collections

376次阅读

---

collections是Python内建的一个集合模块，提供了许多有用的集合类。

namedtuple

我们知道tuple可以表示不变集合，例如，一个点的二维坐标就可以表示成：

```
>>> p = (1, 2)
```

但是，看到(1, 2)，很难看出这个tuple是用来表示一个坐标的。

定义一个class又小题大做了，这时，namedtuple就派上了用场：

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(1, 2)
>>> p.x
1
>>> p.y
2
```

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。

可以验证创建的Point对象是tuple的一种子类：

```
>>> isinstance(p, Point)
True
>>> isinstance(p, tuple)
True
```

类似的，如果要用坐标和半径表示一个圆，也可以用namedtuple定义：

```
# namedtuple('名称', [属性list]):
Circle = namedtuple('Circle', ['x', 'y', 'r'])
```

deque

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。

defaultdict

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。

除了在Key不存在时返回默认值，defaultdict的其他行为跟dict是完全一样的。

OrderedDict

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
>>> 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)
```

Counter

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。

小结

collections模块提供了一些有用的集合类，可以根据需要选用。

---

base64

627次阅读

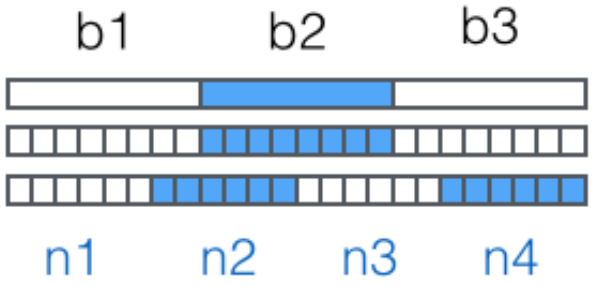
Base64是一种用64个字符来表示任意二进制数据的方法。

用记事本打开exe、jpg、pdf这些文件时，我们都会看到一大堆乱码，因为二进制文件包含很多无法显示和打印的字符，所以，如果要让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

['A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/']

然后，对二进制数据进行处理，每3个字节一组，一共是3x8=24bit，划为4组，每组正好6个bit：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，Base64编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用\x00字节在末尾补足后，再在编码的末尾加上1个或2个=号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的base64可以直接进行base64的编解码：

```
>>> import base64
>>> base64.b64encode('binary\x00string')
'YmluYXJ5AHNOcmluZw=='
>>> base64.b64decode('YmluYXJ5AHNOcmluZw==')
'binary\x00string'
```

由于标准的Base64编码后可能出现字符+和/，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符+和/分别变成-和_：

```
>>> base64.b64encode('i\xb7\x1d\xfb\xef\xff')
'abcd++//'
>>> base64.urlsafe_b64encode('i\xb7\x1d\xfb\xef\xff')
'abcd--__'
>>> base64.urlsafe_b64decode('abcd--__')
'i\xb7\x1d\xfb\xef\xff'
```

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于=字符也可能出现在Base64编码中，但=用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把=去掉：

```
# 标准Base64:
'abcd' -> 'YWJjZA=='
# 自动去掉=:
'abcd' -> 'YWJjZA'
```

去掉=后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上=把Base64字符串的长度变为4的倍数，就可以正常解码了。

请写一个能处理去掉=的base64解码函数：

```
>>> base64.b64decode('YWJjZA==')
'abcd'
>>> base64.b64decode('YWJjZA')
Traceback (most recent call last):
  ...
TypeError: Incorrect padding
>>> safe_b64decode('YWJjZA')
'abcd'
```

小结

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据。

---

struct

630次阅读

准确地讲，Python没有专门处理字节的数据类型。但由于str既是字符串，又可以表示字节，所以，字节数组＝str。而在C语言中，我们可以很方便地用struct、union来处理字节，以及字节和int，float的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的str，你得配合位运算符这么写：

```
>>> n = 10240099
>>> b1 = chr((n & 0xff000000) >> 24)
>>> b2 = chr((n & 0xff0000) >> 16)
>>> b3 = chr((n & 0xff00) >> 8)
>>> b4 = chr(n & 0xff)
>>> s = b1 + b2 + b3 + b4
>>> s
'\x00\x9c@c'
```

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个struct模块来解决str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字符串：

```
>>> import struct
>>> struct.pack('>I', 10240099)
'\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str变成相应的数据类型：

