3.1 数字的四舍五入¶

问题¶

你想对浮点数执行指定精度的舍入运算。

解决方案¶

对于简单的舍入运算，使用内置的 `round(value, ndigits)` 函数即可。比如：

```
>>> round(1.23, 1)
1.2
>>> round(1.27, 1)
1.3
>>> round(-1.27, 1)
-1.3
>>> round(1.25361,3)
1.254
>>>
```

当一个值刚好在两个边界的中间的时候， `round` 函数返回离它最近的偶数。 也就是说，对1.5或者2.5的舍入运算都会得到2。

传给 `round()` 函数的 `ndigits` 参数可以是负数，这种情况下， 舍入运算会作用在十位、百位、千位等上面。比如：

```
>>> a = 1627731
>>> round(a, -1)
1627730
>>> round(a, -2)
1627700
>>> round(a, -3)
1628000
>>>
```

讨论¶

不要将舍入和格式化输出搞混淆了。 如果你的目的只是简单的输出一定宽度的数，你不需要使用 `round()` 函数。 而仅仅只需要在格式化的时候指定精度即可。比如：

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:0.3f}'.format(x)
'value is 1.235'
>>>
```

同样，不要试着去舍入浮点值来”修正”表面上看起来正确的问题。比如，你可能倾向于这样做：

```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.300000000000001
>>> c = round(c, 2) # "Fix" result (???)
>>> c
6.3
>>>
```

对于大多数使用到浮点的程序，没有必要也不推荐这样做。 尽管在计算的时候会有一点点小的误差，但是这些小的误差是能被理解与容忍的。 如果不能允许这样的小误差(比如涉及到金融领域)，那么就得考虑使用 `decimal` 模块了，下一节我们会详细讨论。

3.10 矩阵与线性代数运算¶

问题¶

你需要执行矩阵和线性代数运算，比如矩阵乘法、寻找行列式、求解线性方程组等等。

解决方案¶

NumPy 库有一个矩阵对象可以用来解决这个问题。

矩阵类似于3.9小节中数组对象，但是遵循线性代数的计算规则。下面的一个例子展示了矩阵的一些基本特性：

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2, 3],
        [ 0, 4, 5],
        [ 7, 8, -9]])

>>> # Return transpose
>>> m.T
matrix([[ 1, 0, 7],
        [-2, 4, 8],
        [ 3, 5, -9]])

>>> # Return inverse
>>> m.I
matrix([[ 0.33043478, -0.02608696, 0.09565217],
        [-0.15217391, 0.13043478, 0.02173913],
        [ 0.12173913, 0.09565217, -0.0173913 ]])

>>> # Create a vector and multiply
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],
        [3],
        [4]])
>>> m * v
matrix([[ 8],
        [32],
        [ 2]])
>>>
```

可以在 numpy.linalg 子包中找到更多的操作函数，比如：

```
>>> import numpy.linalg

>>> # Determinant
>>> numpy.linalg.det(m)
-229.99999999999983

>>> # Eigenvalues
>>> numpy.linalg.eigvals(m)
array([-13.11474312, 2.75956154, 6.35518158])

>>> # Solve for x in mx = v
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
>>> m * x
matrix([[ 2.],
        [ 3.],
        [ 4.]])
>>> v
```

```
matrix([[2],
        [3],
        [4]])
>>>
```

讨论¶

很显然线性代数是个非常大的主题，已经超出了本书能讨论的范围。 但是，如果你需要操作数组和向量的话， NumPy 是一个不错的入口点。 可以访问 NumPy 官网 http://www.numpy.org 获取更多信息。

3.11 随机选择¶

问题¶

你想从一个序列中随机抽取若干元素，或者想生成几个随机数。

解决方案¶

`random` 模块有大量的函数用来产生随机数和随机选择元素。 比如，要想从一个序列中随机的抽取一个元素，可以使用 `random.choice()`：

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
>>> random.choice(values)
3
>>> random.choice(values)
1
>>> random.choice(values)
4
>>> random.choice(values)
6
>>>
```

为了提取出N个不同元素的样本用来做进一步的操作，可以使用 `random.sample()`：

```
>>> random.sample(values, 2)
[6, 2]
>>> random.sample(values, 2)
[4, 3]
>>> random.sample(values, 3)
[4, 3, 1]
>>> random.sample(values, 3)
[5, 4, 1]
>>>
```

如果你仅仅只是想打乱序列中元素的顺序，可以使用 `random.shuffle()`：

