Python学习笔记（二）

常见内建模块

1datetime：处理时间和日期

1.1获取当前日期和时间

>>> from datetime import datetime

>>> now = datetime.now() *# 获取当前datetime*

>>> **print**(now)

2015-05-18 16:28:07.198690

>>> **print**(type(now))

<**class 'datetime.datetime'>**

注意到datetime是模块，datetime模块还包含一个datetime类，通过from datetime import datetime导入的才是datetime这个类。

如果仅导入import datetime，则必须引用全名datetime.datetime。

datetime.now()返回当前日期和时间，其类型是datetime

1.2获取指定日期和时间

要指定某个日期和时间，我们直接用参数构造一个datetime：

>>> **from** datetime **import** datetime

>>> dt = datetime(2015, 4, 19, 12, 20) *# 用指定日期时间创建datetime*

>>> print(dt)

2015-04-19 12:20:00

1.3datetime转换为timestamp

在计算机中，时间实际上是用数字表示的。我们把1970年1月1日 00:00:00 UTC+00:00时区的时刻称为epoch time，记为0（1970年以前的时间timestamp为负数），当前时间就是相对于epoch time的秒数，称为timestamp。

你可以认为：

timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00

对应的北京时间是：

timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00

可见timestamp的值与时区毫无关系，因为timestamp一旦确定，其UTC时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以timestamp表示的，因为全球各地的计算机在任意时刻的timestamp都是完全相同的（假定时间已校准）。

把一个datetime类型转换为timestamp只需要简单调用timestamp()方法：

>>> **from** datetime **import** datetime

>>> dt = datetime(2015, 4, 19, 12, 20) *# 用指定日期时间创建datetime*

>>> dt.timestamp() *# 把datetime转换为timestamp*

1429417200.0

注意Python的timestamp是一个浮点数。如果有小数位，小数位表示毫秒数。

某些编程语言（如Java和JavaScript）的timestamp使用整数表示毫秒数，这种情况下只需要把timestamp除以1000就得到Python的浮点表示方法。

1.4timestamp转换为datetime

要把timestamp转换为datetime，使用datetime提供的fromtimestamp()方法：

>>> **from** datetime **import** datetime

>>> t = 1429417200.0

>>> print(datetime.fromtimestamp(t))

2015-04-19 12:20:00

imestamp也可以直接被转换到UTC标准时区的时间：

>>> **from** datetime **import** datetime

>>> t = 1429417200.0

>>> print(datetime.fromtimestamp(t)) *# 本地时间*

2015-04-19 12:20:00

>>> print(datetime.utcfromtimestamp(t)) *# UTC时间*

2015-04-19 04:20:00

1.5str转换为datetime

很多时候，用户输入的日期和时间是字符串，要处理日期和时间，首先必须把str转换为datetime。转换方法是通过datetime.strptime()实现，需要一个日期和时间的格式化字符串：

>>> **from** datetime **import** datetime

>>> cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')

>>> print(cday)

2015-06-01 18:19:59

1.6datetime转换为str

如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过strftime()实现的，同样需要一个日期和时间的格式化字符串：

>>> **from** datetime **import** datetime

>>> now = datetime.now()

>>> print(now.strftime('%a, %b %d %H:%M'))

Mon, May 05 16:28

1.7datetime加减

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用+和-运算符，不过需要导入timedelta这个类：

>>> **from** datetime **import** datetime, timedelta

>>> now = datetime.now()

>>> now

datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)

>>> now + timedelta(hours=10)

datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)

>>> now - timedelta(days=1)

datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)

>>> now + timedelta(days=2, hours=12)

datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)

1.8本地时间转换为UTC时间

本地时间是指系统设定时区的时间，例如北京时间是UTC+8:00时区的时间，而UTC时间指UTC+0:00时区的时间。

一个datetime类型有一个时区属性tzinfo，但是默认为None，所以无法区分这个datetime到底是哪个时区，除非强行给datetime设置一个时区：

>>> **from** datetime **import** datetime, timedelta, timezone

>>> tz\_utc\_8 = timezone(timedelta(hours=8)) *# 创建时区UTC+8:00*

>>> now = datetime.now()

>>> now

datetime.datetime(2015, 5, 18, 17, 2, 10, 871012)

>>> dt = now.replace(tzinfo=tz\_utc\_8) *# 强制设置为UTC+8:00*

>>> dt

datetime.datetime(2015, 5, 18, 17, 2, 10, 871012, tzinfo=datetime.timezone(datetime.timedelta(0, 28800)))

2collections:许多有用的集合类

1.1Namedtuple：namedtuple是一个函数，它用来创建一个自定义的tuple对象，并且规定了tuple元素的个数，并可以用属性而不是索引来引用tuple的某个元素。

