第 3 章

时间和财务计算

引言

感谢：Gustavo Niemeyer、Facundo Batista

今天、上周、明年。这些词听起来是如此的普通。你可能会想知道，我们的生活与时间观念的关联究竟有多深。关于时间的概念无处不在，当然，它们也在大多数的软件中有所体现。即使是一些非常简单的程序都可能与时间有关，比如时间戳、延迟、超时、速度测量、日历等。为了满足通用程序的这类需求，Python 标准库提供了坚实的基础支持，而更多的其他支持则来自于第三方模块和包。

涉及到财务的计算则是另一个引人注意的有趣问题，因为它与我们的日常生活联系非常紧密。Python 2.4 引入了对十进制数字的支持（当然也可以在 Python 2.3 中引入，参看 http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html），这使得用户可以避免使用二进制浮点数，也使得 Python 成为了执行这类计算的一个很好的选择。

本章将覆盖着两个主题，财务和时间。我们也许应该说这一章其实只有一个主题，毕竟大家都知道一句老话，时间就是金钱。

时间模块

Python 标准库的时间模块使得 Python 应用程序可以利用和借助其运行平台提供的各种与时间相关的功能。因此，在你的平台中，提供等价功能的 C 库文档可能会很有用，而且，在一些比较奇特的平台上，Python 也可能会受到平台自身的一些影响。

时间模块中最常用的一个函数就是获取当前时间的函数 time.time。在未初始化的情况下其返回值看起来实在是不够直观：一个浮点数，代表了从某个特定时间点——也被称为纪元（epoch）——开始所经历的秒数，这个时间点根据不同的平台也可能会有些不同，但通常都是 1970 年 1 月 1 日午夜。

要检查平台所使用的纪元，可以在 Python 交互式解释器的提示符下输入下面语句：

```
>>> import time
>>> print time.asctime(time.gmtime(0))
```

注意，我们给 time.gmtime 函数传递了参数 0（表示从纪元之后 0 秒开始）。time.gmtime 将任何时间戳（从纪元开始所经历的秒数）转化为一个元组，该元组代表了人类容易理解的一种时间格式，而无须进行任何时区转化（GMT 代表了“格林威治标准时间”，也就是大家通常所知的 UTC，“世界标准时间”的另一种说法）。也可以传递一个时间戳（从纪元开始所经历的秒数）给 time.localtime，它会根据当前时区进行时间转化。

理解这个区别很重要，如果获得一个已经根据当地时间进行调整了的时间戳，将其传递给 time.localtime 函数，将不会获得一个预期的结果——除非非常走运，你的当前时区恰好就是 UTC 时区。

这里给出一个从返回的元组中获取当前本地时间的方法：

```
year, month, mday, hour, minute, second, wday, yday = time.localtime( )
```

虽然代码能运行，但是却不是很优雅，最好不要经常这么用。也可以完全避免用这种方式获取时间，因为 time 函数返回的元组提供了有意义的属性名，更加易于使用。比如，获取当前月份，就可以写成简洁而优雅的一行：

```
time.localtime( ).tm_mon
```

注意，我们忽略了传递给 localtime 的参数。当调用 localtime、gmtime 或者 asctime 而不提供参数时，默认使用当前时间。

时间模块中两个非常有用的函数是 strftime——它可以根据返回的时间元组构建一个字符串，以及 strptime——与前者完全相反，它将解析给定的字符串并产生一个时间元组。这两个函数都接受一个格式化字符串，可以指明真正感兴趣的部分（或者你希望从字符串中解析出的部分）。关于传递给这两个函数的格式字符串的更多信息和细节请参看 http://docs.python.org/lib/module-time.html。

时间模块中最后一个重要的函数是 time.sleep，它使得可以在 Python 程序中实现延时。虽然这个函数的 POSIX 对应版本只接受一个整数参数，Python 的版本则支持一个浮点数，并允许非整数秒的延时。比如：

```
for i in range(10):
     time.sleep(0.5)
     print "Tick!"
```

这段代码大概耗时 5s，并不断打印出“Tick!”，大约每秒两次。

时间和日期对象

除了非常有用的时间模块，Python 标准库也同时引入了 datetime 模块，该模块提供了

能更好地对应于抽象的日期和时间的各种类型，比如 time、date 和 datetime 类型。构造这些类型实例的代码也非常优雅和简单：

```
today = datetime.date.today( )
birthday = datetime.date(1977, 5, 4)      #5 月 4 日
currenttime = datetime.datetime.now( ).time( )
lunchtime = datetime.time(12, 00)
now = datetime.datetime.now( )
epoch = datetime.datetime(1970, 1, 1)
meeting = datetime.datetime(2005, 8, 3, 15, 30)
```

更进一步，正如我们所料，通过属性和方法，这些类型提供了很方便的获取和操作信息的方法。下面的代码创建了一个 date 类型，代表当前日期，然后获得下一年的同样的日期，最后将结果打印出来：

```
today = datetime.date.today( )
next_year = today.replace(year=today.year+1).strftime("%Y.%m.%d")
print next_year
```

注意年是怎样递增的，以及 replace 方法的使用。直接给 date 和 time 实例的属性赋值这一想法颇具诱惑力，但实际上这些实例是无法改写的（这其实是好事，意味着我们可以将这些实例用作集合的成员，或者字典的键），所以，新实例只能通过创建获得，而无法通过修改一个旧的实例来达成。

datetime 模块通过 timedelta 类型为时间差（即两个时间实例的差值；可以把它想象为一段时间）提供了一些基本支持。这个类型允许你使用给定的时间片，在一个指定的日期上增减时间，同时这个类型也可以作为 time 和 date 实例的差值计算结果。

```
>>> import datetime
>>> NewYearsDay = datetime.date(2005, 01, 01)
>>> NewYearsEve = datetime.date(2004, 12, 31)
>>> oneday = NewYearsDay - NewYearsEve
>>> print oneday
1 day, 0:00:00
>>>
```

timedelta 实例在内部被表示为天数，秒数和微秒数，但也可以通过直接提供这些参数或者其他参数，比如分钟数、小时数和周数来构建一个 timedelta 实例。其他类型的差值，比如月的，年的，则故意没有提供，因为它们的含义和操作结果等，还存在一些争议。（但第三方 dateutil 包则提供了这些特性，见 https://moin.conectiva.com.br/DateUtil）。

datetime 的设计很有远见，同时也很谨慎。其策略是，不实现难以预测的任务以及那些在不同的系统中需要用到不同实现的任务，不强求做到无所不能。当前的实现，已经提供了良好的接口，并适用于大多数情况，更重要的是，还提供了一个坚实的基础以便第三方模块可以在其基础上继续发展。

另一个反映 datetime 设计谨慎的地方是该模块对时区的支持。虽然 datetime 提供了很好

的查询和设置时区信息的方法，但如果没有一个外部的来源来提供 tzinfo 类型的非抽象化的子类，那些方法不会有什么用。至少有两个第三方包为 datetime 提供了时区支持：前面已经提到过的 dateutil 以及 pyTZ，见 http://sourceforge.net/projects/pytz/。

十进制

decimal 是 Python 标准库提供的模块，也是 Python 2.4 新引入的内容，并最终给 Python 带来了十进制数学计算。感谢 decimal，我们现在终于拥有了十进制数数据类型，具有有界精度和浮点。让我们来仔细看看这三个方面：

十进制数字数据类型

> 数不是被储存为二进制，而是存为一个十进制数字的序列。

有界精度

> 用于存储数的数字的数目是固定的。（对于每个十进制数对象而言，那是一个固定的参数，但是不同的十进制数对象也可以被设置为使用不同数目的数字。）

浮点

> 十进制小数点并没有一个固定的位置。（或者这样讲：所有数字的总数是固定的，小数点后面的数字并没有一个固定的数目。如果数目是固定的，那应该叫做固点数——而非浮点数——数据类型。）

这样的数据类型有很多用途（最大用途就是用于财务计算），特别是，decimal.Decimal 比标准的二进制 float 提供了更多的高级功能。最大的优点是，所有用户输入的数（也就是所有的具有有限位数字的数），都能够被精确地表示出来（作为反例的是，用二进制浮点存储用户输入的数据，常常不能够严格按照原样表示出来）：

```
>>> import decimal
>>> 1.1
1.1000000000000001
>>> 2.3
2.2999999999999998
>>> decimal.Decimal("1.1")
Decimal("1.1")
>>> decimal.Decimal("2.3")
Decimal("2.3")
```

这种完全一致的表示也可用于计算。而关于二进制浮点，举个例子：

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
```

虽然其差异极小，接近于 0，但是这种微小差异却不利于我们进行可靠的相等比较运算；另外，这些微小的差异值还会不断累积。基于这个原因，对于那些涉及账目计算，对相等比较计算要求很严格的程序，decimal 比二进制浮点更加适用：

```
>>> d1 = decimal.Decimal("0.1")
>>> d3 = decimal.Decimal("0.3")
>>> d1 + d1 + d1 - d3
Decimal("0.0")
```

我们可通过整数、字符串或者元组构建 decimal.Decimal。要从一个浮点数创建一个 decimal.Decimal，首先需要将浮点转化为字符串。这是必要的一步，我们在这里可以显式地指明转化的细节，甚至包括了对错误的表示。十进制数字包括了一些特殊的值，比如 NaN（代表了“非数字”）、正负无穷大，还有-0。一旦创建成功，decimal.Decimal 对象也是无法修改的，就像 Python 中的其他数字一样。

decimal 模块本质上只是实现了我们在学校学到的一些数学运算法则。对于指定的精度要求，它总是尽可能给出没有截断的结果：