```
>>> struct.unpack('>IH', '\xf0\xf0\xf0\xf0\x80\x80')
(4042322160, 32896)
```

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

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用struct就方便多了。

struct模块定义的数据类型可以参考Python官方文档：

https://docs.python.org/2/library/struct.html#format-characters

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们来用struct分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

```
>>> s = '\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x28\x00\x00\x00\x80\x02\x00\x00\x68\x01\x00\x00\x01\x00\x18\x00'
```

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节：'BM'表示Windows位图，'BA'表示OS/2位图； 一个4字节整数：表示位图大小； 一个4字节整数：保留位，始终为0； 一个4字节整数：实际图像的偏移量； 一个4字节整数：Header的字节数； 一个4字节整数：图像宽度； 一个4字节整数：图像高度； 一个2字节整数：始终为1； 一个2字节整数：颜色数。

所以，组合起来用unpack读取：

```
>>> struct.unpack('<ccIIIIIIHH', s)
('B', 'M', 691256, 0, 54, 40, 640, 360, 1, 24)
```

结果显示，'B'、'M'说明是Windows位图，位图大小为640x360，颜色数为24。

请编写一个bmpinfo.py，可以检查任意文件是否是位图文件，如果是，打印出图片大小和颜色数。

---

hashlib

520次阅读

摘要算法简介

Python的hashlib提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？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）。

举个例子，你写了一篇文章，内容是一个字符串'how to use python hashlib - by Michael'，并附上这篇文章的摘要是'2d73d4f15c0db7f5ecb321b6a65e5d6d'。如果有人篡改了你的文章，并发表为'how to use python hashlib - by Bob'，你可以一下子指出Bob篡改了你的文章，因为根据'how to use python hashlib - by Bob'计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数f()对任意长度的数据data计算出固定长度的摘要digest，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算f(data)很容易，但通过digest反推data却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in python hashlib?')
print md5.hexdigest()
```

计算结果如下：

```
d26a53750bc40b38b65a520292f69306
```

如果数据量很大，可以分块多次调用update()，最后计算的结果是一样的：

```
md5 = hashlib.md5()
md5.update('how to use md5 in ')
md5.update('python hashlib?')
print md5.hexdigest()
```

试试改动一个字母，看看计算的结果是否完全不同。

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。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

```
import hashlib

sha1 = hashlib.sha1()
sha1.update('how to use sha1 in ')
sha1.update('python hashlib?')
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。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法越慢，而且摘要长度更长。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为任何摘要算法都是把无限多的数据集合映射到一个有限的集合中。这种情况称为碰撞，比如Bob试图根据你的摘要反推出一篇文章'how to learn hashlib in python - by Bob'，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

```
name    | password
--------+----------
michael | 123456
bob     | abc999
alice   | alice2008
```

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5：

```
username | password
---------+----------------------------------
michael  | e10adc3949ba59abbe56e057f20f883e
bob      | 878ef96e86145580c38c87f0410ad153
alice    | 99b1c2188db85afee403b1536010c2c9
```

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

练习：根据用户输入的口令，计算出存储在数据库中的MD5口令：

```
def calc_md5(password):
    pass
```

存储MD5的好处是即使运维人员能访问数据库，也无法获知用户的明文口令。

练习：设计一个验证用户登录的函数，根据用户输入的口令是否正确，返回True或False：

```
db = {
    'michael': 'e10adc3949ba59abbe56e057f20f883e',
    'bob': '878ef96e86145580c38c87f0410ad153',
    'alice': '99b1c2188db85afee403b1536010c2c9'
}

def login(user, password):
    pass
```

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。

练习：根据用户输入的登录名和口令模拟用户注册，计算更安全的MD5：

```
db = {}

def register(username, password):
    db[username] = get_md5(password + username + 'the-Salt')
```

然后，根据修改后的MD5算法实现用户登录的验证：

```
def login(username, password):
    pass
```

小结

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

XML

319次阅读

XML虽然比JSON复杂，在Web中应用也不如以前多了，不过仍有很多地方在用，所以，有必要了解如何操作XML。

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，在读取时；
2. char_data事件，在读取python时；
3. end_element事件，在读取时。

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.returns_unicode = True
parser.StartElementHandler = handler.start_element
parser.EndElementHandler = handler.end_element
parser.CharacterDataHandler = handler.char_data
parser.Parse(xml)
```