```
>>> random.shuffle(values)
>>> values
[2, 4, 6, 5, 3, 1]
>>> random.shuffle(values)
>>> values
[3, 5, 2, 1, 6, 4]
>>>
```

生成随机整数，请使用 `random.randint()`：

```
>>> random.randint(0,10)
2
>>> random.randint(0,10)
5
>>> random.randint(0,10)
0
>>> random.randint(0,10)
7
>>> random.randint(0,10)
10
>>> random.randint(0,10)
3
>>>
```

为了生成0到1范围内均匀分布的浮点数，使用 `random.random()`：

```
>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
>>>
```

如果要获取N位随机位(二进制)的整数，使用 `random.getrandbits()`：

```
>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
>>>
```

讨论¶

`random` 模块使用 *Mersenne Twister* 算法来计算生成随机数。这是一个确定性算法， 但是你可以通过 `random.seed()` 函数修改初始化种子。比如：

```
random.seed() # Seed based on system time or os.urandom()
random.seed(12345) # Seed based on integer given
random.seed(b'bytedata') # Seed based on byte data
```

除了上述介绍的功能，random模块还包含基于均匀分布、高斯分布和其他分布的随机数生成函数。 比如， `random.uniform()` 计算均匀分布随机数， `random.gauss()` 计算正态分布随机数。 对于其他的分布情况请参考在线文档。

在 `random` 模块中的函数不应该用在和密码学相关的程序中。 如果你确实需要类似的功能，可以使用ssl模块中相应的函数。 比如， `ssl.RAND_bytes()` 可以用来生成一个安全的随机字节序列。

3.12 基本的日期与时间转换¶

问题¶

你需要执行简单的时间转换，比如天到秒，小时到分钟等的转换。

解决方案¶

为了执行不同时间单位的转换和计算，请使用 `datetime` 模块。 比如，为了表示一个时间段，可以创建一个 `timedelta` 实例，就像下面这样：

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

如果你想表示指定的日期和时间，先创建一个 `datetime` 实例然后使用标准的数学运算来操作它们。比如：

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

在计算的时候，需要注意的是 `datetime` 会自动处理闰年。比如：

```
>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>
```

讨论¶

对大多数基本的日期和时间处理问题， `datetime` 模块已经足够了。 如果你需要执行更加复杂的日期操作，比如处理时区，模糊时间范围，节假日计算等等， 可以考虑使用 dateutil模块

许多类似的时间计算可以使用 `dateutil.relativedelta()` 函数代替。 但是，有一点需要注意的就是，它会在处理月份(还有它们的天数差距)的时候填充间隙。看例子最清楚：

```
>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this function
>>>
>>> from dateutil.relativedelta import relativedelta
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
>>>
>>> # Time between two dates
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>
```

3.13 计算上一个周五的日期¶

问题¶

你需要一个通用方法来计算一周中某一天上一次出现的日期，例如上一个周五的日期。

解决方案¶

Python的 `datetime` 模块中有工具函数和类可以帮助你执行这样的计算。 下面是对类似这样的问题的一个通用解决方案：

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 最后的周五
Desc :
"""
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            'Friday', 'Saturday', 'Sunday']


def get_previous_byday(dayname, start_date=None):
    if start_date is None:
        start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
    days_ago = (7 + day_num - day_num_target) % 7
    if days_ago == 0:
        days_ago = 7
    target_date = start_date - timedelta(days=days_ago)
    return target_date
```

在交互式解释器中使用如下：

```
>>> datetime.today() # For reference
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Previous week, not today
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

可选的 `start_date` 参数可以由另外一个 `datetime` 实例来提供。比如：

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

讨论¶

上面的算法原理是这样的：先将开始日期和目标日期映射到星期数组的位置上(星期一索引为0)， 然后通过模运算计算出目标日期要经过多少天才能到达开始日期。然后用开始日期减去那个时间差即得到结果日期。

如果你要像这样执行大量的日期计算的话，你最好安装第三方包 `python-dateutil` 来代替。 比如，下面是是使用 `dateutil` 模块中的 `relativedelta()` 函数执行同样的计算：

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
```

```
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday
>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
>>>
```

3.14 计算当前月份的日期范围¶

问题¶

你的代码需要在当前月份中循环每一天，想找到一个计算这个日期范围的高效方法。

解决方案¶

在这样的日期上循环并需要事先构造一个包含所有日期的列表。 你可以先计算出开始日期和结束日期， 然后在你步进的时候使用 `datetime.timedelta` 对象递增这个日期变量即可。