```
>>> 0.9 / 10
0.08999999999999997
>>> decimal.Decimal("0.9") / decimal.Decimal(10)
Decimal("0.09")
```

当结果中的数字的数目超过了精度要求，则根据当前的舍入方式对结果进行舍入处理。有几种舍入方式；但默认的是 round-half-even 方式（译者注：四舍六入，如果是 5，则舍入到最接近的偶数。举个例子：2.45 只保留一位小数，结果是 2.4）。

decimal 模块引入了有效位的概念，例如 1.30+1.20 结果是 2.50。末尾的 0 是为了指出有效位。对于涉及到财务计算的应用程序，这是经常采用的表示方法。对于乘法，“教科书式的”方法会使用乘数中的所有数字：

```
>>> decimal.Decimal("1.3") * decimal.Decimal("1.2")
Decimal("1.56")
>>> decimal.Decimal("1.30") * decimal.Decimal("1.20")
Decimal("1.5600")
```

和其他内建数字类型一样，如 float 和 int，decimal 对象拥有数的一些标准属性，除此之外它还有一些特殊的方法。具体信息请参看文档中对其方法的介绍以及相关的示例。

decimal 数据类型通常都工作在一个具体的环境中，也就是说一些配置信息已经被预先设定了。每个线程都有它自己的当前环境（拥有独立的上下文环境意味着每个线程都可以做自己的改变和设定而不互相影响）；通过 decimal 模块提供的 getcontext 和 setcontext 函数，我们可以访问和修改当前线程的环境。

不像基于硬件的二进制浮点数，decimal 模块的精度可以由用户设置（默认是 28 位）。对于任意一个给定的问题，它都可以被设置得足够大：

```
>>> decimal.getcontext( ).prec = 6              # 精度被设为 6...
>>> decimal.Decimal(1) / decimal.Decimal(7)
```

```
Decimal("0.142857")
>>> decimal.getcontext( ).prec = 60            # ...被设为 60 个数字
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857142857142857142857142857142857142857142857142857142857")
```

但 decimal 中还有一些不是那么简单和基本的东西。本质上，decimal 实现了标准的数学计算，可以在 http://www2.hursley.ibm.com/decimal/查到更多细节。这也意味着 decimal 支持信号的概念。信号表示在计算中出现的一些不正常的情况（比如，1/0，0/0，无穷大/无穷大）。根据各个应用程序的不同需求，有的信号被忽略掉了，有的被用来提供额外信息，有的则被作为异常。对于每个信号，都有一个标志和一个陷阱控制器。当一个信号发生了，它的标志由 0 开始递增，然后如果陷阱控制器被设置为 1，它会抛出一个异常。这给了程序员极大的权力和自由度来配置 decimal，以满足他们特定的需求。

decimal 的优点如此的多，为什么还有人坚持要用 float？为什么 Python（像很多其他流传广泛的语言一样，但 Cobol 和 Rexx 也是很容易想到的两个例外）会一开始就采用浮点二进制数作为它默认的（也是唯一的）非整数数据类型？当然，理由可以给出一大堆，但最终都归结到一点上，速度！看看下面的实验：

```
$ python -mtimeit -s'from decimal import Decimal as D' 'D("1.2")+D("3.4")'
10000 loops, best of 3: 191 usec per loop
$ python -mtimeit -s'from decimal import Decimal as D' '1.2+3.4'
1000000 loops, best of 3: 0.339 usec per loop
```

对上面的输出做个简单的翻译：在这台机器上（一台老的 1.2 GHz Athlon PC，运行 Linux），Python 每秒可以执行 300 万次 float 加法（使用个人计算机提供的支持数学计算的硬件），但只能执行 5000 次 Decimal 的加法（完全由软件完成）。

基本上，如果需要对非整数做上百万次的加法，应该坚持用 float。在过去，一台普通计算机的运算速度上千倍地落后于现在的计算机的速度（其实那也不是太久之前），因此，当程序运行在廉价的机器上时，即使只是做少量的计算也会有诸多限制。而 Rexx 和 Cobol 诞生于大型机系统，这些系统还没有今天最廉价的计算机运算速度快，价格却是这些计算机的成千上万倍。这类大型机系统的买主也许可以负担得起这种方便而易用的十进制数学计算，但绝大多数其他语言，通常诞生于更加廉价的性能有限的机器之上，却无法负担这种开销。

幸运的是，需要对非整数进行大量数学计算的应用程序的数量相对还比较少，在当今的普通计算机上，适量的十进制数学计算也不会引起一些可见的性能问题。因此，绝大多数应用程序都可以利用 decimal 在各方面的优势，包括那些需要继续运行在 Python 2.3 上的程序。从版本 2.4 之后，decimal 已经成为了 Python 标准库的一部分。要了解更多在 Python 2.3 中集成 decimal 的细节，参看 http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html。

3.1 计算昨天和明天的日期

感谢：Andrea Cavalcanti

任务

你想获得今天的日期，并以此计算昨天和明天的日期。

解决方案

无论何时你遇到有关“时间变化”或者“时间差”的问题，先考虑 timedelta：

```
import datetime
today = datetime.date.today( )
yesterday = today - datetime.timedelta(days=1)
tomorrow = today + datetime.timedelta(days=1)
print yesterday, today, tomorrow
#输出：2004-11-17 2004-11-18 2004-11-19
```

讨论

自从 datetime 模块出现以来，这个问题在 Python 邮件列表中频频露面。当首次碰到这个问题时，人们的第一个想法是写这样的代码：yesterday = today – 1，但其结果是一个 TypeError: unsupported operand type(s) for -: ‘datetime.date’ and ‘int’。

一些人认为这是个 bug，并暗示 Python 应该能够猜测出他们的意图。然而，Python 的一个指导原则是：“在模糊含混面前拒绝猜测”，这也是 Python 简洁和强大的原因。猜测意味着需要用启发的方式将 datetime 割裂开来，需要猜测你想减去的究竟是 1 天还是 1 秒，再或者干脆是 1 年？

Python 如同它一贯的方式，并不尝试猜测你的意图，而是期待你明确指定你自己的意图。如果想减去长度为 1 天的一个时间差，应当明确地编写相关代码。再或者，想加上长度为 1 秒的时间差值，可以使用 timedelta 配合 datetime.datetime 对象，这样可以用同样的语法编写相关操作。也许对每个任务都想使用这种方法，因为这种方法给了你足够的自由度，同时还保持着简洁直观。考虑下面的片段：

```
>>> anniversary = today + datetime.timedelta(days=365)# 增加 1 年
>>> print anniversary
2005-11-18
>>> t = datetime.datetime.today( )                    # 获得现在的时间
>>> t
datetime.datetime(2004, 11, 19, 10, 12, 43, 801000)
>>> t2 = t + datetime.timedelta(seconds=1)            # 增加 1 秒
>>> t2
datetime.datetime(2004, 11, 19, 10, 12, 44, 801000)
```

```
>>> t3 = t + datetime.timedelta(seconds=3600)          # 增加 1 小时
>>> t3
datetime.datetime(2004, 11, 19, 11, 12, 43, 801000)
```

如果你想在日期和时间的计算上有点新花样，可以使用第三方包，如 dateutil（可以和内建的 datetime 协同工作）和经典的 mx.DateTime。举个例子：

```
from dateutil import relativedelta
nextweek = today + relativedelta.relativedelta(weeks=1)
print nextweek
#输出: 2004-11-25
```

然而，“总是应该让它尽可能简单地工作”。为了保持简单，本节解决方案中使用了 datetime.timedelta。

更多资料

见 https://moin.conectiva.com.br/DateUtil?action=highlight&value= DateUtil 的 dateutil 文档，*Library Reference* 中关于 datetime 的文档。mx.DateTime 的资料可在 http://www.egenix.com/files/python/mxDateTime.html 找到。

3.2 寻找上一个星期五

感谢：Kent Johnson、Danny Yoo、Jonathan Gennick、Michael Wener

任务

你想知道上一个星期五的日期（包括今天，如果今天是星期五）并以特定格式将其打印出来。

解决方案

通过 Python 标准库的 datetime 模块，此任务可轻松完成：

```
import datetime, calendar
lastFriday = datetime.date.today( )
oneday = datetime.timedelta(days=1)
while lastFriday.weekday( ) != calendar.FRIDAY:
     lastFriday -= oneday
print lastFriday.strftime('%A, %d-%b-%Y')
# 输出: Friday, 10-Dec-2004
```

讨论

上面的代码片段可帮助我们找到上一个星期五的日期，并以正确的格式打印，无论上个星期五是否在同一个月，甚至不在同一年也没有关系。在这个例子中，我们试图寻

找星期五（包括今天，如果今天是星期五）。星期五的整数表示是 4，但我们应该避免直接依赖和使用这个数字，我们导入 Python 标准库的 calendar 模块，并利用它的 calendar.FRIDAY 属性（其实这个属性值就是 4）。首先我们创建一个叫做 lastFriday 的变量，并将它设定为今天的日期，然后不断地向前对比检查，直到找到某个日期，它的 weekday 的值是 4。

一旦找到了需要的日期，使用 datetime.date 类的 strftime 方法，可以很容易地将其转化为我们需要的格式。

还有另一个方法，看上去也更简洁一点，即利用内建的 datetime.date.resolution，而不是显式地创建一个 datetime.timedelta 实例来表示一天的时间长度：