这样一来，我们用namedtuple可以很方便地定义一种数据类型，它具备tuple的不变性，又可以根据属性来引用，使用十分方便。

*# namedtuple('名称', [属性list]):*

>>> **from** collections **import** namedtuple

>>> Point = namedtuple('Point', ['x', 'y'])

>>> p = Point(1, 2)

>>> p.x

1

>>> p.y

2

如果要用坐标和半径表示一个圆，也可以用namedtuple定义：

Circle = namedtuple('Circle', ['x', 'y', 'r'])

1.2deque

使用list存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为list是线性存储，数据量大的时候，插入和删除效率很低。

deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

>>> **from** collections **import** deque

>>> q = deque(['a', 'b', 'c'])

>>> q.append('x')

>>> q.appendleft('y')

>>> q

deque(['y', 'a', 'b', 'c', 'x'])

deque除了实现list的append()和pop()外，还支持appendleft()和popleft()，这样就可以非常高效地往头部添加或删除元素。

1.3defaultdict

使用dict时，如果引用的Key不存在，就会抛出KeyError。如果希望key不存在时，返回一个默认值，就可以用defaultdict：

>>> **from** collections **import** defaultdict

>>> dd = defaultdict(**lambda**: 'N/A')

>>> dd['key1'] = 'abc'

>>> dd['key1'] *# key1存在*

'abc'

>>> dd['key2'] *# key2不存在，返回默认值*

'N/A'

注意默认值是调用函数返回的，而函数在创建defaultdict对象时传入

1.4OrderedDict

使用dict时，Key是无序的。在对dict做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用OrderedDict：

>>> **from** collections **import** OrderedDict

>>> d = dict([('a', 1), ('b', 2), ('c', 3)])

>>> d *# dict的Key是无序的*

{'a': 1, 'c': 3, 'b': 2}

>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

>>> od *# OrderedDict的Key是有序的*

OrderedDict([('a', 1), ('b', 2), ('c', 3)])

注意，OrderedDict的Key会按照插入的顺序排列，不是Key本身排序：

>>> od = OrderedDict()

>>> od['z'] = 1

>>> od['y'] = 2

>>> od['x'] = 3

>>> list(od.keys()) *# 按照插入的Key的顺序返回*

['z', 'y', 'x']

OrderedDict可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key：

**from** collections **import** OrderedDict

**class LastUpdatedOrderedDict(OrderedDict):**

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

super(LastUpdatedOrderedDict, self).\_\_init\_\_()

self.\_capacity = capacity

**def** **\_\_setitem\_\_**(self, key, value):

containsKey = 1 **if** key **in** self **else** 0

**if** len(self) - containsKey >= self.\_capacity:

last = self.popitem(last=False)

print('remove:', last)

**if** containsKey:

**del** self[key]

print('set:', (key, value))

**else**:

print('add:', (key, value))

OrderedDict.\_\_setitem\_\_(self, key, value)

1.5Counter

Counter是一个简单的计数器，例如，统计字符出现的个数：

>>> **from** collections **import** Counter

>>> c = Counter()

>>> **for** ch **in** 'programming':

... c[ch] = c[ch] + 1

...

>>> c

Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})

Counter实际上也是dict的一个子类，上面的结果可以看出，字符'g'、'm'、'r'各出现了两次，其他字符各出现了一次。

3base64:用64个字符来表示任意二进制数据的方法

Python内置的base64可以直接进行base64的编解码：

>>> **import** base64

>>> base64.b64encode(b'binary\x00string')

b'YmluYXJ5AHN0cmluZw=='

>>> base64.b64decode(b'YmluYXJ5AHN0cmluZw==')

b'binary\x00string'

4struct: 来解决bytes和其他二进制数据类型的转换

struct的pack函数把任意数据类型变成bytes：

>>> **import** struct

>>> struct.pack('>I', 10240099)

b'\x00\x9c@c'

pack的第一个参数是处理指令，'>I'的意思是：

>表示字节顺序是big-endian，也就是网络序，I表示4字节无符号整数。

后面的参数个数要和处理指令一致。

unpack把bytes变成相应的数据类型：

>>> struct.unpack('>IH', b'\xf0\xf0\xf0\xf0\x80\x80')

(4042322160, 32896)

根据>IH的说明，后面的bytes依次变为I：4字节无符号整数和H：2字节无符号整数。

5hashlib:提供常见的摘要算啊，如MD5,SHA1

摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）

1.1MD5

**import** hashlib

md5 = hashlib.md5()

md5.update('how to use md5 in python hashlib?'.encode('utf-8'))

print(md5.hexdigest())

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示

采用MD5存储口令是否就一定安全呢？也不一定。假设你是一个黑客，已经拿到了存储MD5口令的数据库，如何通过MD5反推用户的明文口令呢？暴力破解费事费力，真正的黑客不会这么干。

考虑这么个情况，很多用户喜欢用123456，888888，password这些简单的口令，于是，黑客可以事先计算出这些常用口令的MD5值，得到一个反推表：

'e10adc3949ba59abbe56e057f20f883e': '123456'