下面是一个接受任意 `datetime` 对象并返回一个由当前月份开始日和下个月开始日组成的元组对象。

```
from datetime import datetime, date, timedelta
import calendar

def get_month_range(start_date=None):
    if start_date is None:
        start_date = date.today().replace(day=1)
    _, days_in_month = calendar.monthrange(start_date.year, start_date.month)
    end_date = start_date + timedelta(days=days_in_month)
    return (start_date, end_date)
```

有了这个就可以很容易的在返回的日期范围上面做循环操作了：

```
>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
>>> while first_day < last_day:
...     print(first_day)
...     first_day += a_day
...
2012-08-01
2012-08-02
2012-08-03
2012-08-04
2012-08-05
2012-08-06
2012-08-07
2012-08-08
2012-08-09
#... and so on...
```

讨论¶

上面的代码先计算出一个对应月份第一天的日期。 一个快速的方法就是使用 `date` 或 `datetime` 对象的 `replace()` 方法简单的将 `days` 属性设置成1即可。 `replace()` 方法一个好处就是它会创建和你开始传入对象类型相同的对象。 所以，如果输入参数是一个 `date` 实例，那么结果也是一个 `date` 实例。 同样的，如果输入是一个 `datetime` 实例，那么你得到的就是一个 `datetime` 实例。

然后，使用 `calendar.monthrange()` 函数来找出该月的总天数。 任何时候只要你想获得日历信息，那么 `calendar` 模块就非常有用了。 `monthrange()` 函数会返回包含星期和该月天数的元组。

一旦该月的天数已知了，那么结束日期就可以通过在开始日期上面加上这个天数获得。 有个需要注意的是结束日期并不包含在这个日期范围内(事实上它是下个月的开始日期)。 这个和Python的 `slice` 与 `range` 操作行为保持一致，同样也不包含结尾。

为了在日期范围上循环，要使用到标准的数学和比较操作。 比如，可以利用 `timedelta` 实例来递增日期，小于号<用来检查一个日期是否在结束日期之前。

理想情况下，如果能为日期迭代创建一个同内置的 `range()` 函数一样的函数就好了。 幸运的是，可以使用一个生成器来很容易的实现这个目标：

```
def date_range(start, stop, step):
    while start < stop:
        yield start
        start += step
```

下面是使用这个生成器的例子：

```
>>> for d in date_range(datetime(2012, 9, 1), datetime(2012,10,1),
                        timedelta(hours=6)):
...     print(d)
...
2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
2012-09-02 00:00:00
2012-09-02 06:00:00
...
>>>
```

这种实现之所以这么简单，还得归功于Python中的日期和时间能够使用标准的数学和比较操作符来进行运算。

3.15 字符串转换为日期¶

问题¶

你的应用程序接受字符串格式的输入，但是你想将它们转换为 `datetime` 对象以便在上面执行非字符串操作。

解决方案¶

使用Python的标准模块 `datetime` 可以很容易的解决这个问题。比如：

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)
>>>
```

讨论¶

`datetime.strptime()` 方法支持很多的格式化代码， 比如 `%Y` 代表4位数年份， `%m` 代表两位数月份。 还有一点值得注意的是这些格式化占位符也可以反过来使用，将日期输出为指定的格式字符串形式。

比如，假设你的代码中生成了一个 `datetime` 对象， 你想将它格式化为漂亮易读形式后放在自动生成的信件或者报告的顶部：

```
>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
>>>
```

还有一点需要注意的是， `strptime()` 的性能要比你想象中的差很多， 因为它是使用纯Python实现，并且必须处理所有的系统本地设置。 如果你要在代码中需要解析大量的日期并且已经知道了日期字符串的确切格式，可以自己实现一套解析方案来获取更好的性能。 比如，如果你已经知道所以日期格式是 `YYYY-MM-DD` ，你可以像下面这样实现一个解析函数：

