPAN for Perl, and NPM for Node.js.

Install easy\_install

$ sudo apt-get install python-setuptools python-dev build-essential

$ sudo easy\_install pip

or

**$apt-get -y install python-pip**

Python has primitive package manager called easy\_install, which is installed automatically when you install Python itself

$pip --help

$pip list 列出已安装包

$pip install 安装包

$pip install --upgrade 升级包

$pip uninstall 卸载包

$pip show 显示包详细信息

$pip search 搜索包，类似yum里的search

注：若需要代理$pip install **--proxy=http://165.225.96.34:10015/** numpy

## 模块查询

help(‘模块’) 查看模块的帮助信息

**dir(‘模块’) 列出模块的类，函数，变量等**

help(‘模块.函数’) 查看模块内函数的具体信息

## virtual environment

virtualenv is a tool for creating isolated Python environments containing their own copy of python, pip, and their own place to keep libraries installed from PyPI.

simply a directory that contains a complete copy of everything needed to run a Python program, including a copy of the python binary itself, a copy of the entire Python standard library, a copy of the pip installer, and (crucially) a copy of the site-packages directory mentioned above.

$sudo **pip install virtualenv**

若有代理：

$export https\_proxy=http://web-proxy.mydomain.com

$sudo -E pip install virtualenv

Usually pip and virtualenv are the only two packages you ever need to install globally.In fact, virtualenv comes with a copy of pip which gets copied into every new environment you create, so virtualenv is really all you need.

$ cd ~/code/myproject/

$ **virtualenv env**

Start by changing directory into the root of your project directory, and then use the virtualenv command-line tool to create a new environment

It's a common convention to call this directory env, and to put it inside your project directory (so, say you keep your code at ~/code/projectname/, the environment will be at ~/code/projectname/env/ - each project gets its own env).

Note: if you're using a version control system like git, you shouldn't commit the env directory. Add it to your .gitignore file (or similar).

projectname

env

bin

include

lib

helloworld.py

helloworld.py

import requests

print requests.get('http://www.baidu.com')

$env/bin/pip install requests

$env/bin/python helloworld.py

可以简约如下：

$**source env/bin/activate** 激活虚拟环境

虚拟环境Python解释器的路径就被添加进PATH中，但这种改变不是永久性的，它只会影响当前的命令行会话

(env)$ pip install flask

(env)$ python

>>> ...正常python命令行操作

(env)$ **deactivate** 退出虚拟环境

virtualenv默认是采用python2解析器，所以以上生成python2环境

若要生成python3环境呢？

1. pip install --upgrade virtualenv
2. virtual -p python3 env 本地目录下生成env, 即python3环境
3. source env/bin/activate
4. (env)$pip install -r requirements.txt

## Requirements files

Requirements files give you a way to create an environment: a set of packages that work together.

requirements make explicit, repeatable installation of packages.

requirements.txt

MyApp

Framework==0.9.4

Library>=0.2

$pip install -r requirements.txt

Output installed packages in requirements format

$pip freeze > requirements.txt

## Python脚本编程和运行方式

### Interactive Mode Programming

$ python

>>> print "Hello, Python!"

>>> exit()

### Script Mode Programming

test.py

print "Hello, Python"

$ python test.py

### like shell

#!/usr/bin/python

print "Hello, Python!"

$ chmod +x test.py # This is to make file executable

$./test.py

## 语法

### Standard Data Types

Numbers

String

List

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

Tuple

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated

Dictionary

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

### Types of Operator

Arithmetic Operators： +， -， \*， 、， %， \*\*， //

Comparison (Relational) Operators： ==, !=, <>, >, <, >=, <=

Assignment Operators： =, +=, -=, \*=, /=, %=, \*\*=, //=

Logical Operators (Bitwise Operators)： &, |, ^, ~, <<, >>

Membership Operators： in, not in

Identity Operators： is, is not

### 函数

#### 函数定义

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

sum = lambda arg1, arg2: arg1 + arg2;

#### 函数传参

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

def changeme( mylist ):

mylist.append([1,2,3,4]);

print "Values inside the function: ", mylist

return

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

argument is being passed by reference and the reference is being overwritten inside the called function.

def changeme( mylist ):

mylist = [1,2,3,4]; # This would assig new reference in mylist

print "Values inside the function: ", mylist

return

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

#### 函数参数

Required arguments

Keyword arguments：

def printinfo( name, age ):

...

printinfo( age=50, name="miki" )

Default arguments

def printinfo( name, age = 35 ):

...

printinfo( name="miki" )

Variable-length arguments

def printinfo( arg1, \*vartuple ):

...

printinfo( 70, 60, 50 )

结果：arg1=70, vartuple=(60, 50)

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments.

### 模块

#### 模块定义

A module can define functions, classes and variables.

support.py

def print\_func( par ):

print "Hello : ", par

return

#### 模块调用

# Import module support

import support

# Now you can call defined function that module as follows

support.print\_func("Zara")

The from...import Statement

The from...import \* Statement:

#### 模块搜索

1. The current directory.
2. If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
3. If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

### Packages in Python

Phone/Isdn.py file having function Isdn()

Phone/G3.py file having function G3()

Phone/\_\_init\_\_.py

from Pots import Pots

from Isdn import Isdn

from G3 import G3

To make all of your functions available when you've imported Phone, you need to put explicit import statements in \_\_init\_\_.py as above

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()

## 输入输出

#### STDOUT

#!/usr/bin/python

print """Hello, Python!

I am qzlin,

please welcome me"""

$./helloworld.py | grep py

#### STDIN

#!/usr/bin/python

import sys

for line in sys.stdin:

print line

$./helloworld.py | ./stream-read.py

argv

import sys

print 'Then command line arguments are: '

for (i, v) in enumerate(sys.argv):

print i, "\t", v

$./argsv-ex.py arg1 arg2 arg3

1. ./argv-ex.py
2. arg1
3. arg2
4. arg3

#### 文件迭代器

当文件对象用于for loop时，Python会自动调用readline方法

#文件内容如果可以很小，一次性读取所有内容:

f = open(filename)

for line in f.readlines():

process(line)

f.close()

#文件内容很大，流式处理,一行一行读取

import fileinput

for line in fileinput.input(filename):

process(line)

漂亮打印python数据结构

$import pprint

pprint.pprint(sys.path) 用来打印python数据结构

格式化json

json输出 | python -m json.tool

## 惯用法

’/’.join((‘’, ‘usr’, ‘bin’, ‘env’)) => ‘/usr/bin/env’

list2 = [x\*x for x in range(10) if x%3==0]

list1 = [(x,y) for x in range(3) for y in range(3)]

sequence unpacking

**first, second, \*rest = sequence**

等同于first =sequence[0], second=sequence[1], rest=sequence[2:];

Sequence unpacking可以用来传递任意数量的位置参数

map unpacking

args = (600, 900)

kwargs = dict(copies=2, collate=False)

**def print\_args(\*args, \*\*kwargs):**

可以用来传递任意数量的关键字参数

args 是传入位置参数的tuple

kwargs是传入关键字参数的dict

**locals() & glocal()**

它囊括了函数执行到该时间点时所定义的一切变量, 返回的字典对所有局部变量的名称与值进行映射.常配合mapping unpacking，用来实例化字符串模板

str.format(\*\*locals()).

字符串模板形参需要的实参从locals()返回的dict传入

集合

遍历

for (i, v) in enumerate(arr): list

…

for (key, value) in scientists.iteritems(): dict

…

for name, age in zip(names, ages): 并行迭代多个序列

…

## 函数式编程

filter(function, sequence)

map(function, sequence)

reduce(function, sequence, starting\_value)

filter(lambda x: x%2!=0 and x%3!=0, range(2, 25))

map(lambda x: x\*\*2, range(10))

reduce(lambda x,y: x+y, range(1, 100))

## 常用模块

高级文本处理NLTK程序包

大量数值型数据，Pandas程序包

### $import copy, sys, os, glob, tempfile, shutil, base64, logging, smptplib, optparse

**$import copy**

copy.copy()复制列表或字典

copy.deepcopy()复制列表 （含嵌套列表）

**$import sys**

处理Python运行时配置以及资源，从而可以与前当程序之外的系统环境交互

sys.stdin 标准输入流；sys.stdout 标准输出流；sys.stderr 标准错误流。

sys.path 查找模块所在目录的目录名列表。

sys.argv 命令行的参数列表，第一项脚本名称，第二项第一个命令行参数

sys.platform 返回当前系统平台，如：win32、Linux等。

$import os

Python的系统编程的操作模块，可以处理文件和目录

os.sep 更改操作系统中的路径分隔符。

os.getcwd()获取当前路径，这个在Python代码中比较常用。

os.listdir() 列出当前目录下的所有文件和文件夹。

os.remove() 方法可以删除指定的文件。

os.system() 方法用来运行shell命令。

os.chdir() 改变当前目录，到指定目录中。

os.makedirs() 创建新文件夹

**$import os.path**

os.path：多平台上处理文件

os.path.split('/one/two/three') 解析路径

parts = ('/', 'one', 'two', 'three')

os.path.join(\*parts) /one/two/three

os.path.isfile(file)

os.path.isdir(file)

os.path.exists(file)

**import glob**

glob.glob('dir/\*') 匹配目录dir中的所有路径名（文件或目录）

glob.glob('dir/file?.txt')

glob.glob('dir/\*[0-9].\*') 字符区间

**import tempfile**

tempfile模块提供了多个函数来安全地创建临时文件系统资源

临时文件系统对象，文件关闭时会自动删除

TemporaryFile() 打开并返回一个未命名的文件

temp = tempfile.TemporaryFile()

NamedTemporaryFile()打开并返回一个命名文件

with tempfile.NamedTemporaryFile() as temp:

mkdtemp()创建一个临时目录，并返回其目录名

dir\_name = tempfile.mkdtemp()

tempfile.gettempdir() 获取临时文件位置

查找过程：

环境变量TMPDIR -> 环境变量TEMP -> 环境变量TMP -> window(C:\TMP, ...), Linux(/tmp, /var/tmp, usr/tmp) -> 当前工作目录

高级文件操作

**import shutil**

shutil.copyfile('main.py', 'main.py.copy')

shutil.copy('main.py', 'script') 类似于unix的cp,若目标是目录，则copy文件于该目录下

shutil.move('main.py', 'main.txt') 类似于unix的mv,改名或移动文件

shutil.copytree(src, dst) 复制整个文件夹

shutil.rmtree(path) 删除整个文件夹

import send2trash

send2trash.send2trash('bacon.txt') 删除至回收站

**$import base64**

info = “qizhonglin”

info\_b64encode = base64.b64encode(info)

info\_copy = base64.b64decode(info\_b64encode )

**$import logging**

日志信息保存到一个自己设定格式的文件中

loggin.basicConfig(level=loggin.DEBUG, format=' %(asctime)s - %(levelname)s - %(message)s')

