Contents

[<python基础教程> 4](#_Toc461882610)

[模块 4](#_Toc461882611)

[包 5](#_Toc461882612)

[python基本类型 5](#_Toc461882613)

[序列 5](#_Toc461882614)

[通用操作 5](#_Toc461882615)

[列表 6](#_Toc461882616)

[元组 6](#_Toc461882617)

[字符串 6](#_Toc461882618)

[集合:通过列表或元组定义 6](#_Toc461882619)

[字典 7](#_Toc461882620)

[常用操作 7](#_Toc461882621)

[不常用的命令： 8](#_Toc461882622)

[性能比较： 8](#_Toc461882623)

[函数 8](#_Toc461882624)

[异常 9](#_Toc461882625)

[类 9](#_Toc461882626)

[class 继承 10](#_Toc461882627)

[迭代器 11](#_Toc461882628)

[生成器 11](#_Toc461882629)

[文件 12](#_Toc461882630)

[<Python in Practice> 13](#_Toc461882631)

[1. sequence unpacking and map unpacking 13](#_Toc461882632)

[2. locals() & glocal() 13](#_Toc461882633)

[3. @classmethod and @staticmethod的区别: 14](#_Toc461882634)

[5. \_\_slots\_\_ 15](#_Toc461882635)

[6. \_\_init\_\_, \_\_new\_\_, and \_\_call\_\_区别 15](#_Toc461882636)

[7. Exec and eval 16](#_Toc461882637)

[8. Type使用 16](#_Toc461882638)

[9. 面向函数编程：提供的函数 16](#_Toc461882639)

[10. \_\_call\_\_(self) 17](#_Toc461882640)

[<Python编程快速上手—让繁琐工作自动化> 18](#_Toc461882641)

[基础知识 18](#_Toc461882642)

[<Flask Web开发> 18](#_Toc461882643)

[tutorial 18](#_Toc461882644)

[视图 20](#_Toc461882645)

[模板 20](#_Toc461882646)

[集成Twitter Bootstrap 22](#_Toc461882647)

[自定义错误页面 23](#_Toc461882648)

[链接 23](#_Toc461882649)

[静态文件 23](#_Toc461882650)

[使用Flask-Moment本地化日期和时间 23](#_Toc461882651)

[Web表单 24](#_Toc461882652)

[Flash消息 （相当于toast） 25](#_Toc461882653)

[数据库 25](#_Toc461882654)

[电子邮件 27](#_Toc461882655)

[大型程序结构 27](#_Toc461882656)

# <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)”

## 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就使用引用。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

# <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

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