13 常用内建模块

Python之所以自称“batteries included”，就是因为内置了许多非常有用的模块，无需额外安装和配置，即可直接使用。

本章将介绍一些常用的内建模块。

datetime

datetime是Python处理日期和时间的标准库。

获取当前日期和时间

我们先看如何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`。

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
```

datetime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。

timestamp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
```

注意到timestamp是一个浮点数，它没有时区的概念，而datetime是有时区的。上述转换是在timestamp和本地时间做转换。

本地时间是指当前操作系统设定的时区。例如北京时区是东8区，则本地时间：

```
2015-04-19 12:20:00
```

实际上就是UTC+8:00时区的时间：

```
2015-04-19 12:20:00 UTC+8:00
```

而此刻的格林威治标准时间与北京时间差了8小时，也就是UTC+0:00时区的时间应该是：

```
2015-04-19 04:20:00 UTC+0:00
```

timestamp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
```

str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
```

字符串`'%Y-%m-%d %H:%M:%S'`规定了日期和时间部分的格式。详细的说明请参考Python文档。

注意转换后的datetime是没有时区信息的。

datetime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
```

datetime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)
```

可见，使用`timedelta`你可以很容易地算出前几天和后几天的时刻。

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)))
```

如果系统时区恰好是UTC+8:00，那么上述代码就是正确的，否则，不能强制设置为UTC+8:00时区。

时区转换

我们可以先通过`utcnow()`拿到当前的UTC时间，再转换为任意时区的时间：

```
# 拿到UTC时间，并强制设置时区为UTC+0:00:
>>> utc_dt =
datetime.utcnow().replace(tzinfo=timezone.utc)
>>> print(utc_dt)
2015-05-18 09:05:12.377316+00:00
# astimezone()将转换时区为北京时间:
>>> bj_dt =
utc_dt.astimezone(timezone(timedelta(hours=8)))
>>> print(bj_dt)
2015-05-18 17:05:12.377316+08:00
```

```
# astimezone()将转换时区为东京时间:
>>> tokyo_dt =
utc_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt)
2015-05-18 18:05:12.377316+09:00
# astimezone()将bj_dt转换时区为东京时间:
>>> tokyo_dt2 =
bj_dt.astimezone(timezone(timedelta(hours=9)))
>>> print(tokyo_dt2)
2015-05-18 18:05:12.377316+09:00
```

时区转换的关键在于，拿到一个`datetime`时，要获知其正确的时区，然后强制设置时区，作为基准时间。

利用带时区的`datetime`，通过`astimezone()`方法，可以转换到任意时区。

注：不是必须从UTC+0:00时区转换到其他时区，任何带时区的`datetime`都可以正确转换，例如上述`bj_dt`到`tokyo_dt`的转换。

小结

- `datetime`表示的时间需要时区信息才能确定一个特定的时间，否则只能视为本地时间。
- 如果要存储`datetime`，最佳方法是将其转换为timestamp再存储，因为timestamp的值与时区完全无关。

练习

假设你获取了用户输入的日期和时间如`2015-1-21 9:01:30`，以及一个时区信息如`UTC+5:00`，均是`str`，请编写一个函数将其转换为timestamp：

```
# -*- coding:utf-8 -*-

import re
from datetime import datetime, timezone, timedelta
def to_timestamp(dt_str, tz_str):
    pass
```

```
# 测试:
t1 = to_timestamp('2015-6-1 08:10:30', 'UTC+7:00')
assert t1 == 1433121030.0, t1

t2 = to_timestamp('2015-5-31 16:10:30', 'UTC-09:00')
assert t2 == 1433121030.0, t2

print('ok')
```

collections

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
>>> 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)
```

ChainMap

`ChainMap`可以把一组`dict`串起来并组成一个逻辑上的`dict`。`ChainMap`本身也是一个dict，但是查找的时候，会按照顺序在内部的dict依次查找。

什么时候使用`ChainMap`最合适？举个例子：应用程序往往都需要传入参数，参数可以通过命令行传入，可以通过环境变量传入，还可以有默认参数。我们可以用`ChainMap`实现参数的优先级查找，即先查命令行参数，如果没有传入，再查环境变量，如果没有，就使用默认参数。

下面的代码演示了如何查找`user`和`color`这两个参数：

```
from collections import ChainMap
import os, argparse

# 构造缺省参数:
defaults = {
    'color': 'red',
    'user': 'guest'
}
```

```
# 构造命令行参数:
parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = { k: v for k, v in
vars(namespace).items() if v }

# 组合成ChainMap:
combined = ChainMap(command_line_args, os.environ,
defaults)

# 打印参数:
print('color=%s' % combined['color'])
print('user=%s' % combined['user'])
```