'21218cca77804d2ba1922c33e0151105': '888888'

'5f4dcc3b5aa765d61d8327deb882cf99': 'password'

这样，无需破解，只需要对比数据库的MD5，黑客就获得了使用常用口令的用户账号。

对于用户来讲，当然不要使用过于简单的口令。但是，我们能否在程序设计上对简单口令加强保护呢？

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

**def** **calc\_md5**(password):

**return** get\_md5(password + 'the-Salt')

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

但是如果有两个用户都使用了相同的简单口令比如123456，在数据库中，将存储两条相同的MD5值，这说明这两个用户的口令是一样的。有没有办法让使用相同口令的用户存储不同的MD5呢？

如果假定用户无法修改登录名，就可以通过把登录名作为Salt的一部分来计算MD5，从而实现相同口令的用户也存储不同的MD5。

1.2SHA1

import hashlib

sha1 = hashlib.sha1()

sha1.**update**('how to use sha1 in '.encode('utf-8'))

sha1.**update**('python hashlib?'.encode('utf-8'))

print(sha1.hexdigest())

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

6hmac: 通过一个标准算法，在计算哈希的过程中，把key混入计算过程中。

加salt的哈希就是：计算一段message的哈希时，根据不通口令计算出不同的哈希。要验证哈希值，必须同时提供正确的口令。

>>> **import** hmac

>>> message = b'Hello, world!'

>>> key = b'secret'

>>> h = hmac.new(key, message, digestmod='MD5')

>>> *# 如果消息很长，可以多次调用h.update(msg)*

>>> h.hexdigest()

'fa4ee7d173f2d97ee79022d1a7355bcf'

可见使用hmac和普通hash算法非常类似。hmac输出的长度和原始哈希算法的长度一致。需要注意传入的key和message都是bytes类型，str类型需要首先编码为bytes

7itertools: 用于操作迭代对象的函数

1.1几个无限的迭代器

Count（）：因为count()会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按Ctrl+C退出。

>>> **import** itertools

>>> natuals = itertools.count(1)

>>> **for** n **in** natuals:

... print(n)

...

1

2

3

...

Cycle():会把传入的一个序列无限重复下去

>>> **import** itertools

>>> cs = itertools.cycle('ABC') *# 注意字符串也是序列的一种*

>>> **for** c **in** cs:

... print(c)

...

'A'

'B'

'C'

'A'

'B'

'C'

...

repeat()负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

>>> ns = itertools.repeat('A', 3)

>>> **for** n **in** ns:

... print(n)

...

A

A

A

无限序列只有在for迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过takewhile()等函数根据条件判断来截取出一个有限的序列：

>>> natuals = itertools.count(1)

>>> ns = itertools.takewhile(**lambda** x: x <= 10, natuals)

>>> list(ns)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

1.2迭代器操作函数

chain()可以把一组迭代对象串联起来，形成一个更大的迭代器：

>>> **for** c **in** itertools.chain('ABC', 'XYZ'):

... print(c)

*# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'*

groupby()

groupby()把迭代器中相邻的重复元素挑出来放在一起：

>>> **for** key, group **in** itertools.groupby('AAABBBCCAAA'):

... print(key, list(group))

...

A ['A', 'A', 'A']

B ['B', 'B', 'B']

C ['C', 'C']

A ['A', 'A', 'A']

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

>>> **for** key, group **in** itertools.groupby('AaaBBbcCAAa', **lambda** c: c.upper()):

... print(key, list(group))

...

A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']

8contextlib

任何对象，只要正确实现了上下文管理，就可以用于with语句

实现上下文管理, Python的标准库contextlib提供了简单的写法:

**from** contextlib **import** contextmanager

**class Query(object):**

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

self.name = name

**def** **query**(self):

print('Query info about %s...' % self.name)

@contextmanager

**def** **create\_query**(name):

print('Begin')

q = Query(name)

**yield** q

print('End')

@contextmanager这个decorator接受一个generator，用yield语句把with ... as var把变量输出出去，然后，with语句就可以正常地工作了：

**with** create\_query('Bob') **as** q:

q.query()

很多时候，我们希望在某段代码执行前后自动执行特定代码，也可以用@contextmanager实现。例如：

@contextmanager

**def** **tag**(name):

print("<%s>" % name)

**yield**

print("</%s>" % name)

**with** tag("h1"):

print("hello")

print("world")

上述代码执行结果为：

<h1>

hello

world

</h1>

代码的执行顺序是：

1. with语句首先执行yield之前的语句，因此打印出<h1>；
2. yield调用会执行with语句内部的所有语句，因此打印出hello和world；
3. 最后执行yield之后的语句，打印出</h1>。

因此，@contextmanager让我们通过编写generator来简化上下文管理。

@closing

如果一个对象没有实现上下文，我们就不能把它用于with语句。这个时候，可以用closing()来把该对象变为上下文对象。例如，用with语句使用urlopen()：