#logging.diable(logging.CRITICAL) 禁用接下来的日志

logging.debug('Start of program')

**$import smptplib**

**$import email.utils**

$from email.mime.text import MIMEText

SMTP（简单邮件传输协议）规定电子邮件应该如何格式化、加密、在邮件服务器之间传递，以及在你点击发送后，计算机要处理的所有其他细节

处理Email，发送、接收、抄送、下载邮件内容等操作

msg = MIMEText('这是消息的主体部分.')

msg['To'] = email.utils.formataddr(('admin', 'admin@example.com'))

msg['From'] = email.utils.formataddr(('Author', 'author@example.com'))

msg['Subject'] = 'Simple test message'

smtpObj = smtplib.SMTP(‘smtp.gmail.com’, 587) 连接到SMTP服务器

try:

smtpObj.ehlo() 向SMTP电子邮件服务器打招呼

smtpObj.starttls() 开始TLS加密

smtpObj.login(‘author@example.com’, ‘password’) 登录到SMTP服务器

smtpObj.sendmail('author@example.com', ['admin@example.com'], msg.as\_string())

finally:

smtpObj.quit()

IMAP（因特网消息访问协议）规定了如何与电子邮件服务提供商的服务器通信，取回发送到你的电子邮件地址的电子邮件

//用IMAP获取和删除电子邮件

import imapclient, pyzmail

imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True) 连接IMAP服务器

imapObj.login('author@gmail.com', 'password') 登录

imapObj.select\_folder('INBOX', readonly=True) 搜索电子邮件

UIDs = imapObj.search(['SINCE 05-Jul-2014']) 执行搜索

rawMessages = imapObj.fetch([UIds], ['BODY[]', 'FLAGS']) 取邮件

//解析邮件内容

message = pyzmail.PyzMessage.factory(rawMessages[UIDs[i]['BODY[]'])

message.get\_subject()

message.get\_addresses('from')

message.get\_addresses('to')

message.get\_addresses('cc')

message.html\_part.get\_payload().decode(message.html\_part.charset)

参数命令行化

parser = optparse.OptionParser()

parser.add\_option('-u', '--url',

dest="url",

default="http://161.92.142.80:8080/mediawiki")

options, remainder = parser.parse\_args()

url = options.url

python main.py -u http://..."

### 日期和时间

$import time

time.time() 返回从纪元以来的秒数

time.ctime() 返回可读的时间

time.clock() 返回处理器时钟时间，用于性能测试、基准测试等

time.gmtime() 以UTC格式返回当前时间

time.localtime() 返回应用了当前时区的当前时间

time.sleep(5) 程序暂停5秒，阻塞直到5秒时间过去，不要一次调用time.sleep(30)来暂停30秒，而是使用for循环执行30次time.sleep(1)调用

for i in range(30):

time.sleep(1)

解析和格式化时间

now = time.ctime()

parsed = time.strptime(now)

print ' tm\_year : ', parsed.tm\_year

time.strftime("%a %b ...", parsed)

$import datetime

t = datetime.time() 时间

t.hour, t.minute, ...

today = datetime.date.today() 日期

today.tm\_year, today.tu\_mon, today.tm\_hour, ...

时间运算

yesterday = today - datetime.timedelta(days=1)

tommorrow = toda + datetime.timedelta(days=1)

today > yesterday

today < tommorrow

$import calendar

c = calendar.TextCalendar(calendar.SUNDAY)

c.prmonth(2011, 7) 返回一个月的格式化文本输出

print c.formatyear(2011, 2, 1, 1, 3) 返回一年的格式化文本输出

### 数学

$import decimal

Decimal类实现定点和浮点运算，构造函数取一个整数或字符串作为参数,重载了算术运算符

decimal.Decimal('3.14') + decimal.Decimal(str(f))

$import fractions

Fraction类基于numbers模块中Rational定义的API，实现了有理数的数值运算,重载了算术运算符

fractions.Fraction(1, 2) ==1/2

fractions.Fraction('1/2')

fractions.Fraction('0.5')

$import math

pring "pi = %.30f" % math.pi 取小数点后30位的值

math.trunc()

math.floor()

math.ceil()

math.modf() 返回tuple,输入值的小数和整数

math.fabs()

math.pow(x, y)

math.log()

math.exp()

math.radians() <-> math.degrees()

math.hypot(deltaX, deltaY)

$import random

生成随机浮点数、整数、字符串，甚至帮助你随机选择列表序列中的一个元素，打乱一组数据等。

random.seed(1)

random() 返回0<=n<1之间的随机实数n；

random.uniform(a, b)设定浮点数的范围，一个是上限，一个是下限。

random.randint(-100, 100)随机生一个整数int类型，可以指定这个整数的范围

random.randrange(0, 101, 5)区间选择值

random.choice(seq) 从序列seq中返回随机的元素；

random.getrandbits(n) 以长整型形式返回n个随机位；

random.shuffle(seq[, random]) 原地指定seq序列；

random.sample(seq, n) 从序列seq中选择n个随机且独立的元素；

非均分分布

random.normalvariate()

random.gauss()

random.lognormvariate()

random.triangular()

random.expovariate()

random.paretovariate()

random.vonmisesvariate()

random.betavariate()

random.gammavariate()

random.weibullvariate()

### 迭代器函数

与使用列表的代码相比，基于迭代器的算法可以提供更好的内存使用特性。在真正需要数据之前，并不从迭代器生成数据，由于这个原因，不需要将所有数据都同时存储在内存中，这种“懒”处理模型可以减少内存使用，相应地还可以减少交换以及大数据集的其他副作用，从而改善性能

$from itertools import \*

for i in chain([1, 2], ['a', 'b']) //chain合并迭代器成列表迭代器

print i

for i in izip([1,2], ['a', 'b']): //izip，合并迭代器成元组迭代器

print i //(1, 'a')

//imap函数返回迭代器，对输入迭代器中的值调用一个函数并返回结果

for i in imap(lambda x,y: (x, y, x\*y), xrange(5), xrange(5, 10):

print '%d \* %d = %d' % i //0 \* 5 = 0

//starmap函数类似imap,不过使用\*语法分解迭代器中的元素作为映射函数的函数

values = [(0, 5), (1, 6)]

for i in starmap(lambda x,y: (x, y, x\*y), values):

print '%d \* %d = %d' % i //0 \* 5 = 0

count(start)返回迭代器，无限地生成连续整数

islice(迭代器，start, end, step)

fro i in islice(count(100), 0, 100, 10):

print i

结果是100, 110, 120, ...

//tee()函数根据一个原输入迭代器返回多个独立的迭代器（默认为两个），

r = islice(count(), 5)

i1, i2 = tee(r)

print 'i1:' list(i1)

print 'i2:' list(i2)

//cycle()

for i, item in izip(xrange(7), cycle(['a', 'b', 'c']):

print (i, item)

item是从['a', 'b', 'c']循环获取

对应str \* 10生成列表[str str str...]

也可以采用迭代器方式

for i in repeat(str, 10):

print i

filter()生成列表

ifilter()生成迭代器

for i in ifilter(filter\_condition, [-1, 2, ...]):

print 'Yielding:', i

groupby()返回迭代器，每个迭代器键-值列表

根据一个属性对相关的值分组

data = [(0, 0), (1, 0), (0, 1), (1, 1)]

for k, g in groupby(data, operator.attrgetter('x')):

print list(g)

结果：

[(0, 0), (0, 1)]

[(1, 0), (1, 1)]

### $import re

1、匹配字符

. 匹配任意除换行符，也就是“\n”以外的任何字符。

\ 转义符，改变原来符号含义，后面会有演示。

[ ] 中括号用来创建一个字符集，第一个出现字符如果是^，表示反向匹配。

2、预定义字符集

\d 匹配数字，如：[0-9]

\D 与上面正好相反，匹配所有非数字字符。

\s 空白字符，如：空格，\t\r\n\f\v等。

\S 非空白字符。

\w 单词字符，如：大写A~Z，小写a~z，数字0~9。

\W 非上面这些字符。

[^abc] 匹配不在这些字符类的所有字符

^Hello 表明匹配必须发生在被查找文本开始处

Hello$ 表明被查找文本必须以这个正则表达式的模式结束

3、可选项与重复子模式

｜ 用管道匹配或关系 re.compile(r’Batman|Tina’)

\* 匹配前一个字符0次或无限次数。

+ 匹配前一个字符1次或无限次数。\*

? 匹配前一个字符0次或1次。

{m} 匹配前一个字符m次。

{m,n} 匹配前一个字符m至n次。

compile(模式字符串) 根据正则表达式字符串，创建模式的对象。

search(待查找字符串) 返回第一个匹配项（Match对象）。

split() 根据模式的匹配项来分割字符串。

findall(待查找字符串) 返回所有匹配项(文本列表,不是Match对象)，若有分组，返回元组的列表

sub(new, 待处理字符串) 用将所有匹配项用new替换掉

escape() 将字符串中所有特殊正则表达式字符转义

import re

phoneRegex = re.compile(r'\d{3}-\d{3}-\d{4}') 创建正则表达式对象

matchObj = phoneRegex.search(待查找字符串) 查找字符串,返回Match对象

matchObj.group() 返回实际匹配文本的字符串

利用括号分组,第一对括号是第一组，第二对括号是第2组

phoneRegex = re.compile(r'(\d{3})-(\d{3}-\d{4})')

matchObj = phoneRegex.search(待查找字符串)

matchObj.group(1) 返回第一个分组

matchObj.group(2) 返回第二个分组

matchObj.group() 返回整个匹配文本即\d{3}-\d{3}-\d{4}

areaCode, mainNumber = matchObj.groups() 返回所有分组的元组

Python的正则表达式默认是贪心的，这表示在有二义的情况下，它们会尽可能匹配最长的字符串

## 文件解析

### import pprint

pprint.pprint()美观打印python数据结构

pprint.pformat()返回python数据结构的字符串表示。一般用来保存变量和内容到\*.py文件中，该文件将成为你自己的模块，如果你需要使用存储在其中的变量，就可以导入它

from pprint import pformat

cats = [{'name': 'qzlin', 'desc': 'sleepy'}, {...}]

fileObj = open('myCats.py', 'w')

fileObj.write('cats = ' + pformat(cats) + '\n')

fileObj.close()

### copy and paste

import pyperclip

pyperclip.copy('Hello world')

pyperclip.paste()