```
import datetime, calendar
lastFriday = datetime.date.today( )
while lastFriday.weekday( ) != calendar.FRIDAY:
     lastFriday -= datetime.date.resolution
print lastFriday.strftime('%d-%b-%Y')
```

datetime.date.resolution 这个类属性的值和第一段代码中的 oneday 变量的值完全一样。不过，resolution 可能会让你犯错。datetime 模块的不同类的 resolution 属性的值也不同——对于 date 类，这个值是 timedelta(days=1)，但对于 time 和 datetime 类，其值为 timedelta(microseconds=1)。可以将它们混合搭配（比如，给 datetime.datetime 实例加上一个 datetime.date.resolution），但有时却很容易混淆并误用。本节解决方案中用了一个命名的 onday 变量，更加通用也更加明确，没有任何可能引起误解的地方。因此，这种方式也更加具有 Python 风格（这也是为什么它被当做“正式”的解决方案的原因）。

还有一点可以改进，连循环都可以省略，所以也不用在循环中每次减去一天并做比较：借助模运算，可以一次计算出需要减去的天数：

```
import datetime, calendar
today = datetime.date.today( )
targetDay = calendar.FRIDAY
thisDay = today.weekday( )
deltaToTarget = (thisDay - targetDay) % 7
lastFriday = today - datetime.timedelta(days=deltaToTarget)
print lastFriday.strftime('%d-%b-%Y')
```

如果对这段代码如何工作还有疑问，也许需要复习一下模运算的知识，参看：http://www.cut-the-knot.org/blue/Modulo.shtml。

但应该使用你认为最清楚明白的方法，而不用担心性能问题。还记得 Hoare 的名言吗（总是被错误的归为 Knuth 的言论，但实际上他只是引用 Hoare）：“过早优化是万恶之源。”让我们看看为什么在这里的优化还太早。

减去公共的计算部分（计算当前日期，格式化并打印），在一台 4 岁高龄的运行着 Linux 和 Python 2.4 的老个人计算机上，最慢的方法（也就是被用作“解决方案”的方法，因

为它足够清晰直观）耗时 18.4μs；最快的方法（避免循环，并使用各种技巧使之提速）耗时 10.1μs。

真的需要很频繁地运行这段代码吗？以至于那 8μs 的差异都变得很重要（如果用当前较新的硬件平台，这个差异值还会变得更小）？如果要考虑计算今天的日期和格式化结果的开销，还需要再加上 37μs，包括了 print 语句的 I/O 开销；这样，最慢的也是最清晰的方法将耗时 55μs，而最快的，也是最精炼的方式，耗时 47μs，这点差异实在是不值得担心。

更多资料

Library Reference 中 datetime 模块和 strftime 的文档（见 http://www.python.org/doc/lib/module-datetime.html 和 http://www.python.org/doc/current/lib/node208.html）。

3.3 计算日期之间的时段

感谢：Andrea Cavalcanti

任务

给定两个日期，需要计算这两个日期之间隔了几周。

解决方案

标准的 datetime 和第三方的 dateutil 模块（准确地说是 dateutil 的 rrule.count 方法）很易于使用。在导入了正确的模块之后，任务变得非常简单：

```
from dateutil import rrule
import datetime
def weeks_between(start_date, end_date):
     weeks = rrule.rrule(rrule.WEEKLY, dtstart=start_date, until=end_date)
     return weeks.count( )
```

讨论

函数 weeks_between 用一个开始日期和一个结束日期作为参数，并实例化一个规则用于计算两者之间的周数，最后返回此规则的 count 方法的结果——写代码比描述这个过程简单多了。这个方法只返回一个整数（它不可能是所谓的“半”周）。举个例子，8 天被认为是 2 周。下面做个测试：

```
if __name__=='__main__':
     starts = [datetime.date(2005, 01, 04), datetime.date(2005, 01, 03)]
     end = datetime.date(2005, 01, 10)
     for s in starts:
          days = rrule.rrule(rrule.DAILY, dtstart=s, until=end).count( )
          print "%d days shows as %d weeks "% (days, weeks_between(s, end))
```

测试结果是：

```
7 days shows as 1 weeks
8 days shows as 2 weeks
```

如果不喜欢，就没有必要给递归规则指定名字，而是修改函数体，比如下面这样，用一条语句完成所有操作：

```
return rrule.rrule(rrule.WEEKLY,dtstart=start_date,until=end_date).count()
```

这样也工作得很好。但坦率地说，我还是倾向于给递归规则指定一个名字，不然我会觉得不习惯。当然代码已经很好用了，这也许只是我的个人偏好。

更多资料

参考 dateutil 模块的文档，见 https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil，以及 *Library Reference* 中 datetime 的文档。

3.4 计算歌曲的总播放时间

感谢：Anna Martelli Ravenscroft

任务

你想获取一个列表中的所有歌曲的播放时间之和。

解决方案

我们使用 datetime 标准模块和内建的 sum 函数来完成这个任务：

```
import datetime
def totaltimer(times):
     td = datetime.timedelta(0)       # 将总和初始化（必须是 timedelta）
     duration = sum([
          datetime.timedelta(minutes=m, seconds=s) for m, s in times],
          td)
     return duration
if __name__== '__main__':             # 测试模块是否作为主脚本运行
     times1 = [(2, 36),               # 列出包含的元组（分，秒）
               (3, 35),
               (3, 45),]
     times2 = [(3, 0),
               (5, 13),
               (4, 12),
               (1, 10),]
     assert totaltimer(times1) == datetime.timedelta(0, 596)
     assert totaltimer(times2) == datetime.timedelta(0, 815)
```

```
print ("Tests passed.\n"
         "First test total: %s\n"
         "Second test total: %s" % (
         totaltimer(times1), totaltimer(times2)))
```

讨论

在工作之余，我喜欢听歌，我有很长的一个歌曲列表。我希望能够从中挑选一些歌曲并获得它们的总播放时间，而无须事先建立一个新的播放列表。我写了一个脚本来完成这个任务。

当计算两个 datetime 对象之间的差异时，一般返回一个 datetime.timedelta 对象，可以创建你自己的 timedelta 实例来代表任何给定的时长（datetime 模块中其他的类，比如 datetime 类，只是代表一个时间点）。这里，我们需要计算的是总时长，所以很明显，我们需要使用 timedelta。

datetime.timedelta 可以有很多不同的可选参数：天数、秒数、微秒数、毫秒数、分钟数、小时数和周数。因此，要创建一个实例，需要明确地传递一个参数以避免混淆。如果简单地调用 datetime.timedelta(m, n)，而不指明参数，这个类会通过位置来确定参数，把 m 和 n 当做天数和秒数，从而产生奇怪的结果。（我为这个错误曾经郁闷过一段时间，所以一定要做好测试工作。）

要对一个对象列表，比如 timedelta 列表，使用内建的 sum 函数，必须给 sum 传入第二个参数作为初始化的值——否则，默认的初始值是 0，整数 0。当你试图给它加上一个 timedelta 对象时，你会得到一个错误。另外，传入的第一个参数是一个可迭代体，其中的所有对象都应该支持数字加法。（特别指出，字符串不支持数字加法，我在这里强烈建议：不要使用 sum 把很多的列表连接起来）在 Python 2.4 中，我们可以将方括号[]替换成圆括号()，从而使用生成器表达式而不是列表推导来作为 sum 的第一个参数，当需要处理几千首歌的时候，这样做会更方便。

至于测试用例，我手工创建了一个元组的列表，每个元素记录了歌曲的时长，具体的数据是分钟数和秒数。脚本还可以再进一步改进加强，比如增强对不同格式的辨识能力（如类似 mm:ss 的格式），甚至增加直接从文件中读取信息或者直接访问音乐库的能力。

更多资料

Library Reference 中 sum 和 datetime 部分。

3.5 计算日期之间的工作日

感谢：Anna Martelli Ravenscroft

任务

你想计算两个日期之间的工作日，而非天数。

解决方案

由于工作日和节日在不同的国家，不同地区，甚至在同一家公司不同部门之间，都会有所不同，所以并不存在一个内建的简单办法来解决这个问题。不过，利用 dateutil，配合 datetime 对象，完成这个任务也不是很难：