**from** contextlib **import** closing

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

**with** closing(urlopen('https://www.python.org')) **as** page:

**for** line **in** page:

print(line)

closing也是一个经过@contextmanager装饰的generator，这个generator编写起来其实非常简单：

@contextmanager

**def** **closing**(thing):

**try**:

**yield** thing

**finally**:

thing.close()

它的作用就是把任意对象变为上下文对象，并支持with语句。

9urllib

urllib的request模块可以非常方便地抓取URL内容，也就是发送一个GET请求到指定的页面，然后返回HTTP的响应：

例如，对豆瓣的一个URLhttps://api.douban.com/v2/book/2129650进行抓取，并返回响应：

**from** urllib **import** request

**with** request.urlopen('https://api.douban.com/v2/book/2129650') **as** f:

data = f.read()

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', data.decode('utf-8'))

可以看到HTTP响应的头和JSON数据：

Status: 200 OK

Server: nginx

Date: Tue, 26 May 2015 10:02:27 GMT

Content-Type: application/json; char**set**=utf-8

Content-Length: 2049

**Connection**: **close**

Expires: Sun, 1 Jan 2006 01:00:00 GMT

**Pragma**: **no**-cache

Cache-Control: must-revalidate, **no**-cache, private

X-DAE-Node: pidl1

Data: {"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},"subtitle":"","author":["廖雪峰编著"],"pubdate":"2007-6",...}

如果我们要想模拟浏览器发送GET请求，就需要使用Request对象，通过往Request对象添加HTTP头，我们就可以把请求伪装成浏览器。例如，模拟iPhone 6去请求豆瓣首页：

**from** urllib **import** request

req = request.Request('http://www.douban.com/')

req.add\_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8\_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')

**with** request.urlopen(req) **as** f:

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', f.read().decode('utf-8'))

这样豆瓣会返回适合iPhone的移动版网页：

...

<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0">

<meta name="format-detection" content="telephone=no">

<link rel="apple-touch-icon" sizes="57x57" href="http://img4.douban.com/pics/cardkit/launcher/57.png" />

...

如果要以POST发送一个请求，只需要把参数data以bytes形式传入。

我们模拟一个微博登录，先读取登录的邮箱和口令，然后按照weibo.cn的登录页的格式以username=xxx&password=xxx的编码传入：

**from** urllib **import** request, parse

print('Login to weibo.cn...')

email = input('Email: ')

passwd = input('Password: ')

login\_data = parse.urlencode([

('username', email),

('password', passwd),

('entry', 'mweibo'),

('client\_id', ''),

('savestate', '1'),

('ec', ''),

('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=mweibo&r=http%3A%2F%2Fm.weibo.cn%2F')

])

req = request.Request('https://passport.weibo.cn/sso/login')

req.add\_header('Origin', 'https://passport.weibo.cn')

req.add\_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8\_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')

req.add\_header('Referer', 'https://passport.weibo.cn/signin/login?entry=mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')

**with** request.urlopen(req, data=login\_data.encode('utf-8')) **as** f:

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', f.read().decode('utf-8')

urllib

阅读: 201904

urllib提供了一系列用于操作URL的功能。

Get

urllib的request模块可以非常方便地抓取URL内容，也就是发送一个GET请求到指定的页面，然后返回HTTP的响应：

例如，对豆瓣的一个URLhttps://api.douban.com/v2/book/2129650进行抓取，并返回响应：

**from** urllib **import** request

**with** request.urlopen('https://api.douban.com/v2/book/2129650') **as** f:

data = f.read()

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', data.decode('utf-8'))

可以看到HTTP响应的头和JSON数据：

Status: 200 OK

Server: nginx

Date: Tue, 26 May 2015 10:02:27 GMT

Content-Type: application/json; char**set**=utf-8

Content-Length: 2049

**Connection**: **close**

Expires: Sun, 1 Jan 2006 01:00:00 GMT

**Pragma**: **no**-cache

Cache-Control: must-revalidate, **no**-cache, private

X-DAE-Node: pidl1

Data: {"rating":{"max":10,"numRaters":16,"average":"7.4","min":0},"subtitle":"","author":["廖雪峰编著"],"pubdate":"2007-6",...}

如果我们要想模拟浏览器发送GET请求，就需要使用Request对象，通过往Request对象添加HTTP头，我们就可以把请求伪装成浏览器。例如，模拟iPhone 6去请求豆瓣首页：

**from** urllib **import** request

req = request.Request('http://www.douban.com/')

req.add\_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8\_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')

**with** request.urlopen(req) **as** f:

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', f.read().decode('utf-8'))

这样豆瓣会返回适合iPhone的移动版网页：

...

<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0">

<meta name="format-detection" content="telephone=no">

<link rel="apple-touch-icon" sizes="57x57" href="http://img4.douban.com/pics/cardkit/launcher/57.png" />

...