没有任何参数时，打印出默认参数：

```
$ python3 use_chainmap.py
color=red
user=guest
```

当传入命令行参数时，优先使用命令行参数：

```
$ python3 use_chainmap.py -u bob
color=red
user=bob
```

同时传入命令行参数和环境变量，命令行参数的优先级较高：

```
$ user=admin color=green python3 use_chainmap.py -u bob
color=green
user=bob
```

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})
>>> c.update('hello') # 也可以一次性update
>>> c
Counter({'r': 2, 'o': 2, 'g': 2, 'm': 2, 'l': 2, 'p': 1,
'a': 1, 'i': 1, 'n': 1, 'h': 1, 'e': 1})
```

`Counter`实际上也是`dict`的一个子类，上面的结果可以看出每个字符出现的次数。

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`

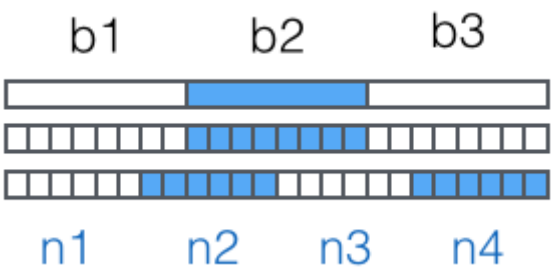
Base64是一种用64个字符来表示任意二进制数据的方法。

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=24`bit，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(b'binary\x00string')
b'YmluYXJ5AHN0cmluZw=='
>>> base64.b64decode(b'YmluYXJ5AHN0cmluZw==')
b'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(b'i\xb7\x1d\xfb\xef\xff')
b'abcd++//'
>>> base64.urlsafe_b64encode(b'i\xb7\x1d\xfb\xef\xff')
b'abcd--__'
>>> base64.urlsafe_b64decode('abcd--__')
b'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。

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。

练习

请写一个能处理去掉=的base64解码函数：

```
# -*- coding: utf-8 -*-
import base64
def safe_base64_decode(s):
    pass
```

```
# 测试:
assert b'abcd' == safe_base64_decode(b'YWJjZA=='),
safe_base64_decode('YWJjZA==')
assert b'abcd' == safe_base64_decode(b'YWJjZA'),
safe_base64_decode('YWJjZA')
print('ok')
```

struct

准确地讲，Python没有专门处理字节的数据类型。但由于b'str'可以表示字节，所以，字节数组＝二进制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bytes，你得配合位运算符这么写：

```
>>> n = 10240099
>>> b1 = (n & 0xff000000) >> 24
>>> b2 = (n & 0xff0000) >> 16
>>> b3 = (n & 0xff00) >> 8
>>> b4 = n & 0xff
>>> bs = bytes([b1, b2, b3, b4])
>>> bs
b'\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。

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/3/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 =
b'\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\x
18\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'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)
```

结果显示，`b'B'`、`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。

```
# -*- coding: utf-8 -*-
import base64, struct
bmp_data =
base64.b64decode('Qk1oAgAAAAAAADYAAAAoAAAAHAAAAAoAAAABABAA
AAAAADICAAASCwAAEgsAAAAAAAAAAAAA/3//f/9//3//f/9//3//f/9//3
//f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9/AHwA
fAB8AHwAfAB8AHwAfP9//3//fwB8AHwAfAB8/3//f/9/AHwAfAB8AHz/f/
9//3//f/9//38AfAB8AHwAfAB8AHwAfAB8AHz/f/9//38AfAB8/3//f/9/
/3//fwB8AHz/f/9//3//f/9//3//f/9/AHwAfP9//3//f/9/AHwAfP9//3
//fwB8AHz/f/9//3//f/9/AHwAfP9//3//f/9//38AfAB8AHwA
fAB8AHwAfP9//3//f/9/AHwAfP9//3//f/9//38AfAB8/3//f/9//3//f/
9//3//fwB8AHwAfAB8AHwAfAB8/3//f/9//38AfAB8/3//f/9//3//fwB8
AHz/f/9//3//f/9//3//f/9/AHwAfP9//3//f/9/AHwAfP9//3//fwB8AH
z/f/9/AHz/f/9/AHwAfP9//38AfP9//3//f/9/AHwAfAB8AHwAfAB8AHwA
fAB8/3//f/9/AHwAfP9//38AfAB8AHwAfAB8AHwAfAB8/3//f/9//38AfA
B8AHwAfAB8AHwAfAB8/3//f/9/AHwAfAB8AHz/fwB8AHwAfAB8AHwAfAB8
AHz/f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9//3//f/9//3
//f/9//3//f/9//38AAA==')
def bmp_info(data):
    return {
        'width': 200,
        'height': 100,
        'color': 24
    }
```