### $import json

list = [“iplaypython”, [1,2,3], {“name”: “xiaoming”}] 列表

//dumps()将Python数据结构转换成json字符串

encoded\_json = json.dumps(list) 将列表进行json格式化编码

//loads()读取json字符串

decode\_json = json.loads(encoded\_json) 解码python json格式

data\_json = json.loads(open(‘data/input.json’).read())

print json.dumps(data\_json, indent=4) 漂亮打印json数据

### $import csv

读取写入操作csv数据文件

reader()读取csv文件数据的函数方法；

writer()写入csv文件数据的函数方法；

#!/usr/bin/python

import csv

def write\_csv():

with open('data/some.csv', 'wb') as f:

writer = csv.writer(f, delimiter=' ', quotechar='|', quoting=csv.QUOTE\_MINIMAL)

writer.writerow(['a', '1', '2'])

writer.writerow(['b', '2', '3'])

def read\_csv():

with open('data/some.csv', 'rb') as f:

reader = csv.reader(f, delimiter=' ', quotechar='|')

for row in reader:

print ','.join(row)

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

write\_csv()

read\_csv()

### Beautiful Soup --HTML and XML 解析

### PDF解析

方法一：pdf2htmlEX -> BeautifulSoup

方法二：PyPDF2

PyPDF2 is a pure-python PDF library capable of splitting, merging together, cropping, and transforming the pages of PDF files.PyPDF2 is a pure-python PDF library capable of splitting, merging together, cropping, and transforming the pages of PDF files

<https://www.binpress.com/tutorial/manipulating-pdfs-with-python/167>

PyPDF2没有办法从PDF文档中提取图像、图表或其他媒体，但它可以提取文本，并将文本返回为Python字符串

import PyPDF2

pdfFileObj = open('example.pdf', 'rb')

pdfReader = PyPDF2.PdfFileReader(pdfFileObj)

//解密PDF

pdfReader.isEncrypted

pdfReader.decrypt('password')

pdfReader.numPages

pageObj = pdfReader.getPage(0)

//提取文本，可以会出错

pdfObj.extractText()

//拷贝页面，旋转页面，叠加页面，加密页面，

加水印

import os

from pyPdf import PdfFileWriter, PdfFileReader

from reportlab.pdfgen import canvas

def create\_watermark(content, pdf\_watermark):

c = canvas.Canvas(pdf\_watermark)

c.setFont('Courier', 10)

c.saveState()

c.translate(300, 15)

c.drawCentredString(0, 0, content)

c.restoreState()

c.save()

pdf\_watermark = PdfFileReader(file(pdf\_watermark, 'rb'))

return pdf\_watermark

def add\_watermark(pdf\_file, pdf\_watermark):

pdf\_output = PdfFileWriter()

pdf\_input = PdfFileReader(file(pdf\_file, 'rb'))

watermark = pdf\_watermark.getPage(0)

numPages = pdf\_input.getNumPages()

for i in xrange(numPages):

page = pdf\_input.getPage(i)

page.mergePage(watermark)

pdf\_output.addPage(page)

with open('output.pdf', 'wb') as f:

pdf\_output.write(f)

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

pdf\_watermark = create\_watermark('www.site-digger.com', 'watermark.pdf')

add\_watermark('test.pdf', pdf\_watermark)

### Excel解析

一个Excel电子表格文档称为一个工作簿。每个工作簿含多个工作表，用户当前查看的表，称为活动表

每个表都有一些列（地址是从A开始的字母）和一些行（地址是从1开始的数字）

import openpyxl

wb = openpyxl.load\_workbook('example.xlsx')

wb.get\_sheet\_names()

sheet = wb.get\_sheet\_name('Sheet3')

sheet.title

activeSheet = wb.get\_active\_sheet()

sheet['A1'].value

sheet.cell(row=1, column=2).value

for rowOfCellObjects in sheet['A1':'C3']:

for cellObj in rowOfCellObjects:

fun(cellObj.coordinate, cellObj.value)

wb.save('exmaple\_copy.xlsx')

from openpyxl.styles import Font, Style

styleOj = Style(font=Font(size=24, italic=True))

sheet['A'].style = styleObj

sheet['A1'] = 'Hello world'

wb.save('styled.xlsx')

设置行高列宽，合并拆分单元格，图表等。。。

### Word解析

在最高一层，Document对象表示整个文档，Document对象包含一个Paragraph对象的列表，表示文档中的段落，每个Paragraph对象都包含一个Run对象的列表。

Word文档中的文本不仅仅是字符串，它包含与之相关的字体、大小、颜色和其他样式信息

pip install python-docx

import docx

doc = docx.Document('demo.docx')

paragraphList = doc.paragraphs

paragraphList[0].text

runList = paragraphList[0].runs

runList[0].text

fullText = []

for para in doc.paragraphs:

fullText.append(para.text)

'\n'.join(fullText)

设置Paragraph and Run对象的样式

对于Word文档，有3种类型的样式：段落样式可以应用于Paragraph对象，字符样式可以应用于Run对象，链接样式可以应用于这两种对象

具体见<Python编程快速上手>p256

paraObj = doc.add\_paragraph('...')

paraObj.add\_run('...')

doc.add\_picture('zo.png', width=docx.shared.Inches(1), height=docx.shared.Cm(4))

在处理PDF and Word文档时有很多限制，因为这些格式的本章是很好地展示给人看，而不是让软件易于解析。而Json and CSV文件是设计给计算机使用

## 多线程，定时任务和GUI自动化

### 多线程

计算机的时钟可以调度程序，在特定的时间和日期运行，或定期运行

利用subprocess and threading模块，可以编程按时启动其他程序

不必让所有的代码等待，直到time.sleep()函数完成，你可以使用Python的threading模块，在单独的线程中执行延迟或安排的代码。这个单独的线程将因为time.sleep()调用而暂停。同时，程序可以在原来的线程中做其他工作

import threading, time

print('Start of program')

def takeANap():

time.sleep(5)

print('Wake up!')

threadObj = threading.Thread(target=takeANap)

threadObj.start()

print('End of program.')

向线程的目标函数传递参数

threadObj = threading.Thread(target=print, args=['cats', 'dogs', 'frogs'], kwargs={'sep': '&'})

threadObj.start()

常规参数可以作为一个列表，传递给threading.Thread()中的args关键字参数。关键字参数可以作为一个字典，传递给threading.Thread()中的kwargs关键字参数

### 启动进程

import subprocess

subprocess.Popen([r'c:\Windows\notepad.exe', r'c:\hello.txt'])

向Popen()传递一个列表，作为唯一的参数。该列表中的第一个字符串是要启动的程序的可执行文件名，所有后续的字符串将是该程序启动时，传递给该程序的命令行参数。实际上，这个列表将作为被启动程序的sys.argv的值

### 定时任务

cron job被用于安排那些需要被周期性执行的命令。利用它，你可以配置某些命令或者脚本，让它们在某个设定的时间内周期性地运行。cron 是 Linux 或者类 Unix 系统中最为实用的工具之一。cron 服务（守护进程）在系统后台运行，并且会持续地检查 /etc/crontab 文件和 /etc/cron.\*/目录。它同样也会检查 /var/spool/cron/ 目录。

crontab 是用来安装、卸载或者列出定时任务列表的命令。cron 配置文件则用于驱动 Vixie Cron 的 cron(8) 守护进程。每个用户都可以拥有自己的 crontab 文件

编辑定时任务

$ crontab -e

m h dom mon dow /path/to/command arg1 arg2

m字段：分钟 (0-59)

h字段：小时 (0-23)

dom字段：日期 (1-31)

mon字段：月份 (1-12 [12 代表 December])

dow字段：一周当中的某天 (0-7 [7 或 0 代表星期天])

/path/to/command - 计划执行的脚本或命令的名称

简单的 crontab 示例：

### 每隔 5 分钟运行一次 backupscript 脚本 ##

\*/5 \* \* \* \* /root/backupscript.sh

### 每天的凌晨 1 点运行 backupscript 脚本 ##

0 1 \* \* \* /root/backupscript.sh

### 每月的第一个凌晨 3:15 运行 backupscript 脚本 ##

15 3 1 \* \* /root/backupscript.sh

每隔1分钟运行一次/home/qlin/.../thread-ex.py,并且将output输出到out-e

\*/1 \* \* \* \* /home/qzlin/…/threading-ex.py >> /home/qzlin/…/out-threading-ex.txt

## GUI自动化

<python编程快速上手>p352

Pyautogui模块包含一些函数，可以模拟鼠标移动、按键和滚动鼠标滚轮

$pip install python-xlib

$apt-get install scrot python-tk python-dev

$pip install pyautogui

import pyautogui

pyautogui.PAUSE = 1 每个PyAutoGUI函数调用后，暂停一秒

自动防故障功能，鼠标移到屏幕左上角，导致pyautogui.FailSafeException异常

pyautogui.FAILSAFE = True

pyautogui.moveTo(x, y, duration=0.25) 移至(x,y),需0.25秒

pyautogui.moveRel(deltaX, deltaY, duration=0.25) 相对于当前位置

try:

while True:

x, y = pyautogui.position()

positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)

print(positionStr, end='')

print('\b' \* len(positionStr), end='', flush=True)

except KeyboardInterrupt:

print('\nDone.')

pyautogui.click(x, y)

pyautogui.dragRel(deltaX, deltaY, duration=0.25)

滚动鼠标：单位的意义在每个操作系统和应用上不一样，滚动发生在鼠标的当前位置。传递正整数表示向上滚动，传递负整数表示向下滚动

pyautogui.scroll(200)

屏幕快照

im = pyautogui.screenshot()

im.getpixel((x, y))

pyautogui.pixelMathesColor(50, 200, (130, 135, 144))

打字

pyautogui.click(x, y)

pyautogui.typewrite('Hello world')

pyautogui.hotkey('ctrl', 'c')

## 数据库

### Sqlite3

SQLite是一个进程中关系数据库，SQLite设计为嵌入在应用中，而不是作为一个单独的数据库服务器程序

$sudo apt-get install sqlite3 安装sqlite3

**todo\_schema.sql**

**sqlite-generate-db.py**

import os

import sqlite3

db\_filename = 'todo.db'

schema\_filename = 'todo\_schema.sql'

db\_is\_new = not os.path.exists(db\_filename)

with sqlite3.connect(db\_filename) as conn:

if db\_is\_new:

print 'Creating schema'

with open(schema\_filename, 'rt') as f:

schema = f.read()

conn.executescript(schema)

else:

print 'Database exists'

**sqlite-query.py**

import sqlite3

db\_filename = 'todo.db'

with sqlite3.connect(db\_filename) as conn:

conn.row\_factory = sqlite3.Row

cursor = conn.cursor()

query = """

select id, priority, details, status, deadline

from task

where project = :project\_name

order by deadline

"""

cursor.execute(query, {'project\_name': 'pymotw'})

for row in cursor.fetchall():

print '%2d {%d} %-25s [%-8s] (%s)' % (row['id'], row['priority'], row['details'], row['status'], row['deadline'])

事务性

with sqlite3.connect(db\_filename) as conn:

try:

cursor = conn.cursor()

cursor.execute("""...""")

cursor.execute("""...""")