```
from datetime import datetime
def parse_ymd(s):
    year_s, mon_s, day_s = s.split('-')
    return datetime(int(year_s), int(mon_s), int(day_s))
```

实际测试中，这个函数比 `datetime.strptime()` 快7倍多。 如果你要处理大量的涉及到日期的数据的话，那么最好考虑下这个方案！

3.16 结合时区的日期操作¶

问题¶

你有一个安排在2012年12月21日早上9:30的电话会议，地点在芝加哥。 而你的朋友在印度的班加罗尔，那么他应该在当地时间几点参加这个会议呢？

解决方案¶

对几乎所有涉及到时区的问题，你都应该使用 `pytz` 模块。这个包提供了Olson时区数据库， 它是时区信息的事实上的标准，在很多语言和操作系统里面都可以找到。

`pytz` 模块一个主要用途是将 `datetime` 库创建的简单日期对象本地化。 比如，下面如何表示一个芝加哥时间的示例：

```
>>> from datetime import datetime
>>> from pytz import timezone
>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Localize the date for Chicago
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
>>>
```

一旦日期被本地化了， 它就可以转换为其他时区的时间了。 为了得到班加罗尔对应的时间，你可以这样做：

```
>>> # Convert to Bangalore time
>>> bang_d = loc_d.astimezone(timezone('Asia/Kolkata'))
>>> print(bang_d)
2012-12-21 21:00:00+05:30
>>>
```

如果你打算在本地化日期上执行计算，你需要特别注意夏令时转换和其他细节。 比如，在2013年，美国标准夏令时时间开始于本地时间3月13日凌晨2:00(在那时，时间向前跳过一小时)。 如果你正在执行本地计算，你会得到一个错误。比如：

```
>>> d = datetime(2013, 3, 10, 1, 45)
>>> loc_d = central.localize(d)
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> later = loc_d + timedelta(minutes=30)
>>> print(later)
2013-03-10 02:15:00-06:00 # WRONG! WRONG!
>>>
```

结果错误是因为它并没有考虑在本地时间中有一小时的跳跃。 为了修正这个错误，可以使用时区对象 `normalize()` 方法。比如：

```
>>> from datetime import timedelta
>>> later = central.normalize(loc_d + timedelta(minutes=30))
>>> print(later)
2013-03-10 03:15:00-05:00
>>>
```

讨论¶

为了不让你被这些东东弄的晕头转向，处理本地化日期的通常的策略先将所有日期转换为UTC时间， 并用它来执行所有的中间存储和操作。比如：

```
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> utc_d = loc_d.astimezone(pytz.utc)
>>> print(utc_d)
2013-03-10 07:45:00+00:00
>>>
```

一旦转换为UTC，你就不用去担心跟夏令时相关的问题了。 因此，你可以跟之前一样放心的执行常见的日期计算。 当你想将输出变为本地时间的时候，使用合适的时区去转换下就行了。比如：

```
>>> later_utc = utc_d + timedelta(minutes=30)
>>> print(later_utc.astimezone(central))
2013-03-10 03:15:00-05:00
>>>
```

当涉及到时区操作的时候，有个问题就是我们如何得到时区的名称。 比如，在这个例子中，我们如何知道“Asia/Kolkata”就是印度对应的时区名呢？ 为了查找，可以使用ISO 3166国家代码作为关键字去查阅字典 `pytz.country_timezones` 。比如：

```
>>> pytz.country_timezones['IN']
['Asia/Kolkata']
>>>
```

注：当你阅读到这里的时候，有可能 `pytz` 模块已经不再建议使用了，因为PEP431提出了更先进的时区支持。 但是这里谈到的很多问题还是有参考价值的(比如使用UTC日期的建议等)。

3.2 执行精确的浮点数运算¶

问题¶

你需要对浮点数执行精确的计算操作，并且不希望有任何小误差的出现。

解决方案¶

浮点数的一个普遍问题是它们并不能精确的表示十进制数。 并且，即使是最简单的数学运算也会产生小的误差，比如：

```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a + b) == 6.3
False
>>>
```

这些错误是由底层CPU和IEEE 754标准通过自己的浮点单位去执行算术时的特征。 由于Python的浮点数据类型使用底层表示存储数据，因此你没办法去避免这样的误差。

如果你想更加精确(并能容忍一定的性能损耗)，你可以使用 `decimal` 模块：

```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
```

初看起来，上面的代码好像有点奇怪，比如我们用字符串来表示数字。 然而， `Decimal` 对象会像普通浮点数一样的工作(支持所有的常用数学运算)。 如果你打印它们或者在字符串格式化函数中使用它们，看起来跟普通数字没什么两样。

`decimal` 模块的一个主要特征是允许你控制计算的每一方面，包括数字位数和四舍五入运算。 为了这样做，你先得创建一个本地上下文并更改它的设置，比如：