```
# 测试
bi = bmp_info(bmp_data)
assert bi['width'] == 28
assert bi['height'] == 10
assert bi['color'] == 16
print('ok')
```

hashlib

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?'.encode('utf-8'))
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：

```
import hashlib

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

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法不仅越慢，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。

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：

`# -*- coding: utf-8 -*- db = {   'michael': 'e10adc3949ba59abbe56e057f20f883e',   'bob': '878ef96e86145580c38c87f0410ad153',   'alice': '99b1c2188db85afee403b1536010c2c9' } ``# 测试: assert login('michael', '123456') assert login('bob', 'abc999') assert login('alice', 'alice2008') assert not login('michael', '1234567') assert not login('bob', '123456') assert not login('alice', 'Alice2008') print('ok')` Run

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：

```
# -*- coding: utf-8 -*-
import hashlib, random

def get_md5(s):
```

```
    return hashlib.md5(s.encode('utf-8')).hexdigest()

class User(object):
    def __init__(self, username, password):
        self.username = username
        self.salt = ''.join([chr(random.randint(48, 122))
for i in range(20)])
        self.password = get_md5(password + self.salt)
db = {
    'michael': User('michael', '123456'),
    'bob': User('bob', 'abc999'),
    'alice': User('alice', 'alice2008')
}
def login(username, password):
    user = db[username]
    return user.password == get_md5(password)
```

```
# 测试:
assert login('michael', '123456')
assert login('bob', 'abc999')
assert login('alice', 'alice2008')
assert not login('michael', '1234567')
assert not login('bob', '123456')
assert not login('alice', 'Alice2008')
print('ok')
```

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。

hmac

通过哈希算法，我们可以验证一段数据是否有效，方法就是对比该数据的哈希值，例如，判断用户口令是否正确，我们用保存在数据库中的`password_md5`对比计算`md5(password)`的结果，如果一致，用户输入的口令就是正确的。

为了防止黑客通过彩虹表根据哈希值反推原始口令，在计算哈希的时候，不能仅针对原始输入计算，需要增加一个salt来使得相同的输入也能得到不同的哈希，这样，大大增加了黑客破解的难度。

如果salt是我们自己随机生成的，通常我们计算MD5时采用`md5(message + salt)`。但实际上，把salt看做一个“口令”，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。

这实际上就是Hmac算法：Keyed-Hashing for Message Authentication。它通过一个标准算法，在计算哈希的过程中，把key混入计算过程中。

和我们自定义的加salt算法不同，Hmac算法针对所有哈希算法都通用，无论是MD5还是SHA-1。采用Hmac替代我们自己的salt算法，可以使程序算法更标准化，也更安全。

Python自带的hmac模块实现了标准的Hmac算法。我们来看看如何使用hmac实现带key的哈希。

我们首先需要准备待计算的原始消息message，随机key，哈希算法，这里采用MD5，使用hmac的代码如下：

```
>>> 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`。

练习

将上一节的salt改为标准的hmac算法，验证用户口令：

```
# -*- coding: utf-8 -*-
import hmac, random

def hmac_md5(key, s):
    return hmac.new(key.encode('utf-8'), s.encode('utf-
8'), 'MD5').hexdigest()

class User(object):
    def __init__(self, username, password):
        self.username = username
        self.key = ''.join([chr(random.randint(48, 122))
for i in range(20)])
        self.password = hmac_md5(self.key, password)

db = {
    'michael': User('michael', '123456'),
    'bob': User('bob', 'abc999'),
    'alice': User('alice', 'alice2008')
}
def login(username, password):
    user = db[username]
    return user.password == hmac_md5(user.key, password)
```

```
# 测试:
assert login('michael', '123456')
assert login('bob', 'abc999')
assert login('alice', 'alice2008')
assert not login('michael', '1234567')
assert not login('bob', '123456')
assert not login('alice', 'Alice2008')
print('ok')
```

小结

- Python内置的hmac模块实现了标准的Hmac算法，它利用一个key对message计算“杂凑”后的hash，使用hmac算法比标准hash算法更安全，因为针对相同的message，不同的key会产生不同的hash。

itertools

Python的内建模块`itertools`提供了非常有用的用于操作迭代对象的函数。

首先，我们看看`itertools`提供的几个“无限”迭代器：

```
>>> import itertools
>>> natuals = itertools.count(1)
>>> for n in natuals:
...     print(n)
...
1
2
3
...
```

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。

`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'
...
```

同样停不下来。

`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]
```

`itertools`提供的几个迭代器操作函数更加有用：

chain()

`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']
```