execpt Exception, err:

conn.rollback() 回滚

else:

conn.commit() 保存

## 网络数据采集

### 网络请求和响应 --requests

$pip install requests

Requests模块是“人类“用的http模块，类似的模块有urllib，urllib2，httplib，httplib2

#### 7个http动作 --发送请求, 返回Response 对象

r = requests.get('https://github.com/timeline.json')

r = requests.post("http://httpbin.org/post")

r = requests.put("http://httpbin.org/put")

r = requests.delete("http://httpbin.org/delete")

r = requests.head("http://httpbin.org/get")

r = requests.options("http://httpbin.org/get")

#### 为URL传递参数

payload = {'key1': 'value1', 'key2': 'value2'}

r = requests.get("http://httpbin.org/get", params=payload)

r.url == http://httpbin.org/get?key2=value2&key1=value1

#### 定制请求头

url = 'https://api.github.com/some/endpoint'

payload = {'some': 'data'}

headers = {'content-type': 'application/json'}

r = requests.post(url, data=json.dumps(payload), headers=headers)

表单形式的POST请求

payload = {'key1': 'value1', 'key2': 'value2'}

r = requests.post("http://httpbin.org/post", data=payload)

POST一个多部分编码(Multipart-Encoded)的文件

url = 'http://httpbin.org/post'

files = {'file': open('report.xls', 'rb')}

r = requests.post(url, files=files)

Basic Authentication

requests.get('https://api.github.com/user', auth=('user', 'pass'))

OAuth 1 Authentication

auth = OAuth1('YOUR\_APP\_KEY', 'YOUR\_APP\_SECRET',

'USER\_OAUTH\_TOKEN', 'USER\_OAUTH\_TOKEN\_SECRET')

requests.get(url, auth=auth)

Proxies

proxies = {

'http': 'http://165.225.96.34:10015',

'https': ' http://165.225.96.34:10015',

}

requests.get('http://example.org', proxies=proxies)

#### 响应对象

r.headers 响应头

r.status\_code == requests.codes.ok 响应状态码

r.cookies Cookies

r.text 响应内容

r.content 二进制响应内容

from PIL import Image

from StringIO import StringIO

i = Image.open(StringIO(r.content))

r.json JSON响应内容

r.raw 原始响应内容

#### 保存响应内容

with open(filename, 'wb') as fd:

for chunk in r.iter\_content(chunk\_size):

fd.write(chunk)

### 解析HTML and XML --Beautiful Soup

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

$pip install beautifulsoup4

$pip install lxml HTML解析器 速度快

$pip install html5lib 解析方式与浏览器相同

from bs4 import BeautifulSoup

#### 创建BeautifulSoup对象

soup = BeautifulSoup(open("index.html"))

soup = BeautifulSoup("<html>data</html>")

Beautiful Soup将复杂HTML文档转换成一个复杂的树形结构,每个节点都是Python对象,所有对象可以归纳为4种: Tag , NavigableString , BeautifulSoup , Comment .

指定文档解析器

如果仅是想要解析HTML文档,只要用文档创建 BeautifulSoup 对象就可以了，Beautiful Soup会自动选择一个解析器来解析文档.但是还可以通过参数指定使用那种解析器来解析当前文档.

BeautifulSoup 第一个参数应该是要被解析的文档字符串或是文件句柄,第二个参数用来标识怎样解析文档.如果第二个参数为空,那么Beautiful Soup根据当前系统安装的库自动选择解析器,解析器的优先数序: lxml, html5lib, Python标准库.在下面两种条件下解析器优先顺序会变化:

要解析的文档是什么类型: 目前支持, “html”, “xml”, 和 “html5”

指定使用哪种解析器: 目前支持, “lxml”, “html5lib”, 和 “html.parser”

编码

任何HTML或XML文档都有自己的编码方式,比如ASCII 或 UTF-8,但是使用Beautiful Soup解析后,文档都被转换成了Unicode:

Beautiful Soup用了编码自动检测子库来识别当前文档编码并转换成Unicode编码

编码自动检测功能大部分时候都能猜对编码格式,但有时候也会出错.有时候即使猜测正确,也是在逐个字节的遍历整个文档后才猜对的,这样很慢.如果预先知道文档编码,可以设置编码参数来减少自动检查编码出错的概率并且提高文档解析速度.在创建 BeautifulSoup 对象的时候设置 from\_encoding 参数.

#### 标签Tag对象

Tag 对象与XML或HTML原生文档中的tag相同:

soup = BeautifulSoup('<b class="boldest">Extremely bold</b>')

tag = soup.b

tag中最重要的属性: name和attributes

str(tag) tag -> str

tag.name == 'b'

tag.attrs 字典

tag['class']

tag.string == u'Extremely bold'

unicode\_string = unicode(tag.string)

tag的 .contents tag的子节点 （列表）

tag.contents[0]

tag.descendants tag的所有子孙节点 （列表）

tag.strings 子孙字符串 （列表）

tag.stripped\_strings 去除多余空白的子孙字符串 （列表）

tag.parent 元素的父节点

tag.parents 元素的所有父辈节点 (列表）

tag.next\_sibling 和 .previous\_sibling 属性来查询兄弟节点

tag.next\_siblings 和 .previous\_siblings 属性可以对当前节点的兄弟节点迭代输出

tag.get\_text() 方法,这个方法获取到tag中包含的所有文版内容包括子孙tag中的内容

#### Tag查询

find\_all( name , attrs , recursive , text , \*\*kwargs )

soup.a 第一个<a>标签:

soup.find\_all('a') 所有的<a>标签

soup.find\_all(["a", "b"]) 所有<a>标签和<b>标签

soup.find\_all(id='link2')

soup.find\_all("a", attrs={"class": "sister"})

soup.find\_all(text=re.compile("Dormouse"))

soup.select("p nth-of-type(3)") 基于css选择器方法，返回Tag列表

Tag创建与替换

创建一个tag最好的方法是调用工厂方法 BeautifulSoup.new\_tag() :

tag.append(new\_tag)

PageElement.replace\_with() 方法移除文档树中的某段内容,并用新tag或文本节点替代它

PageElement.wrap() 方法可以对指定的tag元素进行包装

#### Tag美观打印

prettify() 方法将Beautiful Soup的文档树格式化后以Unicode编码输出,每个XML/HTML标签都独占一行

#### 常用场景：

从文档中找到所有<a>标签的链接

for link in soup.find\_all('a'):

print(link.get('href'))

# http://example.com/elsie

# http://example.com/lacie

# http://example.com/tillie

从文档中获取所有文字内容

print(soup.get\_text())

# The Dormouse's story

#

# ...

下载所有XKCD漫画

import requests, os, bs4

url ='http://xkcd.com'

os.makedirs('xkcd', exist\_ok=True)

while not url.endswith('#'):

res = requests.get(url)

res.raise\_for\_status() #如果下载文件出错，抛出异常

soup = bs4.BeautifulSoup(res.text)

comicElem = soup.select('#comic img')

if comicElem == []:

print('Could not find comic image.')

else:

comicUrl = 'http:' + comicElem[0].get('src')

res = requests.get(comicUrl)

res.raise\_for\_status()

即使页面是纯文本的，你也需要写入二进制数据，而不是文本数据，目的是为了保存该文本中的"unicode编码"

imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)), 'wb')

for chunk in res.iter\_content(100\*1000):

imageFile.write(chunk)

imageFile.close()

#Get the Prev button's url

prevLink = soup.select('a[rel="prev"]')[0]

url = 'http://xkcd.com' + prevLink.get('href')

用requests的响应text创建BeautifulSoup会有中英文乱码问题，建议使用如下

import urllib2, bs4

html = urllib2.urlopen(url).read()

soup = bs4.BeautifulSoup(html, 'lxml')

### 模拟浏览器操作

#### import webbrowser

webbrowser.open('http://inventwithpython.com/') 启动浏览器，打开url

selenium模块让Python直接控制浏览器，实际点击链接，填写登录信息, 与Requests and Beautiful Soup相比，Selenium允许你用高级得多的方式与网页交互。但因为它启动了Web浏览器，假如你只是想从网络上下载一些文件，会有点慢，并且难以在后台运行

具体见<Python编程快速上手—让繁琐工作自动化>p210

#### from selenium import webdriver

browser = webdriver.Firefox()

browser.get('http://inventwithpython.com')

查找

browser.find\_elements\_by\_css\_selector(selector)

如果页面上没有元素匹配该方法要查找的元素，selenium模块就会抛出NoSuchElement异常

点击

linkElem = browser.find\_element\_by\_link\_text('Read It Online')

linkElem.click()

填写并提交表单

browser.get('http://gmail.com')

emailElem = browser.find\_element\_by\_id('Email')

passwordElem = browser.find\_element\_by\_id('Passwd')

emailElem.send\_keys('lin.qizhong@gmail.com')

passwordElem.send\_keys('...')

passwordElem.submit() 在任何元素上调用submit()方法，等同于该元素所在表单的submit按钮

from selenium.webdriver.common.keys import Keys

htmlElem = browser.find\_element\_by\_tag\_name('html')

htmlElem.send\_keys(Keys.END)

htmlElem.send\_keys(Keys.HOME) 按下home键，从而页面滚动至顶

## web framework

### build-in web server to setup a web server

$python -m SimpleHTTPServer 8080 默认是8000

$curl localhost:8080

相当于ls .

