**数字日期和时间**

0x00

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。

## 0x01 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：

1. >>> round(1.23, 1)
2. 1.2
3. >>> round(1.27, 1)
4. 1.3
5. >>> round(-1.27, 1)
6. -1.3
7. >>> round(1.25361,3)
8. 1.254
9. >>>

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：

1. >>> a = 1627731
2. >>> round(a, -1)
3. 1627730
4. >>> round(a, -2)
5. 1627700
6. >>> round(a, -3)
7. 1628000
8. >>>

## 0x02 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：

1. >>> a = 4.2
2. >>> b = 2.1
3. >>> a + b
4. 6.300000000000001
5. >>> (a + b) == 6.3
6. False
7. >>>

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：

1. >>> **from** decimal **import** Decimal
2. >>> a = Decimal('4.2')
3. >>> b = Decimal('2.1')
4. >>> a + b
5. Decimal('6.3')
6. >>> **print**(a + b)
7. 6.3
8. >>> (a + b) == Decimal('6.3')
9. 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：

1. >>> **from** decimal **import** localcontext
2. >>> a = Decimal('1.3')
3. >>> b = Decimal('1.7')
4. >>> **print**(a / b)
5. 0.7647058823529411764705882353
6. >>> with localcontext() as ctx:
7. ...     ctx.prec = 3
8. ...     **print**(a / b)
9. ...
10. 0.765
11. >>> with localcontext() as ctx:
12. ...     ctx.prec = 50
13. ...     **print**(a / b)
14. ...
15. 0.76470588235294117647058823529411764705882352941176
16. >>>

## 0x03 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：

1. >>> x = 1234.56789
3. >>> # Two decimal places of accuracy
4. >>> format(x, '0.2f')
5. '1234.57'
7. >>> # Right justified in 10 chars, one-digit accuracy
8. >>> format(x, '>10.1f')
9. '    1234.6'
11. >>> # Left justified
12. >>> format(x, '<10.1f')
13. '1234.6    '
15. >>> # Centered
16. >>> format(x, '^10.1f')
17. '  1234.6  '
19. >>> # Inclusion of thousands separator
20. >>> format(x, ',')
21. '1,234.56789'
22. >>> format(x, '0,.1f')
23. '1,234.6'
24. >>>

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：

1. >>> format(x, 'e')
2. '1.234568e+03'
3. >>> format(x, '0.2E')
4. '1.23E+03'
5. >>>

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：

1. >>> 'The value is {:0,.2f}'.format(x)
2. 'The value is 1,234.57'
3. >>>

## 0x04 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：

1. >>> x = 1234
2. >>> bin(x)
3. '0b10011010010'
4. >>> oct(x)
5. '0o2322'
6. >>> hex(x)
7. '0x4d2'
8. >>>

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：

1. >>> format(x, 'b')
2. '10011010010'
3. >>> format(x, 'o')
4. '2322'
5. >>> format(x, 'x')
6. '4d2'
7. >>>

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：

1. >>> x = -1234
2. >>> format(x, 'b')
3. '-10011010010'
4. >>> format(x, 'x')
5. '-4d2'
6. >>>

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：

1. >>> x = -1234
2. >>> format(2\*\*32 + x, 'b')
3. '11111111111111111111101100101110'
4. >>> format(2\*\*32 + x, 'x')
5. 'fffffb2e'
6. >>>

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：

1. >>> int('4d2', 16)
2. 1234
3. >>> int('10011010010', 2)
4. 1234
5. >>>

## 0x05 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：

1. 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：

1. >>> len(data)
2. 16
3. >>> int.from\_bytes(data, 'little')
4. 69120565665751139577663547927094891008
5. >>> int.from\_bytes(data, 'big')
6. 94522842520747284487117727783387188
7. >>>

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：

1. >>> x = 94522842520747284487117727783387188
2. >>> x.to\_bytes(16, 'big')
3. b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
4. >>> x.to\_bytes(16, 'little')
5. b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00V4\x12\x00'
6. >>>

## 0x06 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：

1. >>> a = complex(2, 4)
2. >>> b = 3 - 5j
3. >>> a
4. (2+4j)
5. >>> b
6. (3-5j)
7. >>>

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

1. >>> a.real
2. 2.0
3. >>> a.imag
4. 4.0
5. >>> a.conjugate()
6. (2-4j)
7. >>>

另外，所有常见的数学运算都可以工作：

1. >>> a + b
2. (5-1j)
3. >>> a \* b
4. (26+2j)
5. >>> a / b
6. (-0.4117647058823529+0.6470588235294118j)
7. >>> abs(a)
8. 4.47213595499958
9. >>>