```
from dateutil import rrule
import datetime
def workdays(start, end, holidays=0, days_off=None):
     if days_off is None:
          days_off = 5, 6           # 默认：周六和周日
     workdays = [x for x in range(7) if x not in days_off]
     days = rrule.rrule(rrule.DAILY, dtstart=start, until=end,
                        byweekday=workdays)
     return days.count( ) - holidays
if __name__ == '__main__':
# 主脚本运行时的测试代码
     testdates=[(datetime.date(2004,9,1),datetime.date(2004,11,14),2),
                (datetime.date(2003,2,28),datetime.date(2003,3,3),1),]
     def test(testdates, days_off=None):
          for s, e, h in testdates:
               print'total workdays from %s to %s is %s with %s holidays'%(
                        s, e, workdays(s, e, h, days_off), h)
     test(testdates)
     test(testdates, days_off=[6])
```

讨论

这也是我的第一个 Python 项目：给定一个开始日期和一个结束日期（含），我需要为培训者们确切地计算出培训天数。在 Python 2.2 中这个问题可能会有点麻烦；而现在，在 datetime 模块和第三方包的 dateutil 的帮助下，这个问题简单多了。

函数 workdays 给变量 day_off 赋了一个合适的默认值（除非明确地传入了一个参数作为 dayoff 的值），这个值其实是一个非工作日的序列。在我的公司里，不同的人有不同的非工作日，但非工作日数量上通常都是少于工作日的，所以记录和修改非工作日总的来说要更容易一些。我把非工作日当做一个参数，是为了在遇到不同需求时可以给 days_off 传递不同的值。接着，这个函数使用列表推导来创建一个实际的工作日列表，也就是那些不在非工作日列表 days_off 中的日子。之后，函数就可以进行实际计算了。

本节例子中的主要变量叫做 days，它是 dateutil 的 rrule（递归规则）类的一个实例。我们可以给 rrule 类传入不同的参数以创建不同的规则对象。在本例中，我传递了一个常

用的(rrule.DAILY)的规则，指定了起始日期和结束日期——它们必须都是 datetime.date 对象，同时还指定了需要统计在内的工作日（weekdays）。最后，根据这个规则，我只需要调用 days.count 方法来计算符合条件的情况发生了多少次即可。(见第 3.3 节中 rrule 的 count 方法的其他应用。)

可以很容易地定义自己的周末：给 days_off 传递任何需要的值。在本节中，其默认值是标准的美国周末时间，周六和周日。然而，如果你的公司只需要工作 4 天，比如，周二到周五，可以传递参数使得 days_off=(5, 6, 0)。即使如第二个测试所示，容器中实际上只有一天，也要确保传递给 days_off 的值是可迭代对象，如列表或者元组。

还可以做一点简单但有用的加强，比如检查你的开始日期和结束日期是否是周末，并用一个 if/else 语句来处理周末轮班，同时适当地修改 days_off。当然还有更多可以加强的地方，比如加入病休处理，进行自动的节日查询处理，因而无须直接传入节日的数目（本节的做法）。基于这个目的，第 3.6 节实现了一个节日列表。

更多资料

参考 dateutil 文档，见 https://moin.conectiva.com.br/DateUtil?action=highlight&value= DateUtil，*Library Reference* 中的 datetime 部分；第 3.3 节中 rrule.count 的其他用法；第 3.5 节中的自动节日查询。

3.6 自动查询节日

感谢：Anna Martelli Ravenscroft、Alex Martelli

任务

不同的国家，不同的地区，甚至同一个公司不同的工会，节日都可能有所不同。需要找到一个办法，能够在给定起始日期和结束日期的条件下，自动获取总共的节假日天数。

解决方案

给定两个日期，它们之间的节日的日期有可能是不固定的，比如复活节和劳工节（美国）；基于复活节的节日，比如节礼日；以及固定日期的节日，比如圣诞节；或者你们公司的节日（比如 CEO 的生日）。不过你都可以利用 datetime 和第三方模块 dateutil 处理所有这些节日。

一个灵活的结构，应该能够提取出诸多的可能性，将其分别包装成函数，并在适当的时候调用那些函数：

```
import datetime
from dateutil import rrule, easter
try: set
```

```
except NameError: from sets import Set as set
def all_easter(start, end):
     # 返回在开始和结束日期之间的复活节列表
     easters = [easter.easter(y)
                    for y in xrange(start.year, end.year+1)]
     return [d for d in easters if start<=d<=end]
def all_boxing(start, end):
     #返回在开始和结束日期之间的节礼日列表
     one_day = datetime.timedelta(days=1)
     boxings = [easter.easter(y)+one_day
                  for y in xrange(start.year, end.year+1)]
     return [d for d in boxings if start<=d<=end]
def all_christmas(start, end):
     #返回在开始和结束日期之间的圣诞节列表
     christmases = [datetime.date(y, 12, 25)
                     for y in xrange(start.year, end.year+1)]
     return [d for d in christmases if start<=d<=end]
def all_labor(start, end):
     #返回在开始和结束日期之间的劳动节列表
     labors = rrule.rrule(rrule.YEARLY, bymonth=9, byweekday=rrule.MO(1),
                                dtstart=start, until=end)
     return [d.date( ) for d in labors]   # 无须测试是否在两个日期之间
def read_holidays(start, end, holidays_file='holidays.txt'):
     # 返回在开始和结束日期之间的假期列表
     try:
          holidays_file = open(holidays_file)
     except IOError, err:
          print 'cannot read holidays (%r):' % (holidays_file,), err
          return [  ]
     holidays = [  ]
     for line in holidays_file:
          # 跳过空行和注释
          if line.isspace( ) or line.startswith('#'):
               continue
          # 试图解析格式: YYYY, M, D
          try:
               y, m, d = [int(x.strip( )) for x in line.split(',')]
               date = datetime.date(y, m, d)
          except ValueError:
               # 检测无效行并继续
               print "Invalid line %r in holidays file %r" % (
                      line, holidays_file)
               continue
          if start<=date<=end:
               holidays.append(date)
     holidays_file.close( )
     return holidays
holidays_by_country = {
     # 将各个国家代码映射到一系列函数
```

```
        'US': (all_easter, all_christmas, all_labor),
        'IT': (all_easter, all_boxing, all_christmas),
    }
    def holidays(cc, start, end, holidays_file='holidays.txt'):
        # 从文件中读取可用的假期
        all_holidays = read_holidays(start, end, holidays_file)
        # 加入用函数计算出的假期
        functions = holidays_by_country.get(cc, ( ))
        for function in functions:
            all_holidays += function(start, end)
        # 消除重复
        all_holidays = list(set(all_holidays))
        # 使用下面两行可以返回一个排序后的列表
        # all_holidays.sort( )
        # return all_holidays
        return len(all_holidays)    # 如果想返回列表注释此行
    if __name__ == '__main__':
        test_file = open('test_holidays.txt', 'w')
        test_file.write('2004, 9, 6\n')
        test_file.close( )
        testdates = [ (datetime.date(2004, 8,  1), datetime.date(2004, 11, 14)),
                      (datetime.date(2003,2,28),datetime.date(2003,5,30)),
                      (datetime.date(2004,2,28),datetime.date(2004,5,30)),
                    ]
        def test(cc, testdates, expected):
            for (s, e), expect in zip(testdates, expected):
                print 'total holidays in %s from %s to %s is %d(exp %d)' % (
                              cc,s,e,holidays(cc,s,e,test_file.name),expect)
                print
        test('US', testdates, (1,1,1) )
        test('IT', testdates, (1,2,2) )
        import os
        os.remove(test_file.name)
```

讨论

我曾经工作的一家公司里有 3 个不同的工会，根据契约，这 3 个工会也有不同的节假日。同时，我们还得考虑诸如下雪天（snow days）或其他类型的休假，基本上它们也可以等同于节日。为了处理所有这些可能发生变化的节假日，我们最好将标准的节日计算抽取出来，封装成单个的函数，比如我们完成了 all_easter，all_labor，等等。不同类型的计算可以在需要的时候调用。

虽然半开区间（左边界是包括的，但右边界却并不包括）在 Python 中是允许的（而且，在计算上也更具灵活性，出问题的可能也更小），本节仅仅处理闭区间（左右边界都包括）。而那也正是日期区间的概念所定义的，dateutil 也正是基于这个概念工作，所以，这是一个很显然的选择。

每个函数都负责确保返回的结果满足我们的需要：返回在两个日期（含）之间的 datetime.date 实例的列表。举个例子，比如 all_labor，我们用 date 方法，强制将 dateutil 的 rrule 返回的 datetime.datetime 结果转化为 datetime.date 实例。

公司也可能会将某天设定为一个节假日（比如下雪天），“只此一次”，并可能会用一个文本文件来记录这些特殊的日子。在本例中，read_holidays 函数执行读取文本文件的任务，每行读取一个日期，格式为年、月、日。也可以选择将这个函数重构成更“模糊”的日期解析器，如同第 3.7 节中展示的那样。

如果每次运行程序都需要查询节日很多次，可能想做一些优化工作，让它只读取和解析一次，然后每次在需要的时候调用这些解析结果。不过，“过早优化是万恶之源，”Knuth 引用 Hoare 的话这么说过：通过避免这种“显然的”优化，我们获得了清晰和灵活。假设这些函数是在交互式环境中调用的，而日期文件有可能在两次读取之间被编辑和修改：如果我们每次都不做任何假设地读取整个文件，则完全不需要检查自从上次读取之后文件有无做过任何修改。

由于不同的国家庆祝不同的节日，本节例子中提供了一个很基本的 holidays_by_country 字典。可以在因特网上做很多调查并根据需要不断地充实这个字典。很重要的一点是，这个字典允许调用不同的节日函数，根据传递给 holidays 函数的不同的国家编码来确定。如果你的公司有很多工会，也可以很容易地创建一个基于工会的字典，传递工会编码而非国家编码给 holidays。holidays 函数再去调用合适的函数（包括了无条件调用的 read_holidays 函数），连接所有的结果，清除重复内容，然后返回列表的长度。如果你乐意，也可以直接返回列表，只需简单地让代码中注释掉的两行开始工作即可。

更多资料

3.7 节中的模糊解析；dateutil 的文档，见 https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil，*Library Reference* 中的 datetime 文档。

3.7 日期的模糊查询

感谢：Andrea Cavalcanti

任务

你的程序需要读取并接受一些并不符合标准的“yyyy, mm, dd”datetime 格式。

解决方案

第三方 dateutil.parser 模块给出了一个简单的解答：