$curl localhost:8080/static/data.txt

相当于cat static/data.txt

### Flask

具体见Python.docx

## 表单自动填写

<https://selenium-python.readthedocs.io/installation.html>

$apt-get -y install python-pip

$ pip install -U selenium (若不成功)

or

download selenium\*.tar.gz from <https://pypi.python.org/pypi/selenium#downloads>

tar –xvzf selenium\*.tar.gz

cd selenium\*

selenium\*$sudo python setup.py install

若浏览器用ubuntu自带的firefox，selenium 3版本以上需要geckodriver

download geckodriver <https://github.com/mozilla/geckodriver/releases> and mv geckordriver to /usr/bin

# <python基础教程>

python.py -> Python解释器 -> 操作系统 -> 硬件

## 模块

模块（即一个py文件）主要用于定义：在模块作用域里定义类，函数，变量等等，然后**将模块放在可以被解释器找到的路径中**。

引用: 任何Python程序都可以作为模块导入,如math.py

import math; 用import导入模块，然后按照模块.函数的格式使用这个模块的函数

from math import sqrt; 直接使用math模块里的sqrt函数

模块能够明白自己是否是主程序,如Convert.py

def to\_celsius(t):

return (t - 32.0) \* 5.0 / 9.0

**if \_\_name\_\_ == '\_\_main\_\_': #主要是测试代码**

print to\_celsius(80)

如果Convert.py当成main运行,则\_\_name\_\_变成\_\_main\_\_

如果Convert.py被别的模块引用，则\_\_name\_\_变成模块名Convert

如何搜索到模块？

import sys, pprint

pprint.pprint(**sys.path**) #查看Python解释器的搜索目录

探究模块:

import media, pprint

pprint.pprint([n for n in dir(media) if not n.startswith('\_')])

print media.\_\_all\_\_

**help(media)**

print media.\_\_file\_\_ #查看源代码

标准库：

import sys #访问与Python解释器联系紧密的变量和函数 (如：启动外部程序os.startfile(r’…exe’)

import time, random, re

## 包

相关的模块存储在各自的文件中(.py)，包将模块组织在一起，包就是模块所在的目录。为了让python将其作为包对待，它必须包含一个命令为\_\_init\_\_.py的文件（模块）

如：drawing包

site-packages/drawing/ #包目录

site-packages/drawing/\_\_init\_\_.py #包代码(drawing模块)

site-packages/drawing/colors.py #包中colors模块

site-packages/drawing/shapes.py #包中shapes模块

如何引用？

import drawing #import the drawing package

import drawing.colors #import the colors module,使用时需用drawing.colors.fun()

from drawing import shapes #import the shapes model, 使用时只需用shapes.fun()

## python基本类型

类似c语言sys(“pause”), python语言raw\_input(“press<enter>”)

python以:和缩进格式表示block，function…并且没有分号结尾 （C#,java用{}表示）

python也有docstring,模块头，函数头用’’’注释内容‘’’,这样help(模块名)或help（模块名.函数名）就能看到注释文档

字符串:

单引号 ‘hello world’

双引号”hello world”

长字符跨多行:三引号’’’hello

Qzlin’’’

或

‘Hello \

Qzlin’

原始字符串（无转义，类似C#中的@）： r’c:\Program Files\fnord’

unicode字符串: u’林其忠’

toString(): Javascript: “value=” +３　结果是value=3 ; Python: “value”+3 非法，需改为”value” + str(3)

print: 用逗号，要以打印所有类型，如print ‘Age:’, 22

条件测试：false none 0 “” () [] {} 被解释为假. “==”判定同一性(别名)，”is”判定相等性(值相等)

debug： assert （同c#, java）

## 序列

### 通用操作

索引: arr[i]

切片: arr[0:4]

连接(记住：同类型才能相加): [1, 2, 3] + [4]

复制多次: ‘python’\*5, [42]\*10

初始值: []空列表, [0]\*10长度为10，值为0的列表，[none]\*10预留长度为10，没有值的列表

成员资格: ‘[low]’ in arr 检查子串是否在长字串

内建函数: len, min, max

### 列表

是异构的,内容可变，除了pop之外，所有方法只会返回一个特殊值None,也就是说只会修改原列表，而不会创建新列表（因为变量是引用）

初始化：arr = [‘Hello’, 1] ;

序列->list: list(‘Hello’)

插入元素: arr[1:1] = [‘world’]

删除元素：arr[i:j] = [] or del arr[i]

List对象的方法: append, count, extend, index, insert, pop, remove, reverse, sort,

List遍历:

for (i, v) in enumerate(arr):

values[i] = 2\*v ;

enumerate(arr) 返回元组(i, value[i]), Python允许多值赋值，即左侧有多个变量，且右侧有相同数量的值，Python就会对它们进行配对，并一次性完成所有赋值操作.

### 元组

创建： 通过逗号分隔值，或通过圆括号如

Tuple0 = ()

Tuple1 = 1，or (1,)

Tuple3 = 1,2,3 or (1, 2, 3)

序列 -> 元组: tuple([1,2,3])

元组操作: 创建元组,访问元组元素

元组的应用: 元组不可变，所以可以在映射（和集合的成员）中当作键使用.元组作为很多内建函数和方法的返回值存在.

### 字符串

格式化：print ‘%s plus %s equals %s’ % (1, 1, 2)

方法：find,join,lower, replace, split, strip, translate,

如:’/’.join((‘’, ‘usr’, ‘bin’, ‘env’)) => ‘/usr/bin/env’

### 集合:通过列表或元组定义

如: set((2,3,4)), set([2, 3, 4]) ，集合的基本操作同数学如: set1 & set2

### 字典

字典的键只能为不可变类型，如整形，浮点型，字符串，或元组

scientists = {‘Newton’:1632, ‘Darwin’:1809}

序列转字典: dict([(‘name’, ‘Gumby’), (‘age’, 42)]

x = []

x[42] = ‘ff’ ; 出错，列表不可以，需要预留空间x = [none] \* 43

x = {}

x[42] = ‘ff’ ; 字典可以，自动添加

方法：clear,copy,fromkeys,get，has\_key,items,iteritems,keys,iterkeys,popitem, setdefault,update,values, itervalues

一般来说，如果试图访问字典中不存在的项时，如d[‘qzlin’]会出错。d.get(‘qzlin’, ‘N/A’)；当使用get访问一个不存在的键时，普通返回是none，可以自定义默认值N/A替换none.

for (key, value) in scientists.iteritems():

…

### 常用操作

#### 序列解包

x,y = y,x 赋值同时进行,这个特性在当函数或者方法返回元组（或者其他序列或可迭代对象）时，这个特性尤其有用。它允许函数返回一个以上的值并且打包成元组，然后通过一个赋值语句很容易进行访问。

key, value = scoundrel.popitem()

#### 并行迭代多个序列

for name, age in zip(names, ages):

…

#### 编号迭代

for index, string in enumerate(strings)：

strings[index] =

#### 列表推导式

是利用其他列表创建新列表（类似于数学术语中的集合推导式）

list1 = [(x,y) for x in range(3) for y in range(3)]

 result = []

for x in range(3):

for y in range(3):

result.append((x,y))

list2 = [x\*x for x in range(10) if x%3==0]

## 不常用的命令：

pass: 什么都不做，只是占位符

del: 删除变量，或者数据结构的一部分，但是不能用来删除值。

exec: 执行字符串，字符串里是python语句

eval: 执行字符串表达式，并返回结果

## 性能比较：

import time

def time\_it(search, v, L):

t1 = time.time()

search(v,L)

t2 = time.time()

return (t2-t1)\*1000

## 函数

同C++一样，函数可以有默认的参数值。

同Java一样，可变参数列表

def max(\*values):

…

max(1,2,3) => 输入参数1,2,3会形成一个元组赋给values

同Java,C#一样，可以用名称对参数进行指定能让我们以任意顺序对参数进行排列。比如一个用于在GUI中显示文本的函数可以接收很多个参数，分别用于指定字体，粗细，顔色，边框等信息。这个时间利用名称对参数进行标记就会显得轻松许多.

## 异常

同Java,C++

try:

fun()

except exception, e:

…

finally:

…

在做一件事时去处理可能出现的错误，而不是在开始做事前就进行大量的检查，这个策略可以总结为习语“看前就跳(leap before you look)”

import traceback

try:

...

except:

f=open("c:log.txt",'a')

traceback.print\_exc(file=f)

f.flush()

f.close()

## 类

函数前后双下划线，表示特殊方法，主要表达被python自动调用.

函数前双下划线，表示私有方法

如：

\_\_init\_\_(self,…) 构造器

\_\_del\_\_(self) 析构方法,一般用不上

\_\_inaccessible(self) 私有方法

实现序列和映射 （应用遵寻规则，实现他们基本的行为）

\_\_len\_\_(self), \_\_getitem\_\_(self, key), \_\_setitem\_\_(self, key, value), \_\_delitem\_\_(self, key),\_\_iter\_\_

一般是子类化内建类型list or dict

class SmartList(list):

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

super(SmartList, self).\_\_init\_\_(\*args)

类似C#属性：隐藏访问器方法，让所有特性看起来一样。

class Rectangle(object):

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

self.width = 0

self.height = 0

def setSize(self, size):

self.width, self.height = size

size = property(getSize, setSize)

r = Rectangle()

r.size = 150,100 #给size赋值时，会通过属性调用setSize方法

拦截对象的所有字段访问，为了在访问字段的时候可以执行代码，如：

class Rectangle:

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

self.width = 0

self.height = 0

def \_\_setattr\_\_(self, name, value):

if name == ‘size’:

self.width, self.height = value

else:

self.\_\_dict\_\_[name] = value

def \_\_getattr\_\_(self, name):

If name == ‘size’:

return self.width, self.height

else:

raise AttributeError

r = Rectangle()

r.size = 100,200 #给对象赋值时，会自动调用\_\_setattr\_\_

(width, height) = r.size #取对象属性值时，会自动调用\_\_getattr\_\_

## class 继承

\_\_metaclass\_\_ = type #新式类语法 或者用Class Organism(object):

#Organism 继承于object

class Organism:

#所有数据通过构造器

def \_\_init\_\_(self, name, x, y):

self.name = name

self.x = x

self.y = y

self.atoms = []

#类似于Java toString()

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

return ‘(%s, [%s, %s])’ %\

(self.name, self.x, self.y)

#公有方法

def move(self):

…

#私有方法

def \_\_inaccessible(self):

…

#静态方法

@staticmethod

def smeth():

…

python并没有真正的私有化支持，解释器会对上面翻译成

Organism. move(self)

Organism.\_Organism\_\_inaccessible(self)

所以在实际中，

o = Organism()

b.move() #公有方法访问

b.\_Organism\_\_inaccessible() #私有方法访问

# 继承于Organism

class Arthropod(Organism):

def \_\_init\_\_(self, name, x, y, legs):

#老方法

Organism.\_\_init\_\_(self, name, x, y)

#新方法

super(Arthropod, self).\_\_init\_\_(name, x, y)

self.legs = legs

#以下子类与父类同名函数，实现多态性

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

…

def move(self):

..

## 迭代器

流式处理。 列表（一次性处理）

class Fibs:

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

self.a = 0

self.b = 1

def next(self):

self.a, self.b = self.b, self.a+self.b

return self.a

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

return self

fibs = Fibs()

for f in fibs:

if f > 1000:

print f

break

## 生成器

任何包含yield语句的函数称为生成器。每次产生一个值（使用yield语句），函数就会被冻结：即函数停在那点等待被激活。函数被激活后就从停止的那点开始执行.