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：

1. >>> **import** cmath
2. >>> cmath.sin(a)
3. (24.83130584894638-11.356612711218174j)
4. >>> cmath.cos(a)
5. (-11.36423470640106-24.814651485634187j)
6. >>> cmath.exp(a)
7. (-4.829809383269385-5.5920560936409816j)
8. >>>

## 0x07 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：

1. >>> a = float('inf')
2. >>> b = float('-inf')
3. >>> c = float('nan')
4. >>> a
5. inf
6. >>> b
7. -inf
8. >>> c
9. nan
10. >>>

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

1. >>> math.isinf(a)
2. True
3. >>> math.isnan(c)
4. True
5. >>>

## 0x08 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：

1. >>> **from** fractions **import** Fraction
2. >>> a = Fraction(5, 4)
3. >>> b = Fraction(7, 16)
4. >>> **print**(a + b)
5. 27/16
6. >>> **print**(a \* b)
7. 35/64
9. >>> # Getting numerator/denominator
10. >>> c = a \* b
11. >>> c.numerator
12. 35
13. >>> c.denominator
14. 64
16. >>> # Converting to a float
17. >>> float(c)
18. 0.546875
20. >>> # Limiting the denominator of a value
21. >>> **print**(c.limit\_denominator(8))
22. 4/7
24. >>> # Converting a float to a fraction
25. >>> x = 3.75
26. >>> y = Fraction(\*x.as\_integer\_ratio())
27. >>> y
28. Fraction(15, 4)
29. >>>

## 0x09 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：

1. >>> # Python lists
2. >>> x = [1, 2, 3, 4]
3. >>> y = [5, 6, 7, 8]
4. >>> x \* 2
5. [1, 2, 3, 4, 1, 2, 3, 4]
6. >>> x + 10
7. Traceback (most recent call last):
8. File "<stdin>", line 1, **in** <module>
9. TypeError: can only concatenate list (**not** "int") to list
10. >>> x + y
11. [1, 2, 3, 4, 5, 6, 7, 8]
13. >>> # Numpy arrays
14. >>> **import** numpy as np
15. >>> ax = np.array([1, 2, 3, 4])
16. >>> ay = np.array([5, 6, 7, 8])
17. >>> ax \* 2
18. array([2, 4, 6, 8])
19. >>> ax + 10
20. array([11, 12, 13, 14])
21. >>> ax + ay
22. array([ 6, 8, 10, 12])
23. >>> ax \* ay
24. array([ 5, 12, 21, 32])
25. >>>

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：

1. >>> **def** f(x):
2. ... **return** 3\*x\*\*2 - 2\*x + 7
3. ...
4. >>> f(ax)
5. array([ 8, 15, 28, 47])
6. >>>

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：

1. >>> np.sqrt(ax)
2. array([ 1. , 1.41421356, 1.73205081, 2. ])
3. >>> np.cos(ax)
4. array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
5. >>>

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：

1. >>> grid = np.zeros(shape=(10000,10000), dtype=float)
2. >>> grid
3. array([[ 0., 0., 0., ..., 0., 0., 0.],
4. [ 0., 0., 0., ..., 0., 0., 0.],
5. [ 0., 0., 0., ..., 0., 0., 0.],
6. ...,
7. [ 0., 0., 0., ..., 0., 0., 0.],
8. [ 0., 0., 0., ..., 0., 0., 0.],
9. [ 0., 0., 0., ..., 0., 0., 0.]])
10. >>>

所有的普通操作还是会同时作用在所有元素上：

1. >>> grid += 10
2. >>> grid
3. array([[ 10., 10., 10., ..., 10., 10., 10.],
4. [ 10., 10., 10., ..., 10., 10., 10.],
5. [ 10., 10., 10., ..., 10., 10., 10.],
6. ...,
7. [ 10., 10., 10., ..., 10., 10., 10.],
8. [ 10., 10., 10., ..., 10., 10., 10.],
9. [ 10., 10., 10., ..., 10., 10., 10.]])
10. >>> np.sin(grid)
11. array([[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
12. -0.54402111, -0.54402111],
13. [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
14. -0.54402111, -0.54402111],
15. [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
16. -0.54402111, -0.54402111],
17. ...,
18. [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
19. -0.54402111, -0.54402111],
20. [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
21. -0.54402111, -0.54402111],
22. [-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
23. -0.54402111, -0.54402111]])
24. >>>