```
import datetime
import dateutil.parser
```

```
def tryparse(date):
     # dateutil.parser 需要一个字符串参数：根据一些惯例，我们
     # 可以从 4 种 “date” 参数创建一个
     kwargs = {  }                                        # 假设没有命名参数
       if isinstance(date, (tuple, list)):
            date = ''.join([str(x) for x in date]) # 拼接序列
       elif isinstance(date, int):
            date = str(date)                              # 将整数变为字符串
       elif isinstance(date, dict):
            kwargs = date                                 # 接受命名参数字典
            date = kwargs.pop('date')                     # 带有一个'date'字符串
       try:
            try:
                 parsedate = dateutil.parser.parse(date, **kwargs)
                 print 'Sharp %r -> %s' % (date, parsedate)
            except ValueError:
                 parsedate=dateutil.parser.parse(date,fuzzy=True,**kwargs)
                 print 'Fuzzy %r -> %s' % (date, parsedate)
      except Exception, err:
       print 'Try as I may, I cannot parse %r (%s)' % (date, err)
if __name__ == "__main__":
     tests = (
               "January 3, 2003",                         # 字符串
               (5, "Oct", 55),                            # 元组
               "Thursday, November 18",                   # 没有年的长字符串
               "7/24/04",                                 # 带斜线的字符串
               "24-7-2004",                               # 欧式的字符串格式
               {'date':"5-10-1955","dayfirst":True}, # 包括了 kwarg 的字典
               "5-10-1955",                               # 日在前，无 kwarg
               19950317,                                  # 非字符串
               "11AM on the 11th day of 11th month,in the year of our Lord 1945",
               )
     for test in tests:                                   # 测试日期格式
          tryparse(test)                                  # 尝试解析
```

讨论

dateutil.parser 的 parse 函数可以用于很多数据格式。本节代码中展示了其中的一部分。这个解析器可以处理英语的月名以及两位或者四位的年（带有一些限制）。如果不带名字参数调用 parse，它首先尝试用如下的顺序来解析字符串：mm-dd-yy。如果解析的结果不合逻辑，正如例子中的那样，它尝试解析 “27-7-2004” 这样的字符串，无果。最后它会尝试 “yy-mm-dd”。如果传入了 dayfirst 或 yearfirst 这样的 “关键字”（我们在测试中正是这样做的），parse 会试图根据关键字进行解析。

本节的测试代码定义了解析器可能会碰到的一些边界测试用例，比如通过一个元组，一个整数（ISO 格式，无空格），甚至一个短语来传入日期。为了测试关键字参数，本

节的 tryparse 函数也接受一个字典作为参数，该函数找到“date”键对应的值作为解析的对象，并将其余部分作为关键字参数传递给 dateutil 的解析器。

dateutil 的解析器能够提供一定程度的“模糊”解析，只要你能够给它一点提示以便确定各部分的处理方式，比如小时（测试中所用的短语包含了 AM）。在正式的编写代码工作中，应该避免依赖模糊解析，或者做一些预处理工作，或者至少提供某种机制来检查需要解析的日期的准确性。

更多资料

更多的有关日期解析算法的资料，可参看 dateutil 的文档 https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil；而关于日期处理部分，可参看 *Library Reference* 中 datetime 的文档。

3.8 检查夏令时是否正在实行

感谢：Doug Fort

任务

你想了解当前时区的夏令时是否正在执行。

解决方案

你第一个想法可能是去检查 time.daylight，但很遗憾，那不可行。应该这样做：

```
import time
def is_dst( ):
     return bool(time.localtime( ).tm_isdst)
```

讨论

在我的时区（和大多数其他时区一样），time.daylight 永远是 1，因为 time.daylight 的含义是，该时区是否在一年中的某些时候会执行夏令时制（Daylight Saving Time, DST），这和当前是否正在实行夏令时制无关。

只有当 DST 正在实行的时候，调用 time.localtime 获得元组的最后一项的值才是 1，否则会是 0——这正好就是我们需要检查的。本节将这个检查操作封装为一个函数，调用了内建的 bool 类型来确保返回结果是一个优雅的 true 或者 false，而不是一个粗糙的 1 或者 0——这部分改良不是必须的，但我认为效果还不错。也可以直接访问相关的项，比如 time.localtime()[-1]，但是通过 tm_isdst 属性来获取信息显然可读性会更好。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 time 模块的内容。

3.9 时区转换

感谢：Gustavo Niemeyer

任务

假设你现在身处西班牙，你想将发生在中国的某个事件的时间转换为西班牙时间。

解决方案

第三方包 dateutil 中对 datetime 提供了时区支持。下面给出一个设置本地时区的方法，并最后打印出当前时间来检查结果的正确性：

```
from dateutil import tz
import datetime
posixstr = "CET-1CEST-2,M3.5.0/02:00,M10.5.0/03:00"
spaintz = tz.tzstr(posixstr)
print datetime.datetime.now(spaintz).ctime( )
```

不同时区之间的转换不仅是可能的，而且对于我们来说也是必要的。举个例子，让我们看看如果根据西班牙时间，下一届奥运会什么时候开始：

```
chinatz = tz.tzoffset("China", 60*60*8)
olympicgames = datetime.datetime(2008, 8, 8, 20, 0, tzinfo=chinatz)
print olympicgames.astimezone(spaintz)
```

讨论

名为 posixstr 的字符串是西班牙时区的 POSIX 风格的表示法。这个字符串提供了标准时和夏令时的名字（CST 和 CEST），它们的偏移量（UTC+1 和 UTC+2），以及 DST 开始和结束的确切时间（三月最后一个星期天的凌晨 2 点，以及十月最后一个星期天的凌晨 3 点）。我们还可以检查一下 DST 时区的边界以确保其正确性：

```
assert spaintz.tzname(datetime.datetime(2004, 03, 28, 1, 59)) == "CET"
assert spaintz.tzname(datetime.datetime(2004, 03, 28, 2, 00)) == "CEST"
assert spaintz.tzname(datetime.datetime(2004, 10, 31, 1, 59)) == "CEST"
assert spaintz.tzname(datetime.datetime(2004, 10, 31, 2, 00)) == "CET"
```

所有的这些 asserts 都应该能顺利地通过测试，从而确保时区名在这些不同时间之间的切换是成立的。

注意返回到标准时的时间是凌晨 3 点，而实际的切换时间点却被记为凌晨 2 点。这中间有一个小时的差异，凌晨 2 点和凌晨 3 点，未能保持一致。这种情况发生了两次：

一次是在 CEST 时区，还有一次是 CET 时区。目前，通过标准的 Python 日期和时间支持来无歧义地表示这个时刻还不可能。这就是为什么我会推荐你存储无歧义的 UTC 的 datetime 实例，并为了显示的目的而进行时区转换。

为了将中国时区转换成西班牙时区，我们使用了 tzoffset 来记录这个事实，即中国比 UTC 时间提前了 8 个小时（tzoffset 总是用来和 UTC 比较，而不是和某个特定的时区比较）。注意我们是如何根据时区信息创建出 datetime 实例的。对于两个不同时区之间的转换，即使给出的时间是基于本地时区的，这一步也总是很必要的。如果你不根据时区信息创建实例，你会得到一个 ValueError: astimezone( ) cannot be applied to a naive datetime。在没有时区信息的情况下，创建的 datetime 实例会被简化处理——完全忽略掉时区信息。基于这个目的，dateutil 提供了 tzlocal 类型，创建基于当前平台的本地时区的实例。

除了前面提到的类型，dateutil 还提供了 tzutc，创建基于 UTC 的实例；tzfile，可以使用标准的二进制时区文件；tzical，创建基于 iCalendar 时区的实例；当然还有更多其他类型，这里就不详细介绍了。

更多资料

dateutil 模块的文档，见 https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil，以及 *Library Reference* 中 datetime 的文档。

3.10 反复执行某个命令

感谢：Philip Nunez

任务

需要反复地以需要的时间间隔执行某个命令。

解决方案

time.sleep 函数提供了一个简单的解决办法：

```
import time, os, sys
def main(cmd, inc=60):
     while True:
          os.system(cmd)
          time.sleep(inc)
if __name__ == '__main__' :
     numargs = len(sys.argv) - 1
     if numargs < 1 or numargs > 2:
          print "usage: " + sys.argv[0] + " command [seconds_delay]"
          sys.exit(1)
```