生成器推导式与列表推导式区别：

列表推导式返回的是整个列表，而生成器推导式返回的是一个一个值，进行流式处理

e.g：

list1 = [(i+2)\*\*2 for i in range(2,27)] #返回的是整个列表

generator = ((i+2)\*\*2 for i in range(2,27)) #返回的是生成器

generator.next() #值会是16

generator.next()

…

递归生成器应用：打印列表中的元素，列表元素可能为列表

#遍历列表中的数字

def flatten(nested):

try:

for sublist in nested:

for element in flatten(sublist):

yield element

except TypeError:

yield nested

list(flatten([ [[1],2], 3, 4, [5, [6,7]], 8]) )

>>>[ 1, 2, 3, 4, 5, 6, 7, 8]

#遍历列表中的所有元素

def flatten(nested):

try:

try:

nested + ''

except TypeError:

pass

else:

raise TypeError

for sublist in nested:

for element in flatten(sublist):

yield element

except TypeError:

yield nested

list1 = ['foo', ['bar', ['baz']]]

list\_flatten = flatten(list1)

print list(list\_flatten)

## 文件

返回是文件对象

文件迭代器：当文件对象用于for loop时，Python会自动调用readline方法

e.g:

data = open(‘data.txt’, ‘r’)

for line in data:

print len(line.strip())

data.close()

有类似于C# using：

with open("data.txt’") as file:

for line in file:

print len(line.strip())

open函数的第3个参数控制着文件的缓冲。如果参数是0,I/O就是无缓冲的（所有的读写操作都是直接针对硬盘）；如果是1，I/O就是有缓冲的，（意味Python使用内存来代替硬盘，让程序更快，只有使用flush或close时才会更新硬盘上的数据.写入过的文件总是应该关闭，是因为python可能会缓存（出于效率的考虑而把数据临时存储在某处）写入的数据，如果程序因为某些原因崩溃了，那么数据根本就不会被写入文件。

#文件内容如果可以很小，一次性读取所有内容:

f = open(filename)

for line in f.readlines():

process(line)

f.close()

#文件内容很大，流式处理,一行一行读取

import fileinput

for line in fileinput.input(filename):

process(line)

# <Python in Practice>

1. sequence unpacking and map unpacking

first, second, \*rest = sequence

等同于first =sequence[0], second=sequence[1], rest=sequence[2:]

Sequence unpacking可以用来传递任意数量的位置参数

map unpacking可以用来传递任意数量的关键字参数

args = (600, 900)

kwargs = dict(copies=2, collate=False)

def print\_args(\*args, \*\*kwargs):

args 是传入位置参数的tuple

kwargs是传入关键字参数的dict

1. locals() & glocal()

它囊括了函数执行到该时间点时所定义的一切变量, 返回的字典对所有局部变量的名称与值进行映射.常配合mapping unpacking，用来实例化字符串模板

str.format(\*\*locals()).

字符串模板形参需要的实参从locals()返回的dict传入

重点：

这两个函数主要提供，基于字典的访问局部和全局变量的方式。  
Python使用叫做名字空间的东西来记录变量的轨迹。名字空间只是一个字典，它的键字就是变量名，字典的值就是那些变量的值。实际上，名字空间可以象Python的字典一样进行访问  
每个函数都有着自已的名字空间，叫做局部名字空间，它记录了函数的变量，包括函数的参数  
和局部定义的变量。

每个模块拥有它自已的名字空间，叫做全局名字空间，它记录了模块的变量，包括函数、类、其它导入的模块、模块级的变量和常量。

还有就是内置名字空间，任何模块均可访问它，它存放着内置的函数和异常。  
当一行代码要使用变量 x 的值时，Python会到所有可用的名字空间去查找变量，按照如下顺序：  
a.局部名字空间 - 特指当前函数或类的方法。如果函数定义了一个局部变量 x，Python将使用  
  这个变量，然后停止搜索。  
b.全局名字空间 - 特指当前的模块。如果模块定义了一个名为 x 的变量，函数或类，Python  
  将使用这个变量然后停止搜索。  
c.内置名字空间 - 对每个模块都是全局的。作为最后的尝试，Python将假设 x 是内置函数或变量。

from module import 和 import module之间的不同。使用 import module，模块自身被导入，但是它保持着自已的名字空间，这就是为什么你需要使用模块名来访问它的函数或属性（module.function）的原因。但是使用 from module import，实际上是从另一个模块中将指定的函数和属性导入到你自己的名字空间，这就是为什么你可以直接访问它们却不需要引用它们所来源的模块的原因。

1. @classmethod and @staticmethod的区别:

C++中classmethod和staticmethod是一个概念, 类的静态函数

Python中, staticmethod不常用，可以用模块的函数来替代。Python是脚本语言，classmethod的参数class可以是子类 , 也就是说，classmethod可以通过子类来进行重定义。(相当于接口)

二者最大的区别：对于classmethod的参数，需要隐式地传递类名，而staticmethod参数中则不需要传递类名

范例：

1. class MyClass:
2. @classmethod  # classmethod的修饰符
3. def class\_method(cls, arg1, arg2, ...):
4. @staticmethod  # staticmethod的修饰符
5. def static\_method(arg1, arg2, ...):

除了类级别的函数，还有类级别的变量

1. class MyClass:
2. i = 123 # class-level variable
3. def \_\_init\_\_(self):
4. self.i = 456 # object-level variable

为了清晰地区分上面两个i，最好的办法就是考虑到python中的一切都是object，所以i=123属于class object的，i=456属于class instance object

范例:

def create\_diagram(factory):

diagram = factory.make\_diagram(30, 7)

rectangle = factory.make\_rectangle(4, 1, 22, 5, "yellow")

text = factory.make\_text(7, 3, "Abstract Factory")

diagram.add(rectangle)

diagram.add(text)

return diagram

class DiagramFactory:

@classmethod

def make\_diagram(Class, width, height):

return Class.Diagram(width, height)

@classmethod

def make\_rectangle(Class, x, y, width, height, fill="white",

stroke="black"):

return Class.Rectangle(x, y, width, height, fill, stroke)

@classmethod

def make\_text(Class, x, y, text, fontsize=12):

return Class.Text(x, y, text, fontsize)

class Diagram:

class Rectangle:

class Text:

class SvgDiagramFactory(DiagramFactory):

class Diagram:

class Rectangle:

class Text:

1. \_\_slots\_\_

\_\_slots\_\_限定类对象属性范围，同时优化内存(默认情况下，Python用一个dict来存储对象实例的属性, 对一些在”编译”前就知道该有几个固定属性的小class来说，这个dict就有点浪费内存了。而当你把这个小浪费乘上一百万，那可就大不同了。在Python中，你可以在class中设置\_\_slots\_\_，它是一个包含这些固定的属性名的list。这样Python就不会再使用dict，而且只分配这些属性的空间)

动态语言: 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, 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。

class Student(object):

s = Student()

s.name = 'Michael' # 动态给实例绑定一个属性

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, Student) # 给实例绑定一个方法

s.set\_age(25) # 调用实例方法

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

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 = MethodType(set\_score, None, Student)

给class绑定方法后，所有实例均可调用

但是，如果我们想要限制class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

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',但不在\_\_slots\_\_属性范围内，错误

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

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

1. \_\_init\_\_, \_\_new\_\_, and \_\_call\_\_区别

\_\_new\_\_(cls, \*args, \*\*kwargs)  创建对象时调用，返回当前对象的一个实例;注意：这里的第一个参数是cls即class本身  
\_\_init\_\_(self, \*args, \*\*kwargs) 创建完对象后调用，对当前对象的实例的一些初始化，无返回值,即在调用\_\_new\_\_之后，根据返回的实例初始化；注意，这里的第一个参数是self即对象本身【注意和new的区别】  
\_\_call\_\_(self,  \*args, \*\*kwargs) 如果类实现了这个方法，相当于把这个类型的对象当作函数来使用，相当于 重载了括号运算符

1. Exec and eval

eval(str [,globals [,locals ]])函数将字符串str当成有效Python表达式来求值，并返回计算结果。

globals = {'x': 7, 'y': 10, 'birds': ['Parrot', 'Swallow', 'Albatross']}  
locals = { }  
a = eval("3\*x + 4\*y", globals, locals)

exec语句用来执行储存在字符串或文件中的Python语句,结果存在变量中，而变量会更新locals

1. Type使用

即查看一个对象的类型(构建类的实例),常见的有int, str, float, 自定义类型.还有一种叫类类型type,如内建类型或用户定义类型的类型，也称为元类metaclass(用来构建类的东西)

type('foo')   => <class 'str'>

class Foo(object):

...     pass

type(Foo) => <class 'type'>

type(name, bases, dict) -> a new type

Foo = type('Foo', (), {})   #等同于创建类Foo

Foo = type('FooBar', (Foo), {'always\_false': always\_false})   #创建子类

glocals()[‘Foo’] = Foo #将创建的类添加到当前模块sys.modules[\_\_name\_\_]中等同于:

setattr(sys.module[\_\_name\_\_], ‘Foo’, Foo)

1. 面向函数编程：提供的函数

filter(function, sequence)：对sequence中的item依次执行function(item)，将执行结果为True的item组成一个List/String/Tuple（取决于sequence的类型）返回：  
>>> def f(x): return x % 2 != 0 and x % 3 != 0   
>>> filter(f, range(2, 25))   
[5, 7, 11, 13, 17, 19, 23]  
>>> def f(x): return x != 'a'   
>>> filter(f, "abcdef")   
'bcdef'

map(function, sequence) ：对sequence中的item依次执行function(item)，见执行结果组成一个List返回：  
>>> def cube(x): return x\*x\*x   
>>> map(cube, range(1, 11))   
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]  
>>> def cube(x) : return x + x   
...   
>>> map(cube , "abcde")   
['aa', 'bb', 'cc', 'dd', 'ee']  
另外map也支持多个sequence，这就要求function也支持相应数量的参数输入：  
>>> def add(x, y): return x+y   
>>> map(add, range(8), range(8))   
[0, 2, 4, 6, 8, 10, 12, 14]

reduce(function, sequence, starting\_value)：对sequence中的item顺序迭代调用function，如果有starting\_value，还可以作为初始值调用，例如可以用来对List求和：  
>>> def add(x,y): return x + y   
>>> reduce(add, range(1, 11))   
55 （注：1+2+3+4+5+6+7+8+9+10）  
>>> reduce(add, range(1, 11), 20)   
75 （注：1+2+3+4+5+6+7+8+9+10+20）

lambda：这是Python支持一种有趣的语法，它允许你快速定义单行的最小函数，类似与C语言中的宏，这些叫做lambda的函数，是从LISP借用来的，可以用在任何需要函数的地方：   
>>> g = lambda x: x \* 2   
>>> g(3)   
6

1. \_\_call\_\_(self)

对象通过提供\_\_call\_\_(slef, [,\*args [,\*\*kwargs]])方法可以模拟函数的行为，如果一个对象x提供了该方法，就可以像函数一样使用它，也就是说x(arg1, arg2...) 等同于调用x.\_\_call\_\_(self, arg1, arg2) 。  模拟函数的对象可以用于创建防函数(functor) 或代理(proxy)

class DistanceForm(object):

    def \_\_init\_\_(self, origin):

        self.origin = origin

        print "origin :"+str(origin)

    def \_\_call\_\_(self, x):

        print "x :"+str(x)

p = DistanceForm(100)

p(2000)

1. Tuple and namedtuple

Jane = ('Jane',29,'female')  
print 'Field by index:', Jane[0]

Import collections  
Person = collections.namedtuple('Person','name age gender')

Jane = Person(name='Jane',age=29,gender='female')  
print 'Field by Name:', Jane.name

# <Python编程快速上手—让繁琐工作自动化>

## 基础知识

range(0, 10, 2)将从0数到8,间隔为2

在Python中有一个值称为None,它表示没有值。**None是NoneType数据类型的唯一值（其他编程语言称为null, nil或undefined）**

下标从0开始增长，整数值-1指列表的最后一个下标，-2倒数第二…

del语名将删除列表中下标处的值，表中被删除值后面的所有值，都将向前移动一个下标。

for i in range(len(list)):

...

list[random.randint(0, len(list)-1)] 随机数下标

**列表是可变的**数据类型，它的值可以添加、删除或改变。**字符串和元组是不可变的**。在变量必须保存**可变数据类型**的值时，例如列表或字典，**Python使用引用**。对于**不可变的数据类型**的值，例如字符串、整形或元组，**Python变量就保存值本身**

import copy

copy.copy()复制列表或字典

copy.deepcopy()复制列表 （含嵌套列表）

字典函数

for k in dict.keys():

for v in dict.values():

for k, v in dict.items():

key in dict 检查字典是否含键

dict.get(key, defaultValue) 获取键值，若不存在则返回初始值

dict.setdefault(key, defaultValue) 检查键，若不存在则设置初始值

<Effective Python>

# <Flask Web开发>

## tutorial

$pip install Flask

Pyramid, Django, and Flask

Flask is a "microframework" primarily aimed at small applications with simpler requirements.