练习

计算圆周率可以根据公式：

利用Python提供的itertools模块，我们来计算这个序列的前N项和：

```
# -*- coding: utf-8 -*-
import itertools
def pi(N):
    ' 计算pi的值 '
    # step 1: 创建一个奇数序列: 1, 3, 5, 7, 9, ...

    # step 2: 取该序列的前N项: 1, 3, 5, 7, 9, ..., 2*N-1.

    # step 3: 添加正负符号并用4除: 4/1, -4/3, 4/5, -4/7, 4/9,
...

    # step 4: 求和:
    return 3.14
```

```
# 测试:
print(pi(10))
print(pi(100))
print(pi(1000))
print(pi(10000))
assert 3.04 < pi(10) < 3.05
assert 3.13 < pi(100) < 3.14
assert 3.140 < pi(1000) < 3.141
assert 3.1414 < pi(10000) < 3.1415
print('ok')
```

小结

- `itertools`模块提供的全部是处理迭代功能的函数，它们的返回值不是list，而是`Iterator`，只有用`for`循环迭代的时候才真正计算。

contextlib

在Python中，读写文件这样的资源要特别注意，必须在使用完毕后正确关闭它们。正确关闭文件资源的一个方法是使用`try...finally`：

```
try:
    f = open('/path/to/file', 'r')
    f.read()
finally:
    if f:
        f.close()
```

写`try...finally`非常繁琐。Python的`with`语句允许我们非常方便地使用资源，而不必担心资源没有关闭，所以上面的代码可以简化为：

```
with open('/path/to/file', 'r') as f:
    f.read()
```

并不是只有`open()`函数返回的fp对象才能使用`with`语句。实际上，任何对象，只要正确实现了上下文管理，就可以用于`with`语句。

实现上下文管理是通过`__enter__`和`__exit__`这两个方法实现的。例如，下面的class实现了这两个方法：

```
class Query(object):

    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print('Begin')
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type:
            print('Error')
        else:
            print('End')

    def query(self):
        print('Query info about %s...' % self.name)
```

这样我们就可以把自己写的资源对象用于`with`语句：

```
with Query('Bob') as q:
    q.query()
```

@contextmanager

编写`__enter__`和`__exit__`仍然很繁琐，因此Python的标准库`contextlib`提供了更简单的写法，上面的代码可以改写如下：

```
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``；
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``。

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。

`@contextlib`还有一些其他decorator，便于我们编写更简洁的代码。

urllib

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URL`https://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; charset=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},"subtitl
e":"","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}}
```

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
```

小结

urllib提供的功能就是利用程序去执行各种HTTP请求。如果要模拟浏览器完成特定功能，需要把请求伪装成浏览器。伪装的方法是先监控浏览器发出的请求，再根据浏览器的请求头来伪装，`User-Agent`头就是用来标识浏览器的。

练习

利用urllib读取JSON，然后将JSON解析为Python对象：

```
# -*- coding: utf-8 -*-
from urllib import request
def fetch_data(url):
    return ''
```

```
# 测试
URL = 'https://query.yahooapis.com/v1/public/yql?
q=select%20*%20from%20weather.forecast%20where%20woeid%20%
3D%202151330&format=json'
data = fetch_data(URL)
print(data)
assert data['query']['results']['channel']['location']
['city'] == 'Beijing'
print('ok')
```

XML

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.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)
```

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

请利用SAX编写程序解析Yahoo的XML格式的天气预报，获取天气预报：

`https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20weather.forecast%20where%20woeid%20%3D%2021513330&format=xml`

参数`woeid`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。

```
# -*- coding:utf-8 -*-

from xml.parsers.expat import ParserCreate
from urllib import request
def parseXml(xml_str):
    print(xml_str)
    return {
        'city': '?',
        'forecast': [
            {
                'date': '2017-11-17',
                'high': 43,
                'low' : 26
            },
            {
                'date': '2017-11-18',
```

```
                'high': 41,
                'low' : 20
            },
            {
                'date': '2017-11-19',
                'high': 43,
                'low' : 19
            }
        ]
    }
```

```
# 测试:
URL = 'https://query.yahooapis.com/v1/public/yql?
q=select%20*%20from%20weather.forecast%20where%20woeid%20%
3D%202151330&format=xml'

with request.urlopen(URL, timeout=4) as f:
    data = f.read()

result = parseXml(data.decode('utf-8'))
assert result['city'] == 'Beijing'
```

HTMLParser

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 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 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`A`，这两种字符都可以通过Parser解析出来。

小结

- 利用HTMLParser，可以把网页中的文本、图像等解析出来。

练习

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。