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：

1. >>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
2. >>> a
3. array([[ 1, 2, 3, 4],
4. [ 5, 6, 7, 8],
5. [ 9, 10, 11, 12]])
7. >>> # Select row 1
8. >>> a[1]
9. array([5, 6, 7, 8])
11. >>> # Select column 1
12. >>> a[:,1]
13. array([ 2, 6, 10])
15. >>> # Select a subregion and change it
16. >>> a[1:3, 1:3]
17. array([[ 6, 7],
18. [10, 11]])
19. >>> a[1:3, 1:3] += 10
20. >>> a
21. array([[ 1, 2, 3, 4],
22. [ 5, 16, 17, 8],
23. [ 9, 20, 21, 12]])
25. >>> # Broadcast a row vector across an operation on all rows
26. >>> a + [100, 101, 102, 103]
27. array([[101, 103, 105, 107],
28. [105, 117, 119, 111],
29. [109, 121, 123, 115]])
30. >>> a
31. array([[ 1, 2, 3, 4],
32. [ 5, 16, 17, 8],
33. [ 9, 20, 21, 12]])
35. >>> # Conditional assignment on an array
36. >>> np.where(a < 10, a, 10)
37. array([[ 1, 2, 3, 4],
38. [ 5, 10, 10, 8],
39. [ 9, 10, 10, 10]])
40. >>>

## 0x0A 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：

1. >>> **import** numpy as np
2. >>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
3. >>> m
4. matrix([[ 1, -2, 3],
5. [ 0, 4, 5],
6. [ 7, 8, -9]])
8. >>> # Return transpose
9. >>> m.T
10. matrix([[ 1, 0, 7],
11. [-2, 4, 8],
12. [ 3, 5, -9]])
14. >>> # Return inverse
15. >>> m.I
16. matrix([[ 0.33043478, -0.02608696, 0.09565217],
17. [-0.15217391, 0.13043478, 0.02173913],
18. [ 0.12173913, 0.09565217, -0.0173913 ]])
20. >>> # Create a vector and multiply
21. >>> v = np.matrix([[2],[3],[4]])
22. >>> v
23. matrix([[2],
24. [3],
25. [4]])
26. >>> m \* v
27. matrix([[ 8],
28. [32],
29. [ 2]])
30. >>>

可以在 numpy.linalg 子包中找到更多的操作函数，比如：

1. >>> **import** numpy.linalg
3. >>> # Determinant
4. >>> numpy.linalg.det(m)
5. -229.99999999999983
7. >>> # Eigenvalues
8. >>> numpy.linalg.eigvals(m)
9. array([-13.11474312, 2.75956154, 6.35518158])
11. >>> # Solve for x in mx = v
12. >>> x = numpy.linalg.solve(m, v)
13. >>> x
14. matrix([[ 0.96521739],
15. [ 0.17391304],
16. [ 0.46086957]])
17. >>> m \* x
18. matrix([[ 2.],
19. [ 3.],
20. [ 4.]])
21. >>> v
22. matrix([[2],
23. [3],
24. [4]])
25. >>>

## 0x0B 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() ：

1. >>> **import** random
2. >>> values = [1, 2, 3, 4, 5, 6]
3. >>> random.choice(values)
4. 2
5. >>> random.choice(values)
6. 3
7. >>> random.choice(values)
8. 1
9. >>> random.choice(values)
10. 4
11. >>> random.choice(values)
12. 6
13. >>>

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() ：

1. >>> random.sample(values, 2)
2. [6, 2]
3. >>> random.sample(values, 2)
4. [4, 3]
5. >>> random.sample(values, 3)
6. [4, 3, 1]
7. >>> random.sample(values, 3)
8. [5, 4, 1]
9. >>>

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

1. >>> random.shuffle(values)
2. >>> values
3. [2, 4, 6, 5, 3, 1]
4. >>> random.shuffle(values)
5. >>> values
6. [3, 5, 2, 1, 6, 4]
7. >>>

生成随机整数，请使用 random.randint() ：

1. >>> random.randint(0,10)
2. 2
3. >>> random.randint(0,10)
4. 5
5. >>> random.randint(0,10)
6. 0
7. >>> random.randint(0,10)
8. 7
9. >>> random.randint(0,10)
10. 10
11. >>> random.randint(0,10)
12. 3
13. >>>

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

1. >>> random.random()
2. 0.9406677561675867
3. >>> random.random()
4. 0.133129581343897
5. >>> random.random()
6. 0.4144991136919316
7. >>>

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

1. >>> random.getrandbits(200)
2. 335837000776573622800628485064121869519521710558559406913275
3. >>>

## 0x0C 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：

1. >>> **from** datetime **import** timedelta
2. >>> a = timedelta(days=2, hours=6)
3. >>> b = timedelta(hours=4.5)
4. >>> c = a + b
5. >>> c.days
6. 2
7. >>> c.seconds
8. 37800
9. >>> c.seconds / 3600
10. 10.5
11. >>> c.total\_seconds() / 3600
12. 58.5
13. >>>

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：

1. >>> **from** datetime **import** datetime
2. >>> a = datetime(2012, 9, 23)
3. >>> **print**(a + timedelta(days=10))
4. 2012-10-03 00:00:00
5. >>>
6. >>> b = datetime(2012, 12, 21)
7. >>> d = b - a
8. >>> d.days
9. 89
10. >>> now = datetime.today()
11. >>> **print**(now)
12. 2012-12-21 14:54:43.094063
13. >>> **print**(now + timedelta(minutes=10))
14. 2012-12-21 15:04:43.094063
15. >>>