Pyramid and Django are both aimed at larger applications

hello.py

from flask import Flask

from flask import request

app = Flask(\_\_name\_\_ , static\_url\_path='/static')

@app.route("/")

def hello():

return "Hello World!"

@app.route('/login', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

do\_the\_login()

else:

show\_the\_login\_form()

if \_\_name\_\_ == "\_\_main\_\_":

app.run()

$python hello.py

curl localhost:5000/

常规应用

$sudo pip install virtualenv

flask-app$ virtualenv env

flask-app$ source env/bin/activate

(env)$ pip install flask

(env)$ pip install flask-script

(env)$ gedit hello.py

#!./env/bin/python

from flask import Flask

from flask\_script import Manager

app = Flask(\_\_name\_\_)

manager = Manager(app)

@app.route('/')

def index():

return '<h1>Hello World!</h1>'

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

manager.run()

(env)$ chmod u+x hello.py

默认情况下，Flask开发Web服务器监听localhost上的连接，若让web服务器监听公共网络接口上的连接，允许同网中的其他计算机连接服务器，需要监听所有IP即0.0.0.0

(env)$ ./hello.py runserver --host 0.0.0.0

以shell方式启动hello.py

(env)$ ./hello.py shell

>>> from hello import \*

(env)$ deactivate

在本地上连接

$curl localhost:5000

在连网别的电脑上连接

$curl 192.168.1.12:5000/ 《假定hello.py服务器IP地址是192.168.1.12

## 视图

### 模板

from flask import Flask, render\_template

@app.route('/user/<name>')

def user(name):

return render\_template('user.html', name=name)

Flask提供的render\_template函数把Jinja2模板引擎集成到了程序中。render\_template函数的第一个参数是模板的文件名。随后的参数都是键值对，表示模板中变量对应的真实值。

变量

特殊占位符，告诉模板引擎这个位置的值从渲染模板时使用的数据中获取

Jinja2能识别所有类型的变量，甚至是一些复杂的类型，例如列表、字典和对象。

<p>A value from a dict: {{ mydict['key'] }} </p>

<p>A value from a list: {{ mylist[myintvar] }} </p>

<p>A value from object: {{ obj.somemethod() }} </p>

过滤器

<div>{{ content | safe }} </div> 显示变量存储的html代码，使用safe过滤器

控制结构

{% if condition %}

<div>content1</div>

{% else %}

<div>content2</div>

{% endif %｝

<ul>

{% for item in items %}

<li>{{ item }}</li>

{% endfor %}

</ul>

复用

宏

macros.html

{% macro render\_comment(comment) %}

<li>{{ comment }}</li>

{% endmacro %}

comments.html

{% include 'macros.html' %}

<ul>

{% for item in items %}

{{ render\_comment(item) }}

{% endfor %}

</ul>

模板继承

base.html

<html>

<head>

{% block head %}

<title>{% block title %}{% endblock %} - My Application</title>

{% endblock %}

</head>

<body>

{% block body %}

{% endblock %}

</body>

</html>

extend.html

{% extends "base.html" %}

{% block title %}

Index

{% endblock %}

{% block head %}

{{ super() }}

<style></style>

{% endblock %}

{% block body %}

<h1>Hello, World!</h1>

{% endblock %}

Extends指令声明这个模板继承自base.html,继承之后，基模板中的3个块被重新定义，模板引擎会将其插入适当的位置。super()调用父模板的内容

### 集成Twitter Bootstrap

(env)$ pip install flask-bootstrap

from flask\_bootstrap import Bootstrap

bootstrap = Bootstrap(app)

初始化Flask-Bootstrap之后，就可以在程序中使用一个包含所有Bootstrap文件的基模板。这个模板利用Jinja2的模板继承机制，让程序扩展一个具有基本页面的基模板，其中就有用来引入Bootstrap的元素

Flask-Bootstrap中的基模板提供了一个网页框架，引入了Bootstrap中的所有CSS and JavaScript文件

{% extends "bootstrap/base.html" %}

{% block scripts %}

{{ super() }}

<script type="text/javascript" src="my-script.js"></script>

{% endblock %}

{% block title %} my title {% endblock %}

{% block navbar %}... {% endblock %}

{% block content %} ... {% endblock %}

其余块见p25

### 自定义错误页面

@app.errorhandler(404)

def page\_not\_found(e):

return render\_template('404.html'), 404

@app.errorhandler(500)

def internal\_server\_error(e):

return render\_template('500.html'), 500

404.html

500.html

### 链接

{% block head %}

{{ super() }}

<link rel="shortcut icon" href="{{ url\_for('static', filename='favicon.ico') }}" type="image/x-icon">

{% endblock %}

url\_for()函数，第一个参数是视图函数名，第二个参数动态部

url\_for('user', name='john', \_external=True)返回的结果

<http://localhost:5000/user/john>

### 静态文件

默认设置下，Flask在程序根据目录中名为static的子目录中寻找静态文件

### 使用Flask-Moment本地化日期和时间

要想在服务器上只使用UTC时间，一个优雅的解决方案是：把时间单位发送给Web浏览器，转换成当地时间，然后渲染。Web浏览器可以更好地完成这一任务，因为它能获取用户电脑中的时区和区域设置

(env) $ pip install flask-moment

启动逻辑

from flask.ext.moment import Moment

moment = Moment(app)

from datetime import datetime

@app.route('/')

def index():

return render\_template('index.html', current\_time=datetime.utcnow())

模板

<p>The local date and time is {{ moment(current\_time).format('LLL') }}.</p>

### Web表单

(env)$ pip install flask-wtf

app = Flask(\_\_name\_\_)

app.config[ 'SECRET\_KEY' ] = 'hard to guess string' app.config字典存储框架、扩展和程序本身的配置变量

表单类 (具体见p35)

from flask\_wtf import Form

from wtforms import StringField, SubmitField

from wtforms.validators import Required

class NameForm(Form):

name = StringField('What is your name?', validators=[Required()])

submit = SubmitField('Submit')

把表单渲染成HTML

Flask-Bootstrap提供了一个非常高端的辅助函数，可以使用Bootstrap中预先定义了的表单渲染整个Flask-WTF表单

使用Bootstrap默认样式渲染传入的表单

{% import "bootstrap/wtf.html" as wtf %}

{{ wtf.quick\_form(form) }}

在视图函数中处理表单

@app.route('/', methods=['GET', 'POST']

def index():

name = None

form = NameForm()

if form.validate\_on\_submit():

name = form.name.data

form.name.data = ''

return render\_template('index.html', form=form, name=name)

1. 用户第一次访问程序时，服务器会收到一个没有表单数据的GET请求，所以validate\_on\_submit()将返回False. If语句的内容将被跳过，通过渲染模板处理请求，并传入表单对象和值为None的name变量作为参数。用户会看到浏览器中显示了一个表单
2. 用户提交表单后，服务器收到一个包含数据的POST请求。Validate\_on\_submit()会调用name字段上附属的required()验证函数。如果名字不为空，就能通过验证，validate\_on\_submit()返回True.

推荐如下写法：

@main.route('/login', methods=['GET', 'POST'])

def login():

form = NameForm()

if request.method == 'POST':

if form.validate\_on\_submit():

name = form.name.data

form.name.data = ''

return render\_template('user.html', name=name)

else:

return render\_template('login.html', form=NameForm())

### Flash消息 （相当于toast）

@app.route('/')

def index():

flash('Oops') 相当于toast内容

return render\_template('index.html')

index.html

{% for message in get\_flashed\_messages() %｝ 相当于Bootstrap中的alert

<div class="alert alert-warning">

<button type="button" class="close" data-dismiss="alert">...</button>

{{ message }}

</div>

{% endfor %}

## 数据库

抽象层：也称为对象关系映射ORM (Object-Relational Mapper)或对象文档映射(Object-Document Mapper, ODM),在用户不知觉的情况下把高层的面向对象操作转换成低层的数据库指令

SQLAlchemy是一个很强大的关系型数据库框架，支持多种数据库后台。SQLAlchemy提供了高层ORM,也提供了使用数据库原生SQL的低层功能

(env) $ pip install flask-sqlalchemy

若出现no module mysqldb,解决办法如下：

在非 Virtualenv 状态下运行 sudo apt-get install libmysqlclient-dev

然后在(env)$pip install mysql-python

MySQL mysql://user:password@hostname/db

Postgres postgresql://user:password@hostname/db

SQLite sqlite:///absolute/path/to/database

from flask\_sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.pathdirname(\_\_file\_\_))

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///' + os.path.join(basedir, 'data.sqlite')

app.config['SQLALCHEMY\_COMMIT\_ON\_TEARDOWN'] = True

db = SQLAlchemy(app) 表示程序使用的数据库

模型 （类属性对应数据库表中的列）

SQLAlchemy列类型和列选项，见p48

SQLAlchemy关系选项见p49

class Role(db.Model):

\_\_tablename\_\_ = 'roles'

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(64), unique=True)

users = db.relationship('User', backref='role')

class User(db.Model):

\_\_tablename\_\_ = 'users'

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(64), unique=True, index=True)

role\_id = db.Column(db.Integer, db.ForeignKey('roles.id'))

数据库操作

(env)$python hello.py shell 以shell方式启动

>>> from hello import db

>>> db.create\_all() 创建表

db.create\_all()