```
cmd = sys.argv[1]
if numargs < 3:
     main(cmd)
else:
     inc = int(sys.argv[2])
     main(cmd, inc)
```

讨论

可以用本节中的方法来周期性地运行某命令，以完成某种检查（比如轮询），或者执行不断重复的操作，比如让浏览器加载某个内容不断发生变化的 URL，这样可以确保目前的浏览结果是较新的。本节代码创建了一个叫做 main 的函数，还有一个由 if __name__=='__main__':限定的主体部分，这部分只有在脚本作为主脚本运行时才会被执行。主体部分检查命令行的参数，并调用 main 函数（或者输出使用方法的信息，如果给的参数数目不对的话）。这是很好的脚本编写结构，同时也使得它的功能可以被其他脚本通过模块导入的方式调用。

main 函数接受一个叫做 cmd 的字符串参数，那是你想传递给操作系统 shell 用于执行的命令，还有一个可选的时间周期，默认是 60 秒（1 分钟）。main 函数主体处于永久的循环之中，它或者用 os.system 来执行命令，或者通过 time.sleep 来等待（无须消耗资源）。

脚本的主体会检查你传递给脚本的命令行参数，它通过访问 sys.argv 来获取那些参数。第一个参数，sys.argv[0]，是脚本的名字，当脚本需要确定自己的身份并打印信息时这是很有用的参考。脚本的主体检查一到两个参数。第一个（强制性的）是需要运行的命令。（你可能需要用引号将命令括起来，这是为了防止 shell 解析命令时，将运行命令的参数和脚本的参数混淆起来：最重要的是，加了引号之后，无论里面的内容是什么样的，它都是一个单一的参数。）第二个（可选的）参数是两次运行之间的间隔秒数。如果忽略第二个参数，主体部分会用默认的间隔时间（60s）为周期来调用传入的命令。

注意，如果有第二个参数，主体部分会将它由字符串（sys.argv 中所有的元素都是字符串）转化为整数，可以通过调用内建的 int 类型完成这一转换：

```
inc = int(sys.argv[2])
```

如果第二个参数是一个无法转换成整数的字符串（比如，它是一个不含数字的字符序列），用上面的方式调用 int 会产生一个异常，然后脚本的执行会结束并打印出一些错误信息。正如 Python 的设计哲学那样，“除非明确地指定，错误不应该悄无声息地被略过。”所以，如果允许脚本接受任何字符串作为第二个参数，并在发生转换错误时采取默认的行动，那其实不是一个很好的设计。

如果不愿使用循环和 time.sleep，本章还有一些涉及 Python 标准库模块 sched 的内容，见第 3.11 节。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于标准库模块 os、time 和 sys 的文档；3.11 节。

3.11 定时执行命令

感谢：Peter Cogolo

任务

需要在某个确定的时刻执行某个命令。

解决方案

这正是标准库的 sched 模块所针对的任务：

```
import time, os, sys, sched
schedule = sched.scheduler(time.time, time.sleep)
def perform_command(cmd, inc):
     schedule.enter(inc, 0, perform_command, (cmd, inc)) # re-scheduler
     os.system(cmd)
def main(cmd, inc=60):
     schedule.enter(0, 0, perform_command, (cmd, inc))   # 0==right now
     schedule.run( )
if __name__ == '__main__' :
     numargs = len(sys.argv) - 1
     if numargs < 1 or numargs > 2:
          print "usage: " + sys.argv[0] + " command [seconds_delay]"
          sys.exit(1)
     cmd = sys.argv[1]
     if numargs < 3:
          main(cmd)
     else:
          inc = int(sys.argv[2])
          main(cmd, inc)
```

讨论

上述代码提供了和第 3.10 节中代码类似的功能，只不过它没有使用一个简单的轮询，而是使用了标准库的 sched 模块。

sched 是一个简单灵活却又非常强大的模块，可被用来在未来某个时刻执行某些特定的任务。要使用 sched，首先需要创建一个 scheduler 实例对象，即代码中的 schedule，创建时要带有两个参数。第一个参数是一个被用来调用的函数，此函数确定任务的时间——通常是 time.time，该函数返回从某个特定的时间点到现在所经历的秒数。第二

个参数也是一个供调用的函数，用来等待一段时间——通常是 time.sleep。也可以传递其他函数，该函数以某种人为的方式衡量时间。举个例子，可以将 sched 用于一些模拟程序。不过，以人为的方式干预时间的衡量属于 sched 的较高级的应用，本节并未覆盖这方面内容。

一旦有了一个 sched.scheduler 实例 s，可以通过调用 s.enter 来安排某事件的发生时间，即从现在起的第 n 秒开始启动（也可以将 n 设为 0，这样事件会立即发生），或者调用 s.enterabs，以某个给定的绝对时间作为事件启动时间。对于这两种方式，都需要传递时间（相对时间或者绝对时间），优先级（如果有多个事件被计划在同一时间执行，它们必须按照优先级顺序执行，值最小的会最先执行），一个供调用的函数，一个容纳前面函数所需的参数的元组。这两个调用方式都会返回一个事件标识，可以将这个标识存在什么地方，如果你改变了注意，还可以轻易地取消这个计划中的事件——只需将事件标识作为参数，调用 s.cancel。不过这又是一个较高级的用法，本节并未涵盖。

安排了一些事件之后，可以调用 s.run，它会持续运行，直到计划事件队列变成空的为止。在本节中，我们展示了怎样计划一个周期性反复执行的事件：函数 perform_command 每次被执行，首先都会安排在 inc 秒之后再次运行它自己，然后才运行指定的系统命令。以这种方式安排任务，计划事件队列永远不会变空，函数 perform_command 不断地被周期性调用。这种自我计划安排是一个很重要的概念，不仅仅用于和 sched 相关的应用，任何时候，如果你只有一次机会来计划一个事件，而没有周期性的机会来反复做此事，你都可以考虑这种自我计划安排方式。（举个例子，Tkinter 的 after 方法也正是以此种方式工作的，可以作为这种自我计划安排的一个应用典型）。

即使对于本节所示这么简单的例子，相比于简单轮询，sched 的优势也极其明显。在第 3.10 节中，指定的延迟时间是指从本次 cmd 的执行完成之后，到下一次 cmd 执行之前。如果 cmd 的执行需要较多的时间（这很有可能，比如，某些命令可能需要等待网络返回数据，或者由于某些繁忙的服务器，而不得不等待），那么这些命令其实并不是真正的“周期性的”执行。而本节的时间延迟是指从本次开始运行 cmd，到一下次执行 cmd 之前，因此周期性是被严格保障的。如果某个特定命令所用的时间超过了 inc 秒，那么 schedule 会暂时落后于进度，但最终它会慢慢赶上来，只要 cmd 的平均执行时间不超过 inc 秒：sched 永远不会“略过”事件。（如果你确实需要略过某事件，假设这个事件对你而言不再重要了，必须保存原先获得的事件标识，并调用 cancel 方法来取消此事件。）

关于本节的结构和主体的一些细节的解释，请参考第 3.10 节。

更多资料

第 3.10 节；*Library Reference* 和 *Python in a Nutshell* 中的标准库模块 os、time、sys 和 sched 的文档。

3.12 十进制数学计算

感谢：Anna Martelli Ravenscroft

任务

需要在 Python 2.4 中进行一些简单的数学计算，但需要的是十进制的结果，而不是 Python 默认的 float 类型。

解决方案

为了从这些简单的计算中获得预期的结果，必须导入 decimal 模块：

```
>>> import decimal
>>> d1 = decimal.Decimal('0.3')   # 指定一个十进制对象
>>> d1/3                          # 试试除法
Decimal("0.1")
>>> (d1/3)*3                      #  看看能否复原？
Decimal("0.3")
```

讨论

Python 的新手（尤其是那些没有使用其他语言进行二进制浮点计算经验的）常常会惊异于一些简单计算的结果。举个例子：

```
>>> f1 = .3                       # 赋值为一个浮点数
>>> f1/3                          # 试试除法
0.099999999999999992
>>> (f1/3)*3                      # 看看能否复原？
0.29999999999999999
```

Python 采用二进制浮点数学计算作为默认的计算方式是有很好的理由的。可以读一读 Python FAQ（Frequently Asked Questions）文档，见 http://www.python.org/doc/faq/general.html#why-are-floating-point-calculations-so-inaccurate，或者 *Python Tutorial* 的附录部分，见 http://docs.python.org/tut/node15.html。

很多人对于唯一的选择——二进制浮点，并不满意，他们希望能够指定精度，能够使用十进制数学计算，以便用于财务计算并生成可预期的结果。很多人需要的仅仅是可预测性。（一个真正的数值分析专家，当然会认为二进制浮点的计算其实是完全可以预测的。如果你们三人的某人正在读本章节，请跳过此节继续下一节，谢谢。）（译者注：我猜作者这句话是对她的三个专家朋友说的，其他读者略过就好。）

新的 decimal 类型可以通过环境上下文设置进行深入的控制，这使得可以简单地设置精度以及对结果的截取方法。不过，如果你所需要的只是让那些简单的数学计算返回可预测的结果，那么 decimal 默认的环境已经工作得很好了。

有几点请记住：可以传递字符串、整数、元组或者其他 decimal 对象来创建一个新的 decimal 对象，但如果你有一个浮点数 n，你想通过它来创建一个 decimal 对象，请确保传递的是 str(n)，而不是 n。decimal 对象可以和其他整数、长整数、decimal 对象交互（比如，混用于数学运算），但 float 类型不行。这些限制是强制性的。十进制数被引入到 Python 中，是因为它能提供 float 无法提供的精确度和可预测性：如果允许直接从 float 对象构建十进制数，或者允许二者混用于数学计算，就完全违背了设计十进制数的初衷。另一方面，正如你所预料的那样，decimal 对象也可以被强制转化为 float，long 以及 int 类型。

请记住，decimal 仍然是浮点，而非固点。如果需要的是固点，请参考 Tim Peters 的 FixedPoint，见 http://fixedpoint.sourceforge.net/。另外，Python 中也没有专门的 money 数据类型，不过可以参考 3.13 节，了解如何基于 decimal 建立你自己的财务数据格式。最后一点，也许不是那么明显（至少对我而言），一个中间计算结果产生了比输入更多的数字，可以保留那些多余的数字，并用于进一步的计算，到最后完成了所有计算（准备显示或者存储结果时）时，再对结果进行截取，或者干脆每一步都可以完成截取。对于这一点，不同的书有不同的建议。不过我倾向于前者，因为它至少更加方便。

如果你还在坚持 Python 2.3，可以通过下载和安装第三方扩展来利用 decimal 模块，见 http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html。

更多资料

Python Tutorial 的附录 B 中对于浮点数学运算的解释，见 http://docs.python.org/tut/node15.html；Python FAQ，http://www.python.org/doc/faq/general.html#why-are-floating-point-calculations-so-inaccurate；Tim Peter 的 FixedPoint，见 http://fixedpoint.sourceforge.net/；将 decimal 用于货币，参考第 3.13 节；decimal 已经被记录在 Python 2.4 的 *Library Reference* 文档中了，2.3 的使用者也可通过下载安装使用，见 http://cvs.sourceforge.net/viewcvs.py/python/python/dist/src/Lib/decimal.py；decimal PEP（Python Enhancement Proposal），PEP 327，见 http://www.python.org/peps/pep-0327.html。

3.13 将十进制数用于货币处理

感谢：Anna Martelli Ravenscroft,、Alex Martelli、Raymond Hettinger

解决方案

我们可以使用新的 decimal 模块，配以一个修改过的 moneyfmt 函数（原始版本由 Raymond Hettinger 编写，那是 Python 标准库中 decimal 的文档的组成部分）：

```
import decimal
""" 计算意大利支票税 """
```

```
def italformat(value, places=2, curr='EUR', sep='.', dp=',', pos='', neg='-',
                    overall=10):
     """ 将十进制 value 转化为财务格式的字符串
     places:  十进制小数点后面的数字的位数
     curr:    可选的货币符号（可能为空）
     sep:     可选的分组（三个一组）分隔符（逗号、句号或空白）
     dp:      小数点指示符（逗号或句号）；当 places 是 0 时
                  小数点被指定为空白
     pos:     正数的可选的符号："+"，空格或空白
     neg:     正数的可选的符号："-"、"("，空格或空白
     overall:  最终结果的可选的总长度，若长度不够，左边货币符号
                  和数字之间会被填充符占据
     """
     q = decimal.Decimal((0, (1,), -places))          # 2 places --> '0.01'
     sign, digits, exp = value.quantize(q).as_tuple( )
     result = [ ]
     digits = map(str, digits)
     append, next = result.append, digits.pop
     for i in range(places):
          if digits:
               append(next( ))
          else:
               append('0')
     append(dp)
     i = 0
     while digits:
          append(next( ))
          i += 1
          if i == 3 and digits:
               i = 0
               append(sep)
     while len(result) < overall:
          append(' ')
     append(curr)
     if sign: append(neg)
     else: append(pos)
     result.reverse( )
     return ''.join(result)
# 获得计算用的小计
def getsubtotal(subtin=None):
     if subtin == None:
          subtin = input("Enter the subtotal: ")
     subtotal = decimal.Decimal(str(subtin))
     print "\n subtotal:                ", italformat(subtotal)
     return subtotal
# 指定意大利税法函数
def cnpcalc(subtotal):
     contrib = subtotal * decimal.Decimal('.02')
     print "+ contributo integrativo 2%:    ", italformat(contrib, curr='')
     return contrib
```

```
def vatcalc(subtotal, cnp):
     vat = (subtotal+cnp) * decimal.Decimal('.20')
     print "+ IVA 20%:                    ", italformat(vat, curr='')
     return vat
def ritacalc(subtotal):
     rit = subtotal * decimal.Decimal('.20')
     print "-Ritenuta d'acconto 20%:       ", italformat(rit, curr='')
     return rit
def dototal(subtotal, cnp, iva=0, rit=0):
     totl = (subtotal+cnp+iva)-rit
     print "                  TOTALE: ", italformat(totl)
     return totl
# 最终计算报告
def invoicer(subtotal=None, context=None):
     if context is None:
          decimal.getcontext( ).rounding="ROUND_HALF_UP" # 欧洲截断规则
     else:
          decimal.setcontext(context)                    # 设置上下文环境
     subtot = getsubtotal(subtotal)
     contrib = cnpcalc(subtot)
     dototal(subtot, contrib, vatcalc(subtot, contrib), ritacalc(subtot))
if __name__=='__main__':
     print "Welcome to the invoice calculator"
     tests = [100, 1000.00, "10000", 555.55]
     print "Euro context"
     for test in tests:
          invoicer(test)
     print "default context"
     for test in tests:
          invoicer(test, context=decimal.DefaultContext)
```

讨论

意大利的税务计算颇有点繁琐，比本节展示的情况要复杂多了。本节提供的代码仅仅适用于意大利使用发票的用户。我早就厌倦手工处理发票了，所以我写了个简单的脚本来完成计算。最终，在重构之后代码变成了现在这样，它使用了新的 decimal 模块，并且遵循一个财务计算的原则，那就是永远不使用二进制浮点。

怎么才能尽量地利用 decimal 模块为财务计算服务呢？十进制的数学运算非常直接简单，虽然显示结果时所用的选项没有那么清楚明白。本节的 italformat 函数基于 Raymond Hettinger 的 moneyfmt 函数，你可在 Python 2.4 的 *Library Reference* 的 decimal 模块章节找到。为了更好地生成财报，代码有一些小小的修改。最主要的变化是 overall 参数。这个参数使得函数创建一个由 overall 指定数字位数的 decimal，并在货币符号（如果有的话）和数字之间填充空白符。这将有助于生成标准的预期长度的结果。

注意，我把 subtotal = decimal.Decimal(str(subtin))中的 subtin 强制转化为一个字符串。这将允许 getsubtotal 函数接受浮点数（当然也包括整数和字符串）为参数——而无须担

心浮点数可能会引发异常。如果你的程序还可能处理元组，也可以重构代码以适应那种需求。对我来说，浮点是 getsubtotal 最可能遇到的参数类型，我无须担心元组。

当然，如果想显示美元货币符号，并使用不同的截断规则，也可以很容易地修改代码以符合你的需求。举个例子，为了显示美元符号，只需将 curr，sep 和 dp 的默认值修改为：

```
def USformat(value, places=2, curr='$', sep=',', dp='.', pos='', neg='-',
             overall=10):
...
```

如果经常需要处理很多不同的货币，也可以重构此函数，使得它可以在字典中寻找一些合适的参数，或者能够用其他方法给它传递正确的参数。在理论上，使用 Python 标准库中的 locale 模块应当是最好的用于确定使用者喜好以及关于财务信息的方法，但实际上我觉得使用 locale 并不能给我帮什么忙，不过在这里，我很希望你能够独立实现这个额外的任务，作为检验学习成果的一个练习。

不同的国家通常有不同的截断规则：decimal 使用 ROUND_HALF_EVEN 作为默认的截断方式。然而，欧洲的规矩应当是 ROUND_HALF_UP。为了使用不同的截断规则，如代码所示，只需修改上下文环境即可。最终结果可能不会有太大的变化，但是必须要清楚，修改截断规则（可能影响很小，但是不能忽略）的确会引起差异。

还可以更深入地修改上下文环境，可以创建并设置自己的 context 类实例。无论你以哪种方式影响环境，通过简单的 getcontext 修改属性，或者通过 setcontext(mycontext)传入一个定制的 context 类实例，这些修改对于活动的线程都会即时生效，直到你再次修改上下文环境。如果你考虑要在正式的工作代码（或者为了你自己的家务管理目的）中使用 decimal，请确保根据你的国家的账务统计惯例使用正确的上下文环境（即，正确的截断规则）。

更多资料

参考 Python 2.4 的 *Library Reference* 关于 decimal 的文档，尤其是 decimal.context 部分。

3.14 用 Python 实现的简单加法器

感谢：Brett Cannon

任务

你想用 Python 制作一个简单的加法器，使用精确的十进制数（而不是二进制浮点数），并以整洁的纵列显示计算结果。

解决方案

为了执行计算，必须使用 decimal 模块。我们的程序接受输入行，每行由一个数字紧跟

一个数学运算符组成，空行表示收到了计算当前结果的请求，输入 q 则表示结束当前程序：

```
import decimal, re, operator
parse_input = re.compile(r'''(?x)      # 允许 RE 中的注释和空白符
                (\d+\.?\d*)            # 带有可选的小数部分的数
                \s*                    # 可选的空白符
                ([-+/*])               # 运算符
                $''')                  # 字符串结束
oper = { '+': operator.add, '-': operator.sub,
           '*': operator.mul, '/': operator.truediv,
         }
total = decimal.Decimal('0')
def print_total( ):
     print '==  ==  =\n', total
print """Welcome to Adding Machine:
Enter a number and operator,
an empty line to see the current subtotal,
or q to quit: """
while True:
     try:
          tape_line = raw_input( ).strip( )
     except EOFError:
          tape_line = 'q'
     if not tape_line:
          print_total( )
          continue
     elif tape_line == 'q':
          print_total( )
          break
     try:
          num_text, op = parse_input.match(tape_line).groups( )
     except AttributeError:
          print 'Invalid entry: %r' % tape_line
          print 'Enter number and operator, empty line for total, q to quit'
          continue
     total = oper[op](total, decimal.Decimal(num_text))
```

讨论

Python 的交互式解释器是很有用的计算器，但一个简单的“加法器”也还是有用的。举个例子，像 2345634+2894756-2345823 这样的表达式很不易读，所以检查用于计算的输入数字并不像想象的那么简单。而一个简单的加法器可以用一种很整洁的、纵列的方式显示数字，这样可以很容易地多次检查你的输入。另外，decimal 模块使用一种基于十进制的计算方式，那也正是我们需要的，我们很多时候并不需要科学家、工程师或计算机所偏爱的浮点数学计算。

当在命令行 shell 下运行此脚本时（此脚本并不是为在 Python 交互式解释器中运行而设

计的），它会给出使用提示，然后一直等待输入。可以敲入一个数字（一位或者多位数字，加上一个可选的小数点，然后是可选的小数位数字），紧跟一个运算符（/、*、-和+，这 4 个字符可以在键盘的数字区找到），然后回车。脚本会根据运算符将输入的数计入到总数中。当你输入空行时，程序将打印出当前的计算结果。当输入 q 并回车时，程序会退出。这种简单的输入输出的界面符合一个典型的简单加法器的需求。

decimal 包是 Python 2.4 标准库的一个组成部分。如果你使用 Python 2.3，请访问 http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html，下载并安装这个包。decimal 可支持高精度的十进制数学运算，相比于二进制浮点运算，它的应用面更加广泛（比如涉及财务的计算），当然浮点运算是计算机上速度最快的运算，也是 Python 默认的运算方式。你再也无须花费时间和精力去学习难于理解的二进制浮点运算。如 3.13 节所示，可以将默认的截断规则 ROUND_HALF_EVEN 按需求修改成其他规则。

本节的代码非常简单，也有很多可以提升性能的地方。一个有用的加强方法是，将外界输入记录到硬盘以备查用。可以简单地做到这一点——只需要在循环之前加一条语句，打开一个文本文件以供添加信息：

```
tapefile = open('tapefile.txt', 'a')
```

然后，在获得 tape_line 的值的 try/except 语句之后，加一条语句以便将值写入文件之中：

```
tapefile.write(tape_line+'\n')
```

如果做了上面这些操作，也许你还想继续加强 print_total 函数，使得它既能够输出信息到命令行窗口，同时还能将信息写入到文件中。因此，该函数可以被修改为：

```
def print_total( ):
     print '====  =\n', total
     tapefile.write('====  =\n' + str(total) + '\n')
```

file 对象的 write 方法接受一个字符串作为参数，但是并不默认地在字符串末尾加上换行，这和 print 语句的行为方式有些不同，所以我们要明确地调用内建的 str 函数，明确地在末尾增加了一个'\n'。最后，此版本的 print_total 函数的第二条语句还可以被改写为：

```
print >>tapefile, '====  =\n', total
```

有些人非常讨厌这种 print >>> somefile 的语法，但有时这种写法真的很方便，上面的应用就是一个例子。

还有一些可能的改进，比如现在每次输入完运算符之后都需要按回车键，进一步的改进可以不用按回车就直接进行处理（不过这意味着我们需要处理无缓冲的输入，并且每次要单个处理一个字符，而不能使用方便的基于行的内建函数 raw_input，可以参考 2.23 节提供的一种跨平台的处理无缓冲输入的方法），再比如增加一个 clear 函数（或者增加提示功能，如果用户输入 0*会把结果清空成 0），甚至给这个加法器增加一个 GUI 外壳。不过，我准备把这些部分留给读者作为练习。

本节实现中非常重要的一点是 oper 字典，它用运算符（/、*、-和+）作为键，用内建的 operator 模块中的数学运算函数作为键的对应值。这个功能也可以用更冗长一点的方法实现，比如，一长串 if/elif，如下：

```
if op == '+':
     total = total + decimal.Decimal(num_text)
elif op == '-':
     total = total - decimal.Decimal(num_text)
elif op == '*':
     <line_annotation>... and so on ...</line_annotation>
```

不过，Python 的字典用法明显更理想也更方便，而且不会产生重复性的代码，也更容易维护。

更多资料

decimal 在 Python 2.4 的 *Library Reference* 中有详细介绍，Python 2.3 的用户也可以通过下载来使用，见 http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html；也可以参考十进制 PEP 327，见 http://www.python.org/peps/pep-0327.html。

3.15 检查信用卡校验和

感谢：David Shaw、Miika Keskinen

任务

需要检查一个信用卡号是否遵循工业标准 Luhn 校验和算法。

解决方案

Luhn mod 10 是信用卡业检验和的标准。它不是 Python 内建的算法，不过我们可以很容易地实现这个算法：

```
def cardLuhnChecksumIsValid(card_number):
     """ 通过 luhn mod-10 校验和算法检查信用卡号 """
     sum = 0
     num_digits = len(card_number)
     oddeven = num_digits & 1
     for count in range(num_digits):
          digit = int(card_number[count])
          if not (( count & 1 ) ^ oddeven):
               digit = digit * 2
          if digit > 9:
               digit = digit - 9
          sum = sum + digit
     return (sum % 10) == 0
```

讨论

本节代码的原型来自于 Zope 中一个已经不再使用的电子商务程序。

这个程序完成的简单验证工作，可以让你避免提交一个错误的卡号给信用卡提供商，从而节省时间和金钱，因为没人愿意验证一个错误的信用卡号。本节代码的应用面很广，因为很多政府的身份认证号码也使用了 Luhn（modulus 10）算法。

一个完整的信用卡验证的套件，见 http://david.theresistance.net/files/creditValidation.py。

如果喜欢一行代码解决问题，而不是简洁清晰的编码风格，那么我认为，(a)你可能选错了书（*Perl Cookbook* 是那种会让你满意的类型），(b)同时，在想换本书之前，瞧瞧下面这种写法能否让你高兴：

```
checksum = lambda a: (
  10 - sum([int(y)*[7,3,1][x%3] for x, y in enumerate(str(a)[::-1])])%10)%10
```

更多资料

如果你确实喜欢这个单行解决问题的校验和版本，需要找个心理医生。

3.16 查看汇率

感谢：Victor Yongwei Yang

任务

你想周期性地（用 crontab 或者 Windows 计划任务来运行某 Python 脚本）从 Web 获取数据，监视某两种货币之间的兑换比例，并在两者之间的汇率达到某个阈值时发送提醒邮件。

解决方案

这个任务和一系列的从 Web 获取数据的监控任务很类似，它们包括了监视汇率变化、监视股票行情、天气变化等。下面让我们看看，根据加拿大银行的网站提供的报告（一个简单的 CSV（逗号隔开数值），易于解析），怎样实现对美元和加元之间的汇率变化的监视：

```
import httplib
import smtplib
# 在这里配置脚本的参数
thresholdRate = 1.30
smtpServer = 'smtp.freebie.com'
fromaddr = 'foo@bar.com'
toaddrs = 'your@corp.com'
```

```
# 配置结束
url = '/en/financial_markets/csv/exchange_eng.csv'
conn = httplib.HTTPConnection('www.bankofcanada.ca')
conn.request('GET', url)
response = conn.getresponse()
data = response.read()
start = data.index('United States Dollar')
line = data[start:data.index('\n', start)]     # 获得相关行
rate = line.split(',')[-1]                      # 行的最后字段
if float(rate) < thresholdRate:
     # 发送 email
     msg = 'Subject: Bank of Canada exchange rate alert %s' % rate
     server = smtplib.SMTP(smtpServer)
     server.sendmail(fromaddr, toaddrs, msg)
     server.quit()
conn.close()
```

讨论

在处理外国货币时，自动完成转换的能力是非常有用的。本节用一种简单而直接的方式实现了这个功能。当 cron 运行这个脚本时，脚本先访问网站，获得 CSV，该文件提供了包括今天的最近 7 天的最新汇率：

```
Date (m/d/year),11/12/2004,11/15/2004, ... ,11/19/2004,11/22/2004
$Can/US closing rate,1.1927,1.2005,1.1956,1.1934,1.2058,1.1930,
United States Dollar,1.1925,1.2031,1.1934,1.1924,1.2074,1.1916,1.1844
...
```

然后脚本继续找特定的货币（“United States Dollar”）并读取最后一列的数据，以获得今天的汇率。如果你觉得理解起来还有困难，可以把它一步一步拆解开来，这样可能会更清楚：

```
US = data.find('United States Dollar')     # 寻找货币的索引
endofUSline = data.index('\n', US)         # 找到行末的索引
USline = data[US:endofUSline]              # 切片获取一个字符串
rate = USline.split(',')[-1]               # 根据逗号切割并返回最后的字段
```

本节代码还提供了一个重要功能，当汇率触及阈值时，它会自动发送邮件提醒用户。当然这个阈值也可以随意设置（比如可以设置当汇率跳出了预先设置的阈值范围时发送邮件提醒）。

更多资料

见 *Library Reference* 和 *Python in a Nutshell* 中的 httplib、smtplib 和字符串函数的相关文档。