Post

如果要以POST发送一个请求，只需要把参数data以bytes形式传入。

我们模拟一个微博登录，先读取登录的邮箱和口令，然后按照weibo.cn的登录页的格式以username=xxx&password=xxx的编码传入：

**from** urllib **import** request, parse

print('Login to weibo.cn...')

email = input('Email: ')

passwd = input('Password: ')

login\_data = parse.urlencode([

('username', email),

('password', passwd),

('entry', 'mweibo'),

('client\_id', ''),

('savestate', '1'),

('ec', ''),

('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=mweibo&r=http%3A%2F%2Fm.weibo.cn%2F')

])

req = request.Request('https://passport.weibo.cn/sso/login')

req.add\_header('Origin', 'https://passport.weibo.cn')

req.add\_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8\_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')

req.add\_header('Referer', 'https://passport.weibo.cn/signin/login?entry=mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')

**with** request.urlopen(req, data=login\_data.encode('utf-8')) **as** f:

print('Status:', f.status, f.reason)

**for** k, v **in** f.getheaders():

print('%s: %s' % (k, v))

print('Data:', f.read().decode('utf-8'))

如果登录成功，我们获得的响应如下：

Status: 200 OK

Server: nginx/1.2.0

...

**Set**-Cookie: SSOLoginState=1432620126; path=/; domain=weibo.cn

...

Data: {"retcode":20000000,"msg":"","data":{...,"uid":"1658384301"}}

如果登录失败，我们获得的响应如下：

...

Data: {"retcode":50011015,"msg":"\u7528\u6237\u540d\u6216\u5bc6\u7801\u9519\u8bef","data":{"username":"example@python.org","errline":536}}

如果登录成功，我们获得的响应如下：

Status: 200 OK

Server: nginx/1.2.0

...

**Set**-Cookie: SSOLoginState=1432620126; path=/; domain=weibo.cn

...

Data: {"retcode":20000000,"msg":"","data":{...,"uid":"1658384301"}}

如果登录失败，我们获得的响应如下：

...

Data: {"retcode":50011015,"msg":"\u7528\u6237\u540d\u6216\u5bc6\u7801\u9519\u8bef","data":{"username":"example@python.org","errline":536}}

Handler

如果还需要更复杂的控制，比如通过一个Proxy去访问网站，我们需要利用ProxyHandler来处理，示例代码如下：

proxy\_**handler** = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})

proxy\_auth\_handler = urllib.request.ProxyBasicAuthHandler()

proxy\_auth\_handler.add\_password('realm', 'host', 'username', 'password')

opener = urllib.request.build\_opener(proxy\_handler, proxy\_auth\_handler)

**with** opener.**open**('http://www.example.com/login.html') **as** f:

pass

10XML

DOM vs SAX

操作XML有两种方法：DOM和SAX。DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是我们需要自己处理事件。

正常情况下，优先考虑SAX，因为DOM实在太占内存。

在Python中使用SAX解析XML非常简洁，通常我们关心的事件是start\_element，end\_element和char\_data，准备好这3个函数，然后就可以解析xml了。

举个例子，当SAX解析器读到一个节点时：

<a href="/">python</a>

会产生3个事件：

1. start\_element事件，在读取<a href="/">时；
2. char\_data事件，在读取python时；
3. end\_element事件，在读取</a>时。

用代码实验一下：

**from** xml.parsers.expat **import** ParserCreate

**class DefaultSaxHandler(object):**

**def** **start\_element**(self, name, attrs):

print('sax:start\_element: %s, attrs: %s' % (name, str(attrs)))

**def** **end\_element**(self, name):

print('sax:end\_element: %s' % name)

**def** **char\_data**(self, text):

print('sax:char\_data: %s' % text)

xml = r'''<?xml version="1.0"?>

<ol>

<li><a href="/python">Python</a></li>

<li><a href="/ruby">Ruby</a></li>

</ol>

'''

handler = DefaultSaxHandler()

parser = ParserCreate()

parser.StartElementHandler = handler.start\_element

parser.EndElementHandler = handler.end\_element

parser.CharacterDataHandler = handler.char\_data

parser.Parse(xml)

需要注意的是读取一大段字符串时，CharacterDataHandler可能被多次调用，所以需要自己保存起来，在EndElementHandler里面再合并。

除了解析XML外，如何生成XML呢？99%的情况下需要生成的XML结构都是非常简单的，因此，最简单也是最有效的生成XML的方法是拼接字符串：

L = []

L.append(r'<?xml version="1.0"?>')

L.append(r'<root>')

L.append(encode('some & data'))

L.append(r'</root>')