```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.76470588235294117647058823529
>>> with localcontext() as ctx:
...     ctx.prec = 3
...     print(a / b)
...
0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>
```

讨论¶

`decimal` 模块实现了IBM的"通用小数运算规范"。不用说，有很多的配置选项这本书没有提到。

Python新手会倾向于使用 `decimal` 模块来处理浮点数的精确运算。 然而，先理解你的应用程序目的是非常重要的。 如

果你是在做科学计算或工程领域的计算、电脑绘图，或者是科学领域的大多数运算， 那么使用普通的浮点类型是比较普遍的做法。 其中一个原因是，在真实世界中很少会要求精确到普通浮点数能提供的17位精度。 因此，计算过程中的那么一点点的误差是被允许的。 第二点就是，原生的浮点数计算要快的多-有时候你在执行大量运算的时候速度也是非常重要的。

即便如此，你却不能完全忽略误差。数学家花了大量时间去研究各类算法，有些处理误差会比其他方法更好。 你也得注意下减法删除以及大数和小数的加分运算所带来的影响。比如：

```
>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums) # Notice how 1 disappears
0.0
>>>
```

上面的错误可以利用 `math.fsum()` 所提供的更精确计算能力来解决：

```
>>> import math
>>> math.fsum(nums)
1.0
>>>
```

然而，对于其他的算法，你应该仔细研究它并理解它的误差产生来源。

总的来说， `decimal` 模块主要用在涉及到金融的领域。 在这类程序中，哪怕是一点小小的误差在计算过程中蔓延都是不允许的。 因此， `decimal` 模块为解决这类问题提供了方法。 当Python和数据库打交道的时候也通常会遇到 `Decimal` 对象，并且，通常也是在处理金融数据的时候。

3.3 数字的格式化输出¶

问题¶

你需要将数字格式化后输出，并控制数字的位数、对齐、千位分隔符和其他的细节。

解决方案¶

格式化输出单个数字的时候，可以使用内置的 `format()` 函数，比如：

```
>>> x = 1234.56789

>>> # Two decimal places of accuracy
>>> format(x, '0.2f')
'1234.57'

>>> # Right justified in 10 chars, one-digit accuracy
>>> format(x, '>10.1f')
'    1234.6'

>>> # Left justified
>>> format(x, '<10.1f')
'1234.6    '

>>> # Centered
>>> format(x, '^10.1f')
'  1234.6  '

>>> # Inclusion of thousands separator
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
>>>
```

如果你想使用指数记法，将f改成e或者E(取决于指数输出的大小写形式)。比如：

```
>>> format(x, 'e')
'1.234568e+03'
>>> format(x, '0.2E')
'1.23E+03'
>>>
```

同时指定宽度和精度的一般形式是 `'[<>^]?width[,]?(.digits)?'`， 其中 `width` 和 `digits` 为整数，？代表可选部分。 同样的格式也被用在字符串的 `format()` 方法中。比如：

```
>>> 'The value is {:0,.2f}'.format(x)
'The value is 1,234.57'
>>>
```

讨论¶

数字格式化输出通常是比较简单的。上面演示的技术同时适用于浮点数和 `decimal` 模块中的 `Decimal` 数字对象。

当指定数字的位数后，结果值会根据 `round()` 函数同样的规则进行四舍五入后返回。比如：

```
>>> x
1234.56789
>>> format(x, '0.1f')
'1234.6'
>>> format(-x, '0.1f')
'-1234.6'
>>>
```

包含千位符的格式化跟本地化没有关系。 如果你需要根据地区来显示千位符，你需要自己去调查下 `locale` 模块中的函数了。 你同样也可以使用字符串的 `translate()` 方法来交换千位符。比如：

```
>>> swap_separators = { ord('.'):',', ord(','):'.' }
>>> format(x, ',').translate(swap_separators)
'1.234,56789'
>>>
```

在很多Python代码中会看到使用%来格式化数字的，比如：

```
>>> '%0.2f' % x
'1234.57'
>>> '%10.1f' % x
'    1234.6'
>>> '%-10.1f' % x
'1234.6    '
>>>
```

这种格式化方法也是可行的，不过比更加先进的 `format()` 要差一点。 比如，在使用%操作符格式化数字的时候，一些特性(添加千位符)并不能被支持。

3.4 二八十六进制整数¶

问题¶

你需要转换或者输出使用二进制，八进制或十六进制表示的整数。

解决方案¶

为了将整数转换为二进制、八进制或十六进制的文本串， 可以分别使用 `bin()`, `oct()` 或 `hex()` 函数：