在计算的时候，需要注意的是 datetime 会自动处理闰年。比如：

1. >>> a = datetime(2012, 3, 1)
2. >>> b = datetime(2012, 2, 28)
3. >>> a - b
4. datetime.timedelta(2)
5. >>> (a - b).days
6. 2
7. >>> c = datetime(2013, 3, 1)
8. >>> d = datetime(2013, 2, 28)
9. >>> (c - d).days
10. 1
11. >>>

## 0x0D 计算最后一个周五的日期

#### 问题

你需要查找星期中某一天最后出现的日期，比如星期五。

#### 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：

1. #!/usr/bin/env python
2. # -\*- encoding: utf-8 -\*-
3. """
4. Topic: 最后的周五
5. Desc :
6. """
7. **from** datetime **import** datetime, timedelta
9. weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
10. 'Friday', 'Saturday', 'Sunday']

13. **def** get\_previous\_byday(dayname, start\_date=None):
14. **if** start\_date **is** None:
15. start\_date = datetime.today()
16. day\_num = start\_date.weekday()
17. day\_num\_target = weekdays.index(dayname)
18. days\_ago = (7 + day\_num - day\_num\_target) % 7
19. **if** days\_ago == 0:
20. days\_ago = 7
21. target\_date = start\_date - timedelta(days=days\_ago)
22. **return** target\_date

在交互式解释器中使用如下：

1. >>> datetime.today() # For reference
2. datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
3. >>> get\_previous\_byday('Monday')
4. datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
5. >>> get\_previous\_byday('Tuesday') # Previous week, not today
6. datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
7. >>> get\_previous\_byday('Friday')
8. datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
9. >>>

可选的 start\_date 参数可以由另外一个 datetime 实例来提供。比如：

1. >>> get\_previous\_byday('Sunday', datetime(2012, 12, 21))
2. datetime.datetime(2012, 12, 16, 0, 0)
3. >>>

## 0x0E 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。

1. **from** datetime **import** datetime, date, timedelta
2. **import** calendar
4. **def** get\_month\_range(start\_date=None):
5. **if** start\_date **is** None:
6. start\_date = date.today().replace(day=1)
7. \_, days\_in\_month = calendar.monthrange(start\_date.year, start\_date.month)
8. end\_date = start\_date + timedelta(days=days\_in\_month)
9. **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：

1. >>> a\_day = timedelta(days=1)
2. >>> first\_day, last\_day = get\_month\_range()
3. >>> **while** first\_day < last\_day:
4. ...     **print**(first\_day)
5. ...     first\_day += a\_day
6. ...
7. 2012-08-01
8. 2012-08-02
9. 2012-08-03
10. 2012-08-04
11. 2012-08-05
12. 2012-08-06
13. 2012-08-07
14. 2012-08-08
15. 2012-08-09
16. #... and so on...

## 0x0F 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：

1. >>> **from** datetime **import** datetime
2. >>> text = '2012-09-20'
3. >>> y = datetime.strptime(text, '%Y-%m-%d')
4. >>> z = datetime.now()
5. >>> diff = z - y
6. >>> diff
7. datetime.timedelta(3, 77824, 177393)
8. >>>

## 0x10 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：

1. >>> **from** datetime **import** datetime
2. >>> **from** pytz **import** timezone
3. >>> d = datetime(2012, 12, 21, 9, 30, 0)
4. >>> **print**(d)
5. 2012-12-21 09:30:00
6. >>>
8. >>> # Localize the date for Chicago
9. >>> central = timezone('US/Central')
10. >>> loc\_d = central.localize(d)
11. >>> **print**(loc\_d)
12. 2012-12-21 09:30:00-06:00
13. >>>

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：

1. >>> # Convert to Bangalore time
2. >>> bang\_d = loc\_d.astimezone(timezone('Asia/Kolkata'))
3. >>> **print**(bang\_d)
4. 2012-12-21 21:00:00+05:30
5. >>>

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：

1. >>> d = datetime(2013, 3, 10, 1, 45)
2. >>> loc\_d = central.localize(d)
3. >>> **print**(loc\_d)
4. 2013-03-10 01:45:00-06:00
5. >>> later = loc\_d + timedelta(minutes=30)
6. >>> **print**(later)
7. 2013-03-10 02:15:00-06:00 # WRONG! WRONG!
8. >>>

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：

1. >>> **from** datetime **import** timedelta
2. >>> later = central.normalize(loc\_d + timedelta(minutes=30))
3. >>> **print**(later)
4. 2013-03-10 03:15:00-05:00
5. >>>