**return** ''.join(L)

11HTMLparser: 非常方便地解析HTML

好在Python提供了HTMLParser来非常方便地解析HTML，只需简单几行代码：

**from** html.parser **import** HTMLParser

**from** html.entities **import** name2codepoint

**class MyHTMLParser(HTMLParser):**

**def** **handle\_starttag**(self, tag, attrs):

print('<%s>' % tag)

**def** **handle\_endtag**(self, tag):

print('</%s>' % tag)

**def** **handle\_startendtag**(self, tag, attrs):

print('<%s/>' % tag)

**def** **handle\_data**(self, data):

print(data)

**def** **handle\_comment**(self, data):

print('<!--', data, '-->')

**def** **handle\_entityref**(self, name):

print('&%s;' % name)

**def** **handle\_charref**(self, name):

print('&#%s;' % name)

parser = MyHTMLParser()

parser.feed('''<html>

<head></head>

<body>

<!-- test html parser -->

<p>Some <a href=\"#\">html</a> HTML&nbsp;tutorial...<br>END</p>

</body></html>''')

feed()方法可以多次调用，也就是不一定一次把整个HTML字符串都塞进去，可以一部分一部分塞进去。

特殊字符有两种，一种是英文表示的&nbsp;，一种是数字表示的&#1234;，这两种字符都可以通过Parser解析出来。

常见第三方模块

1Pillow

PIL：Python Imaging Library，已经是Python平台事实上的图像处理标准库了。PIL功能非常强大，但API却非常简单易用。

由于PIL仅支持到Python 2.7，加上年久失修，于是一群志愿者在PIL的基础上创建了兼容的版本，名字叫[Pillow](https://github.com/python-pillow/Pillow)，支持最新Python 3.x，又加入了许多新特性，因此，我们可以直接安装使用Pillow

1.1操作图像

来看看最常见的图像缩放操作，只需三四行代码：

from PIL import Image

*# 打开一个jpg图像文件，注意是当前路径:*

im = Image.open('test.jpg')

*# 获得图像尺寸:*

w, h = im.size

**print**('Original image size: %sx%s' % (w, h))

*# 缩放到50%:*

im.thumbnail((w*//2, h//2))*

**print**('Resize image to: %sx%s' % (w*//2, h//2))*

*# 把缩放后的图像用jpeg格式保存:*

im.save('thumbnail.jpg', 'jpeg')

其他功能如切片、旋转、滤镜、输出文字、调色板等一应俱全。

比如，模糊效果也只需几行代码：

**from** PIL **import** Image, ImageFilter

# 打开一个jpg图像文件，注意是当前路径:

im = Image.open('test.jpg')

# 应用模糊滤镜:

im2 = im.filter(ImageFilter.BLUR)

im2.save('blur.jpg', 'jpeg')

PIL的ImageDraw提供了一系列绘图方法，让我们可以直接绘图。比如要生成字母验证码图片：

**from** PIL **import** Image, ImageDraw, ImageFont, ImageFilter

**import** random

# 随机字母:

**def** **rndChar**():

**return** chr(random.randint(65, 90))

# 随机颜色1:

**def** **rndColor**():

**return** (random.randint(64, 255), random.randint(64, 255), random.randint(64, 255))

# 随机颜色2:

**def** **rndColor2**():

**return** (random.randint(32, 127), random.randint(32, 127), random.randint(32, 127))

# 240 x 60:

width = 60 \* 4

height = 60

image = Image.new('RGB', (width, height), (255, 255, 255))

# 创建Font对象:

font = ImageFont.truetype('Arial.ttf', 36)

# 创建Draw对象:

draw = ImageDraw.Draw(image)

# 填充每个像素:

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

**for** y **in** range(height):

draw.point((x, y), fill=rndColor())

# 输出文字:

**for** t **in** range(4):

draw.text((60 \* t + 10, 10), rndChar(), font=font, fill=rndColor2())

# 模糊:

image = image.filter(ImageFilter.BLUR)

image.save('code.jpg', 'jpeg')

我们用随机颜色填充背景，再画上文字，最后对图像进行模糊，得到验证码图片

2requests

要通过GET访问一个页面，只需要几行代码：

>>> import requests

>>> r = requests.get('https://www.douban.com/') # 豆瓣首页

>>> r.status\_code

200

>>> r.text

r.text

'**<!DOCTYPE HTML>**\n<html>\n<head>\n<meta name="description" content="提供图书、电影、音乐唱片的推荐、评论和...'

对于带参数的URL，传入一个dict作为params参数：

>>> r = requests.get('https://www.douban.com/search', params={'q': 'python', 'cat': '1001'})

>>> r.url *# 实际请求的URL*

'https://www.douban.com/search?q=python&cat=1001'

requests自动检测编码，可以使用encoding属性查看：

>>> r.encoding

'utf-8'

无论响应是文本还是二进制内容，我们都可以用content属性获得bytes对象：

>>> r.content

b'**<!DOCTYPE html>**\n<html>\n<head>\n<meta http-equiv="Content-Type" content="text/html; charset=utf-8">\n...'

requests的方便之处还在于，对于特定类型的响应，例如JSON，可以直接获取：

>>> r = requests.get('https://query.yahooapis.com/v1/public/yql?q=**select**%20\*%20**from**%20weather.forecast%20**where**%20woeid%20%3D%202151330&format=json')

>>> r.json()

{'query': {'count': 1, 'created': '2017-11-17T07:14:12Z', ...