```
>>> x = 1234
>>> bin(x)
'0b10011010010'
>>> oct(x)
'0o2322'
>>> hex(x)
'0x4d2'
>>>
```

另外，如果你不想输出 `0b`, `0o` 或者 `0x` 的前缀的话，可以使用 `format()` 函数。比如：

```
>>> format(x, 'b')
'10011010010'
>>> format(x, 'o')
'2322'
>>> format(x, 'x')
'4d2'
>>>
```

整数是有符号的，所以如果你在处理负数的话，输出结果会包含一个负号。比如：

```
>>> x = -1234
>>> format(x, 'b')
'-10011010010'
>>> format(x, 'x')
'-4d2'
>>>
```

如果你想产生一个无符号值，你需要增加一个指示最大位长度的值。比如为了显示32位的值，可以像下面这样写：

```
>>> x = -1234
>>> format(2**32 + x, 'b')
'11111111111111111111101100101110'
>>> format(2**32 + x, 'x')
'fffffb2e'
>>>
```

为了以不同的进制转换整数字符串，简单的使用带有进制的 `int()` 函数即可：

```
>>> int('4d2', 16)
1234
>>> int('10011010010', 2)
1234
>>>
```

讨论¶

大多数情况下处理二进制、八进制和十六进制整数是很简单的。 只要记住这些转换属于整数和其对应的文本表示之间的转换即可。永远只有一种整数类型。

最后，使用八进制的程序员有一点需要注意下。 Python指定八进制数的语法跟其他语言稍有不同。比如，如果你像下面这样指定八进制，会出现语法错误：

```
>>> import os
```

```
>>> os.chmod('script.py', 0755)
    File "<stdin>", line 1
        os.chmod('script.py', 0755)
                              ^
SyntaxError: invalid token
>>>
```

需确保八进制数的前缀是 `0o`，就像下面这样：

```
>>> os.chmod('script.py', 0o755)
>>>
```

3.5 字节到大整数的打包与解包¶

问题¶

你有一个字节字符串并想将它解压成一个整数。或者，你需要将一个大整数转换为一个字节字符串。

解决方案¶

假设你的程序需要处理一个拥有128位长的16个元素的字节字符串。比如：

```
data = b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

为了将bytes解析为整数，使用 `int.from_bytes()` 方法，并像下面这样指定字节顺序：

```
>>> len(data)
16
>>> int.from_bytes(data, 'little')
69120565665751139577663547927094891008
>>> int.from_bytes(data, 'big')
94522842520747284487117727783387188
>>>
```

为了将一个大整数转换为一个字节字符串，使用 `int.to_bytes()` 方法，并像下面这样指定字节数和字节顺序：

```
>>> x = 94522842520747284487117727783387188
>>> x.to_bytes(16, 'big')
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> x.to_bytes(16, 'little')
b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00V4\x12\x00'
>>>
```

讨论¶

大整数和字节字符串之间的转换操作并不常见。 然而，在一些应用领域有时候也会出现，比如密码学或者网络。 例如，IPv6网络地址使用一个128位的整数表示。 如果你要从一个数据记录中提取这样的值的时候，你就会面对这样的问题。

作为一种替代方案，你可能想使用6.11小节中所介绍的 `struct` 模块来解压字节。 这样也行得通，不过利用 `struct` 模块来解压对于整数的大小是有限制的。 因此，你可能想解压多个字节串并将结果合并为最终的结果，就像下面这样：

```
>>> data
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> import struct
>>> hi, lo = struct.unpack('>QQ', data)
>>> (hi << 64) + lo
94522842520747284487117727783387188
>>>
```

字节顺序规则(little或big)仅仅指定了构建整数时的字节的低位高位排列方式。 我们从下面精心构造的16进制数的表示中可以很容易的看出来：

```
>>> x = 0x01020304
>>> x.to_bytes(4, 'big')
b'\x01\x02\x03\x04'
>>> x.to_bytes(4, 'little')
b'\x04\x03\x02\x01'
>>>
```