插入行

admin\_role = Role(name='Admin')

user\_role = Role(name='User')

user\_john = User(username='john', role=admin\_role)

通过数据库会话管理对数据库所做的改动

数据库会话也称为事务，数据库会话能保证数据库的一致性

db.session.add\_all([admin\_role, user\_role, user\_john])

db.session.commit()

admin\_role.name = 'Administrator' 修改行

db.session.add(admin\_role)

db.session.commit()

db.session.delete(user\_role) 删除行

db.session.commit()

查询

Flask-SQLAlchemy为每个模型类都提供了query对象

Role.query.all()

User.query.filter\_by(role=user\_role).all()

Role.query.filter\_by(name='User').all()

## 电子邮件

(env) $pip install flask-mail

Flask-Mail连接到SMTP服务器，并把邮件交给这个服务器发送

(env) $ export MAIL\_USERNAME=<Gmail username>

(env) $ export MAIL\_PASSWORD=<Gmail password>

from flask\_mail import Mail, Message

mail = Mail(app)

from threading import Thread

def send\_async\_email(app, msg):

with app.app\_context():

mail.send(msg)

def send\_mail(to, subject, template, \*\*kwargs):

msg = Message(subject, sender, recipients=[to])

msg.body = render\_template(template+'.txt', \*\*kwargs)

msg.html = render\_template(template+'.html', \*\*kwargs)

thr = Thread(target=send\_async\_email, args=[app, msg])

thr.start()

return thr

## 大型程序结构

├── app

│   ├── email.py

│   ├── \_\_init\_\_.py

│   ├── main

│   │   ├── errors.py

│   │   ├── forms.py

│   │   ├── \_\_init\_\_.py

│   │   └── views.py

│   ├── models.py

│   ├── static

│   └── templates

│   ├── 404.html

│   ├── base.html

│   ├── login.html

│   └── user.html

├── config.py

├── manage.py

└── requirements.txt

app/main/\_\_init\_\_.py 导入main目录下的views.py（路由）和errors.py(错误处理程序）

from flask import Blueprint

**main = Blueprint('main', \_\_name\_\_)**

from .import views, errors

app/\_\_init\_\_.py

bootstrap = Bootstrap()

mail = Mail()

moment = Moment()

db = SQLAlchemy()

def create\_app(config\_name):

app = Flask(\_\_name\_\_)

bootstrap.init\_app(app)

mail.init\_app(app)

moment.init\_app(app)

db.init\_app(app)

from .main import main as main\_blueprint

app.register\_blueprint(main\_blueprint) 注册Blueprint, 从而由Blueprint管理路由和错误处理

return app

路由views.py

from .import main

**@main.route('/login')**

def login():

Flask会为蓝本中的全部端点加上一个命名空间，所以视图函数login()注册的端点名是main.login, url\_for('main.login')

视图base.html

<li><a href="{{ url\_for('main.login') }}">Home</a></li>

启动脚本manage.py

app = create\_app(os.getenv('FLASK\_CONFIG') or 'default')

manager = Manager(app)

migrate = Migrate(app, db)

manager.add\_command('db', MigrateCommand)

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

manager.run()

(env)$ pip freeze >requirements.txt 记录所有依赖包及版本号

在新的虚拟环境下

(env) $pip install -r requirements.txt

用户认证

图像处理

json序列化

@app.route('/image')

def get\_image():

gray = cv.imread('../../data/timg.jpeg', cv.IMREAD\_GRAYSCALE)

return jsonify(gray.tolist())

@app.route('/img\_html')

def iimg\_html():

return render\_template('image.html')

image.html

var img = $.ajax({

url: "{{ url\_for('get\_image') }}",

success: function( data ) {

var imgData = ctx.createImageData(data[0].length, data.length);

for (let nRow = 0 ; nRow < imgData.height; ++ nRow) {

for (let nCol = 0 ; nCol < imgData.width; ++ nCol) {

let idx = nRow\*imgData.width + nCol;

imgData.data[idx\*4 + 0] = data[nRow][nCol]; // red

imgData.data[idx\*4 + 1] = data[nRow][nCol]; // green

imgData.data[idx\*4 + 2] = data[nRow][nCol]; // blue

imgData.data[idx\*4 + 3] = 255; // alpha

}

}

ctx.putImageData(imgData,0,0);

}

});

Video Streaming with Flask

<https://blog.miguelgrinberg.com/post/video-streaming-with-flask>

<http://flask.pocoo.org/docs/0.12/patterns/streaming/>

stream < - > generator function

Multipart responses consist of a header that includes one of the multipart content types, followed by the parts, separated by a boundary marker and each having its own part specific content type.

教程

Browser -> web server -> WSGI(e.g. CGI，mod\_python, mod\_wsgi) -> web应用程序

WSGI: Web Server Gateway Interface (PythonWeb服务器网关接口)

网关的作用就是在协议之间进行转换。

类似于Java中的"servlet" API,允许不同web服务器和应用框架基于通用API交互的通用规范

Python Paste - WSGI底层工具集. 包括多线程, SSL和 基于Cookies, sessions等的验证(authentication)库

基本实现

接受HTTP请求、解析HTTP请求、发送HTTP响应

hello.py

def application(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

return '<h1>Hello, web!</h1>'

environ：一个包含所有HTTP请求信息的dict对象；

start\_response：一个发送HTTP响应的函数。

Python内置了一个WSGI服务器，这个模块叫wsgiref

server.py 负责启动WSGI服务器

from wsgiref.simple\_server import make\_server

from hello import application

# 创建一个服务器，IP地址为空，端口是8000，处理函数是application:

httpd = make\_server('', 8000, application)

# 开始监听HTTP请求:

httpd.serve\_forever()

中间件

扩展WSGI应用功能性的方法

def log\_environ(handler):

def \_inner(environ, start\_fn):

pprint.pprint(environ)

return handler(environ, start\_fn)

return \_inner

def wrap\_query\_params(handler):

def \_inner(environ, start\_fn):

qs = environ.get('QUERY\_STRING')

environ['QUERY\_PARAMS'] = urlparse.parse\_qs(qs)

return handler(environ, start\_fn)

return \_inner

# Applied from bottom to top on the way in, then top to bottom on the way out

MIDDLEWARES = [wrap\_query\_params,log\_environ,handle\_error]

# wrap\_query\_params(log\_environ(handler))

app = reduce(lambda h, m: m(h), MIDDLEWARES, handler)

Flask

<http://docs.jinkan.org/docs/flask/tutorial/index.html>

<http://docs.jinkan.org/docs/flask/index.html>

Flask 依赖两个外部库： Jinja2 模板引擎和 Werkzeug WSGI 工具集

路由

通过Python的装饰器在内部自动地把URL和函数给关联起来Flask通过Python的装饰器在内部自动地把URL和函数给关联起来

Flask 的 URL 规则基于 Werkzeug 的路由模块。这个模块背后的思想是基于 Apache

\*.html

url: "{{ url\_for('get\_image', filename=filename) }}"

controller.py

@app.route('/api/v1.0/image')

def get\_image():

filename = str(request.args.get("filename", ""))

Request 请求对象

<http://docs.jinkan.org/docs/flask/api.html#flask.request>

req.form 一个包含解析过的从 POST 或 PUT 请求发送的表单对象的 MultiDict

req.args 一个包含解析过的查询字符串（ URL 中问号后的部分）内容的 MultiDict

req.cookies 一个包含请求中传送的所有 cookie 内容的 dict

req.headers 进入请求的标头存为一个类似字典的对象

req.data 如果进入的请求数据是 Flask 不能处理的 mimetype ，数据将作为字符串存于此

req.files 一个包含 POST 和 PUT 请求中上传的文件的 MultiDict 。每个文件存储为 FileStorage 对象。其基本的行为类似你在 Python 中见到的标准文件对象，差异在于这个对象有一个 save() 方法可以把文件存储到文件系统上

req.environ 底层的 WSGI 环境

req.method 当前请求的 HTTP 方法

req.path, .base\_url, .script\_root, .url, .url\_root

req.is\_xhr 当请求由 JavaScript 的 XMLHttpRequest 触发时，该值为 True 。 这只对支持 X-Requested-With 标头并把该标头设置为 XMLHttpRequest 的库奏效。这么做的库有 prototype 、 jQuery 以及 Mochikit 等更多

req.json parsed json data or none

req.max\_content\_length

response 响应对象

Flask 把返回值转换为响应对象的逻辑是这样：

1. 如果返回的是一个合法的响应对象，它会从视图直接返回。
2. 如果返回的是一个字符串（如render\_template()），响应对象会用字符串数据和默认参数创建。（字符串为主体的、状态码为 200 OK(or other)、 MIME 类型是text/html (or other) 的响应对象）
3. 如果返回的是一个元组，且元组中的元素可以提供额外的信息。这样的元组必须是 (response, status, headers) 的形式，且至少包含一个元素。 status 值会覆盖状态代码， headers 可以是一个列表或字典，作为额外的消息标头值。
4. 如果上述条件均不满足， Flask 会假设返回值是一个合法的 WSGI 应用程序，并转换为一个请求对象。

res = make\_response(render\_template(...))

res.set\_cookie('username', 'the username')

res.headers

res.status

res.mimetype

配置

<http://docs.jinkan.org/docs/flask/config.html>

应用会需要某种配置。你可能会需要根据应用环境更改不同的设置，比如切换调试模式、设置密钥、或是别的设定环境的东西。

app = Flask(\_\_name\_\_)

app.config['DEBUG'] = True

app.config['JSON\_AS\_ASCII'] = False

app.config['JSONIFY\_PRETTYPRINT\_REGULAR'] = False

配置文件其实是 Python 文件。只有大写名称的值才会被存储到配置对象中

实例文件夹

操作系统有进程当前工作目录的概念。不幸的是，你在 Web 应用中不能依赖此概念，因为你可能会在相同的进程中运行多个应用。为此，提供了 app.root\_path 属性以获取应用的路径。配合 os.path 模块使用，轻松可达任意文件

在 Flask 0.8 中，引入了 Flask.instance\_path 并提出了“实例文件夹” 的新概念

app = Flask(\_\_name\_\_, instance\_path='/path/to/instance/folder')

如果 instance\_path 参数没有赋值，会使用下面默认的位置:

未安装的模块:

/myapp.py

/instance

已安装的包或模块:

$PREFIX/lib/python2.X/site-packages/myapp

$PREFIX/var/myapp-instance

上传文件

<http://docs.jinkan.org/docs/flask/patterns/fileuploads.html#uploading-files>

<https://pythonhosted.org/Flask-Uploads/>

重定向和错误

redirect(url\_for('login'))

@app.errorhandler(404)

def page\_not\_found(error):

return render\_template('page\_not\_found.html'), 404

默认情况下，错误代码会显示一个黑白的错误页面。如果你要定制错误页面， 可以使用 errorhandler() 装饰器:

会话

from flask import Flask, session