当设置returns_unicode为True时，返回的所有element名称和char_data都是unicode，处理国际化更方便。

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)
```

如果要生成复杂的XML呢？建议你不要用XML，改成JSON。

小结

解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据。

练习一下解析Yahoo的XML格式的天气预报，获取当天和最近几天的天气：

```
http://weather.yahooapis.com/forecastrss?u=c&w=2151330
```

参数w是城市代码，要查询某个城市代码，可以在weather.yahoo.com搜索城市，浏览器地址栏的URL就包含城市代码。

HTMLParser

337次阅读

如果我们要编写一个搜索引擎，第一步是用爬虫把目标网站的页面抓下来，第二步就是解析该HTML页面，看看里面的内容到底是新闻、图片还是视频。

假设第一步已经完成了，第二步应该如何解析HTML呢？

HTML本质上是XML的子集，但是HTML的语法没有XML那么严格，所以不能用标准的DOM或SAX来解析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 HTMLParser import HTMLParser
from htmlentitydefs 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('<!-- -->')

    def handle_entityref(self, name):
        print('&%s;' % name)

    def handle_charref(self, name):
        print('&#%s;' % name)

parser = MyHTMLParser()
parser.feed('<html><head></head><body><p>Some <a href=\"#\">html</a> 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 ，一种是数字表示的Ӓ，这两种字符都可以通过Parser解析出来。

小结

找一个网页，例如https://www.python.org/events/python-events/，用浏览器查看源码并复制，然后尝试解析一下HTML，输出Python官网发布的会议时间、名称和地点。

常用第三方模块

768次阅读

除了内建的模块外，Python还有大量的第三方模块。

基本上，所有的第三方模块都会在PyPI - the Python Package Index上注册，只要找到对应的模块名字，即可用easy_install或者pip安装。

本章介绍常用的第三方模块。

PIL

239次阅读

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。

安装PIL

在Debian/Ubuntu Linux下直接通过apt安装：

```
$ sudo apt-get install python-imaging
```

Mac和其他版本的Linux可以直接使用easy_install或pip安装，安装前需要把编译环境装好：

```
$ sudo easy_install PIL
```

如果安装失败，根据提示先把缺失的包（比如openjpeg）装上。

Windows平台就去PIL官方网站下载exe安装包。

操作图像

来看看最常见的图像缩放操作，只需三四行代码：

```
import Image

# 打开一个jpg图像文件，注意路径要改成你自己的:
im = Image.open('/Users/michael/test.jpg')
# 获得图像尺寸:
w, h = im.size
# 缩放到50%:
im.thumbnail((w//2, h//2))
# 把缩放后的图像用jpeg格式保存:
im.save('/Users/michael/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：

```
import Image, ImageFilter

im = Image.open('/Users/michael/test.jpg')
im2 = im.filter(ImageFilter.BLUR)
im2.save('/Users/michael/blur.jpg', 'jpeg')
```

效果如下：

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：

```
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，得到验证码图片如下：

如果运行的时候报错：

```
IOError: cannot open resource
```

这是因为PIL无法定位到字体文件的位置，可以根据操作系统提供绝对路径，比如：

```
'/Library/Fonts/Arial.ttf'
```

要详细了解PIL的强大功能，请请参考PIL官方文档：

http://effbot.org/imagingbook/

图形界面

1595次阅读

Python支持多种图形界面的第三方库，包括：

- Tk
- wxWidgets
- Qt
- GTK

等等。

但是Python自带的库是支持Tk的Tkinter，使用Tkinter，无需安装任何包，就可以直接使用。本章简单介绍如何使用Tkinter进行GUI编程。

Tkinter

我们来梳理一下概念：

我们编写的Python代码会调用内置的Tkinter，Tkinter封装了访问Tk的接口；

Tk是一个图形库，支持多个操作系统，使用Tcl语言开发；

Tk会调用操作系统提供的本地GUI接口，完成最终的GUI。

所以，我们的代码只需要调用Tkinter提供的接口就可以了。

第一个GUI程序

使用Tkinter十分简单，我们来编写一个GUI版本的“Hello, world!”。

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。

运行这个GUI程序，可以看到下面的窗口：

点击“Quit”按钮或者窗口的“x”结束程序。

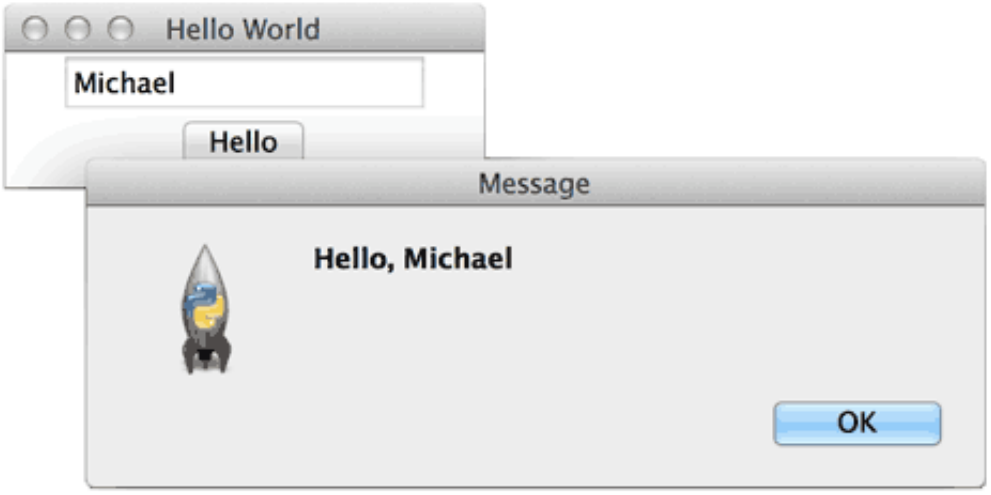
输入文本

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 tkMessageBox

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'
        tkMessageBox.showinfo('Message', 'Hello, %s' % name)
```

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。

程序运行结果如下：



小结

Python内置的Tkinter可以满足基本的GUI程序的要求，如果是非常复杂的GUI程序，建议用操作系统原生支持的语言和库来编写。

源码参考：https://github.com/michaelliao/learn-python/tree/master/gui

---

网络编程

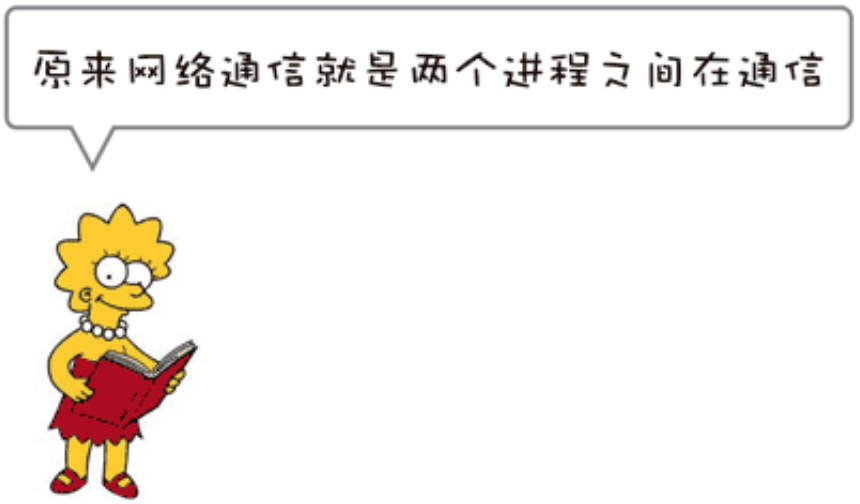
784次阅读

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。

计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。网络编程就是如何在程序中实现两台计算机的通信。

举个例子，当你使用浏览器访问新浪网时，你的计算机就和新浪的某台服务器通过互联网连接起来了，然后，新浪的服务器把网页内容作为数据通过互联网传输到你的电脑上。

由于你的电脑上可能不止浏览器，还有QQ、Skype、Dropbox、邮件客户端等，不同的程序连接的别的计算机也会不同，所以，更确切地说，网络通信是两台计算机上的两个进程之间的通信。比如，浏览器进程和新浪服务器上的某个Web服务进程在通信，而QQ进程是和腾讯的某个服务器上的某个进程在通信。



网络编程对所有开发语言都是一样的，Python也不例外。用Python进行网络编程，就是在Python程序本身这个进程内，连接别的服务器进程的通信端口进行通信。

本章我们将详细介绍Python网络编程的概念和最主要的两种网络类型的编程。