如果你试着将一个整数打包为字节字符串，那么它就不合适了，你会得到一个错误。 如果需要的话，你可以使用 `int.bit_length()` 方法来决定需要多少字节位来存储这个值。

```
>>> x = 523 ** 23
>>> x
```

```
335381300113661875107536852714019056160355655333978849017944067
>>> x.to_bytes(16, 'little')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> x.bit_length()
208
>>> nbytes, rem = divmod(x.bit_length(), 8)
>>> if rem:
... nbytes += 1
...
>>>
>>> x.to_bytes(nbytes, 'little')
b'\x03X\xf1\x82iT\x96\xac\xc7c\x16\xf3\xb9\xcf...\xd0'
>>>
```

3.6 复数的数学运算¶

问题¶

你写的最新的网络认证方案代码遇到了一个难题，并且你唯一的解决办法就是使用复数空间。 再或者是你仅仅需要使用复数来执行一些计算操作。

解决方案¶

复数可以用使用函数 `complex(real, imag)` 或者是带有后缀j的浮点数来指定。比如：

```
>>> a = complex(2, 4)
>>> b = 3 - 5j
>>> a
(2+4j)
>>> b
(3-5j)
>>>
```

对应的实部、虚部和共轭复数可以很容易的获取。就像下面这样：

```
>>> a.real
2.0
>>> a.imag
4.0
>>> a.conjugate()
(2-4j)
>>>
```

另外，所有常见的数学运算都可以工作：

```
>>> a + b
(5-1j)
>>> a * b
(26+2j)
>>> a / b
(-0.4117647058823529+0.6470588235294118j)
>>> abs(a)
4.47213595499958
>>>
```

如果要执行其他的复数函数比如正弦、余弦或平方根，使用 `cmath` 模块：

```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```

讨论¶

Python中大部分与数学相关的模块都能处理复数。 比如如果你使用 `numpy` ，可以很容易的构造一个复数数组并在这个数组上执行各种操作：

```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j, 4.+5.j, 6.-7.j, 8.+9.j])
>>> a + 2
array([ 4.+3.j, 6.+5.j, 8.-7.j, 10.+9.j])
>>> np.sin(a)
```

```
array([ 9.15449915 -4.16890696j, -56.16227422 -48.50245524j,
        -153.20827755-526.47684926j, 4008.42651446-589.49948373j])
>>>
```

Python的标准数学函数确实情况下并不能产生复数值，因此你的代码中不可能会出现复数返回值。比如：

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

如果你想生成一个复数返回结果，你必须显示的使用 `cmath` 模块，或者在某个支持复数的库中声明复数类型的使用。比如：

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
>>>
```

3.7 无穷大与NaN¶

问题¶

你想创建或测试正无穷、负无穷或NaN(非数字)的浮点数。

解决方案¶

Python并没有特殊的语法来表示这些特殊的浮点值，但是可以使用 `float()` 来创建它们。比如：

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

为了测试这些值的存在，使用 `math.isinf()` 和 `math.isnan()` 函数。比如：

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

讨论¶

想了解更多这些特殊浮点值的信息，可以参考IEEE 754规范。 然而，也有一些地方需要你特别注意，特别是跟比较和操作符相关的时候。

无穷大数在执行数学计算的时候会传播，比如：

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
>>>
```

但是有些操作时未定义的并会返回一个NaN结果。比如：

```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
>>>
```

NaN值会在所有操作中传播，而不会产生异常。比如：

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
```

```
>>> math.sqrt(c)
nan
>>>
```

NaN值的一个特别的地方时它们之间的比较操作总是返回False。比如：

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
>>>
```

由于这个原因，测试一个NaN值得唯一安全的方法就是使用 `math.isnan()`，也就是上面演示的那样。

有时候程序员想改变Python默认行为，在返回无穷大或NaN结果的操作中抛出异常。 `fpectl` 模块可以用来改变这种行为，但是它在标准的Python构建中并没有被启用，它是平台相关的， 并且针对的是专家级程序员。可以参考在线的Python文档获取更多的细节。

3.8 分数运算¶

问题¶

你进入时间机器，突然发现你正在做小学家庭作业，并涉及到分数计算问题。 或者你可能需要写代码去计算在你的木工工厂中的测量值。

解决方案¶

`fractions` 模块可以被用来执行包含分数的数学运算。比如：

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Getting numerator/denominator
>>> c = a * b
>>> c.numerator
35
>>> c.denominator
64

>>> # Converting to a float
>>> float(c)
0.546875

>>> # Limiting the denominator of a value
>>> print(c.limit_denominator(8))
4/7

>>> # Converting a float to a fraction
>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

讨论¶

在大多数程序中一般不会出现分数的计算问题，但是有时候还是需要用到的。 比如，在一个允许接受分数形式的测试单位并以分数形式执行运算的程序中， 直接使用分数可以减少手动转换为小数或浮点数的工作。

3.9 大型数组运算¶

问题¶

你需要在大数据集(比如数组或网格)上面执行计算。

解决方案¶

涉及到数组的重量级运算操作，可以使用 NumPy 库。 NumPy 的一个主要特征是它会给Python提供一个数组对象，相比标准的Python列表而已更适合用来做数学运算。 下面是一个简单的小例子，向你展示标准列表对象和 NumPy 数组对象之间的差别：