需要传入HTTP Header时，我们传入一个dict作为headers参数：

>>> r = requests.get('https://www.douban.com/', headers={'User-Agent': 'Mozilla/5.0 (iPhone; CPU iPhone OS 11\_0 like Mac OS X) AppleWebKit'})

>>> r.text

'**<!DOCTYPE html>**\n<html>\n<head>\n<meta charset="UTF-8">\n <title>豆瓣(手机版)</title>...'

要发送POST请求，只需要把get()方法变成post()，然后传入data参数作为POST请求的数据：

>>> r = requests.post('https://accounts.douban.com/login', data={'form\_email': 'abc@example.com', 'form\_password': '123456'})

requests默认使用application/x-www-form-urlencoded对POST数据编码。如果要传递JSON数据，可以直接传入json参数：

params = {'key': 'value'}

r = requests.post(url, json=params) # 内部自动序列化为JSON

类似的，上传文件需要更复杂的编码格式，但是requests把它简化成files参数：

>>> upload\_files = {'file': open('report.xls', 'rb')}

>>> r = requests.post(url, files=upload\_files)

在读取文件时，注意务必使用'rb'即二进制模式读取，这样获取的bytes长度才是文件的长度。

把post()方法替换为put()，delete()等，就可以以PUT或DELETE方式请求资源。

除了能轻松获取响应内容外，requests对获取HTTP响应的其他信息也非常简单。例如，获取响应头：

>>> r.headers

{Content-Type': 'text/html; char**set**=utf-8', 'Transfer-Encoding': 'chunked', 'Content-Encoding': 'gzip', ...}

>>> r.headers['Content-Type']

'text/html; char**set**=utf-8'

requests对Cookie做了特殊处理，使得我们不必解析Cookie就可以轻松获取指定的Cookie：

>>> r.cookies['ts']

'example\_cookie\_12345'

要在请求中传入Cookie，只需准备一个dict传入cookies参数：

>>> cs = {'token': '12345', 'status': 'working'}

>>> r = requests.get(url, cookies=cs)

最后，要指定超时，传入以秒为单位的timeout参数：

>>> r = requests.get(url, timeout=2.5) *# 2.5秒后超时*

3chardet

当我们拿到一个bytes时，就可以对其检测编码。用chardet检测编码，只需要一行代码：

>>> chardet.detect(b'Hello, world!')

{'encoding': 'ascii', 'confidence': 1.0, 'language': ''}

检测出的编码是ascii，注意到还有个confidence字段，表示检测的概率是1.0（即100%）。

我们来试试检测GBK编码的中文：

>>> data = '离离原上草，一岁一枯荣'.encode('gbk')

>>> chardet.detect(data)

{'encoding': 'GB2312', 'confidence': 0.7407407407407407, 'language': 'Chinese'}

检测的编码是GB2312，注意到GBK是GB2312的超集，两者是同一种编码，检测正确的概率是74%，language字段指出的语言是'Chinese'。

4psutil: 可以通过一两行代码实现系统监控，还可以跨平台使用，支持Linux／UNIX／OSX／Windows等，是系统管理员和运维小伙伴不可或缺的必备模块

1.1cpu信息

我们先来获取CPU的信息：

>>> **import** psutil

>>> psutil.cpu\_count() *# CPU逻辑数量*

4

>>> psutil.cpu\_count(logical=False) *# CPU物理核心*

2

*# 2说明是双核超线程, 4则是4核非超线程*

统计CPU的用户／系统／空闲时间：

>>> psutil.cpu\_times()

scputimes(user=10963.31, nice=0.0, system=5138.67, idle=356102.45)

1.2获取内存信息

使用psutil获取物理内存和交换内存信息，分别使用：

>>> psutil.virtual\_memory()

>>> psutil.swap\_memory()

1.3获取磁盘信息

可以通过psutil获取磁盘分区、磁盘使用率和磁盘IO信息：

>>> psutil.disk\_partitions() *# 磁盘分区信息*

[sdiskpart(device='/dev/disk1', mountpoint='/', fstype='hfs', opts='rw,local,rootfs,dovolfs,journaled,multilabel')]

>>> psutil.disk\_usage('/') *# 磁盘使用情况*

sdiskusage(total=998982549504, used=390880133120, free=607840272384, percent=39.1)

>>> psutil.disk\_io\_counters() *# 磁盘IO*

sdiskio(read\_count=988513, write\_count=274457, read\_bytes=14856830464, write\_bytes=17509420032, read\_time=2228966, write\_time=1618405)

可以看到，磁盘'/'的总容量是998982549504 = 930 GB，使用了39.1%。文件格式是HFS，opts中包含rw表示可读写，journaled表示支持日志。

1.4获取网络信息

Virtualenv

用来为一个应用创建一套“隔离”的Python运行环境。

首先，我们用pip安装virtualenv：

$ pip3 install virtualenv

然后，假定我们要开发一个新的项目，需要一套独立的Python运行环境，可以这么做：

第一步，创建目录：

Mac:~ michael$ mkdir myproject

Mac:~ michael$ cd myproject/

Mac:myproject michael$

第二步，创建一个独立的Python运行环境，命名为venv：

Mac:myproject michael$ virtualenv --no-site-packages venv

Using base prefix '/usr/local/.../Python.framework/Versions/3.4'

New python executable **in** venv/bin/python3.4

Also creating executable **in** venv/bin/python

Installing setuptools, pip, wheel...done.

命令virtualenv就可以创建一个独立的Python运行环境，我们还加上了参数--no-site-packages，这样，已经安装到系统Python环境中的所有第三方包都不会复制过来，这样，我们就得到了一个不带任何第三方包的“干净”的Python运行环境。

新建的Python环境被放到当前目录下的venv目录。有了venv这个Python环境，可以用source进入该环境：

Mac:myproject michael$ source venv/bin/activate

(venv)Mac:myproject michael$

注意到命令提示符变了，有个(venv)前缀，表示当前环境是一个名为venv的Python环境。

下面正常安装各种第三方包，并运行python命令：

(venv)Mac:myproject michael$ pip install jinja2

...

Successfully installed jinja2-2.7.3 markupsafe-0.23

(venv)Mac:myproject michael$ python myapp.py

...

在venv环境下，用pip安装的包都被安装到venv这个环境下，系统Python环境不受任何影响。也就是说，venv环境是专门针对myproject这个应用创建的。

退出当前的venv环境，使用deactivate命令：

(venv)Mac:myproject michael$ deactivate

Mac:myproject michael$

此时就回到了正常的环境，现在pip或python均是在系统Python环境下执行。

完全可以针对每个应用创建独立的Python运行环境，这样就可以对每个应用的Python环境进行隔离。

virtualenv是如何创建“独立”的Python运行环境的呢？原理很简单，就是把系统Python复制一份到virtualenv的环境，用命令source venv/bin/activate进入一个virtualenv环境时，virtualenv会修改相关环境变量，让命令python和pip均指向当前的virtualenv环境。

图形界面：如何使用Tkinter进行GUI编程

第一步是导入Tkinter包的所有内容：

**from** tkinter **import** \*

第二步是从Frame派生一个Application类，这是所有Widget的父容器：

**class Application(Frame):**

**def** **\_\_init\_\_**(self, master=None):

Frame.\_\_init\_\_(self, master)

self.pack()

self.createWidgets()

**def** **createWidgets**(self):

self.helloLabel = Label(self, text='Hello, world!')

self.helloLabel.pack()

self.quitButton = Button(self, text='Quit', command=self.quit)

self.quitButton.pack()

在GUI中，每个Button、Label、输入框等，都是一个Widget。Frame则是可以容纳其他Widget的Widget，所有的Widget组合起来就是一棵树。

pack()方法把Widget加入到父容器中，并实现布局。pack()是最简单的布局，grid()可以实现更复杂的布局。

在createWidgets()方法中，我们创建一个Label和一个Button，当Button被点击时，触发self.quit()使程序退出。

第三步，实例化Application，并启动消息循环：

app = Application()

*# 设置窗口标题:*

app.master.title('Hello World')

*# 主消息循环:*

app.mainloop()

GUI程序的主线程负责监听来自操作系统的消息，并依次处理每一条消息。因此，如果消息处理非常耗时，就需要在新线程中处理。

输入文本

我们再对这个GUI程序改进一下，加入一个文本框，让用户可以输入文本，然后点按钮后，弹出消息对话框。

**from** tkinter **import** \*

**import** tkinter.messagebox **as** messagebox

**class Application(Frame):**

**def** **\_\_init\_\_**(self, master=None):

Frame.\_\_init\_\_(self, master)

self.pack()

self.createWidgets()

**def** **createWidgets**(self):

self.nameInput = Entry(self)

self.nameInput.pack()

self.alertButton = Button(self, text='Hello', command=self.hello)

self.alertButton.pack()

**def** **hello**(self):

name = self.nameInput.get() **or** 'world'

messagebox.showinfo('Message', 'Hello, %s' % name)

app = Application()

*# 设置窗口标题:*

app.master.title('Hello World')

*# 主消息循环:*

app.mainloop()

当用户点击按钮时，触发hello()，通过self.nameInput.get()获得用户输入的文本后，使用tkMessageBox.showinfo()可以弹出消息对话框