```
>>> # Python lists
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # Numpy arrays
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6, 8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

正如所见，两种方案中数组的基本数学运算结果并不相同。 特别的， NumPy 中的标量运算(比如 `ax * 2` 或 `ax + 10`)会作用在每一个元素上。 另外，当两个操作数都是数组的时候执行元素对等位置计算，并最终生成一个新的数组。

对整个数组中所有元素同时执行数学运算可以使得作用在整个数组上的函数运算简单而又快速。 比如，如果你想计算多项式的值，可以这样做：

```
>>> def f(x):
...  return 3*x**2 - 2*x + 7
...
>>> f(ax)
array([ 8, 15, 28, 47])
>>>
```

NumPy 还为数组操作提供了大量的通用函数，这些函数可以作为 math 模块中类似函数的替代。比如：

```
>>> np.sqrt(ax)
array([ 1. , 1.41421356, 1.73205081, 2. ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>>
```

使用这些通用函数要比循环数组并使用 math 模块中的函数执行计算要快的多。 因此，只要有可能的话尽量选择 NumPy 的数组方案。

底层实现中， NumPy 数组使用了C或者Fortran语言的机制分配内存。 也就是说，它们是一个非常大的连续的并由同类型数据组成的内存区域。 所以，你可以构造一个比普通Python列表大的多的数组。 比如，如果你想构造一个

10,000*10,000的浮点数二维网格，很轻松：

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
    array([[ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.],
    ...,
    [ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.]])
>>>
```

所有的普通操作还是会同时作用在所有元素上：

```
>>> grid += 10
>>> grid
array([[ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.],
    ...,
    [ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.],
    [ 10., 10., 10., ..., 10., 10., 10.]])
>>> np.sin(grid)
array([[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    ...,
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
    [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111]])
>>>
```

关于 NumPy 有一点需要特别的主意，那就是它扩展Python列表的索引功能 - 特别是对于多维数组。 为了说明清楚，先构造一个简单的二维数组并试着做些试验：

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1, 2, 3, 4],
[ 5, 6, 7, 8],
[ 9, 10, 11, 12]])

>>> # Select row 1
>>> a[1]
array([5, 6, 7, 8])

>>> # Select column 1
>>> a[:,1]
array([ 2, 6, 10])

>>> # Select a subregion and change it
>>> a[1:3, 1:3]
array([[ 6, 7],
        [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1, 2, 3, 4],
        [ 5, 16, 17, 8],
        [ 9, 20, 21, 12]])

>>> # Broadcast a row vector across an operation on all rows
>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
```

```
        [105, 117, 119, 111],
        [109, 121, 123, 115]])
>>> a
array([[ 1, 2, 3, 4],
        [ 5, 16, 17, 8],
        [ 9, 20, 21, 12]])

>>> # Conditional assignment on an array
>>> np.where(a < 10, a, 10)
array([[ 1, 2, 3, 4],
        [ 5, 10, 10, 8],
        [ 9, 10, 10, 10]])
>>>
```

讨论¶

NumPy 是Python领域中很多科学与工程库的基础，同时也是被广泛使用的最大最复杂的模块。 即便如此，在刚开始的时候通过一些简单的例子和玩具程序也能帮我们完成一些有趣的事情。

通常我们导入 NumPy 模块的时候会使用语句 `import numpy as np` 。 这样的话你就不用再你的程序里面一遍遍的敲入 numpy ，只需要输入 np 就行了，节省了不少时间。

如果想获取更多的信息，你当然得去 NumPy 官网逛逛了，网址是： http://www.numpy.org

第三章：数字日期和时间¶

在Python中执行整数和浮点数的数学运算时很简单的。 尽管如此，如果你需要执行分数、数组或者是日期和时间的运算的话，就得做更多的工作了。 本章集中讨论的就是这些主题。

Contents: