2.1 使用多个界定符分割字符串¶

问题¶

你需要将一个字符串分割为多个字段，但是分隔符(还有周围的空格)并不是固定的。

解决方案¶

`string` 对象的 `split()` 方法只适应于非常简单的字符串分割情形， 它并不允许有多个分隔符或者是分隔符周围不确定的空格。 当你需要更加灵活的切割字符串的时候，最好使用 `re.split()` 方法：

```
>>> line = 'asdf fjdk; afed, fjek,asdf, foo'
>>> import re
>>> re.split(r'[;,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

讨论¶

函数 `re.split()` 是非常实用的，因为它允许你为分隔符指定多个正则模式。 比如，在上面的例子中，分隔符可以是逗号，分号或者是空格，并且后面紧跟着任意个的空格。 只要这个模式被找到，那么匹配的分隔符两边的实体都会被当成是结果中的元素返回。 返回结果为一个字段列表，这个跟 `str.split()` 返回值类型是一样的。

当你使用 `re.split()` 函数时候，需要特别注意的是正则表达式中是否包含一个括号捕获分组。 如果使用了捕获分组，那么被匹配的文本也将出现在结果列表中。比如，观察一下这段代码运行后的结果：

```
>>> fields = re.split(r'(;|,|\s)\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ',', 'fjek', ',', 'asdf', ',', 'foo']
>>>
```

获取分割字符在某些情况下也是有用的。 比如，你可能想保留分割字符串，用来在后面重新构造一个新的输出字符串：

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ',', ',', ',', '']
>>> # Reform the line using the same delimiters
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

如果你不想保留分割字符串到结果列表中去，但仍然需要使用到括号来分组正则表达式的话， 确保你的分组是非捕获分组，形如 `(?:...)` 。比如：

```
>>> re.split(r'(?:,|;|\s)\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

2.10 在正则式中使用Unicode¶

问题¶

你正在使用正则表达式处理文本，但是关注的是Unicode字符处理。

解决方案¶

默认情况下 `re` 模块已经对一些Unicode字符类有了基本的支持。 比如， `\\d` 已经匹配任意的unicode数字字符了：

```
>>> import re
>>> num = re.compile('\d+')
>>> # ASCII digits
>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>
>>> # Arabic digits
>>> num.match('\u0661\u0662\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

如果你想在模式中包含指定的Unicode字符，你可以使用Unicode字符对应的转义序列(比如 `\uFFF` 或者 `\UFFFFFFF`)。 比如，下面是一个匹配几个不同阿拉伯编码页面中所有字符的正则表达式：

```
>>> arabic = re.compile('[\u0600-\u06ff\u0750-\u077f\u08a0-\u08ff]+')
>>>
```

当执行匹配和搜索操作的时候，最好是先标准化并且清理所有文本为标准化格式(参考2.9小节)。 但是同样也应该注意一些特殊情况，比如在忽略大小写匹配和大小写转换时的行为。

```
>>> pat = re.compile('stra\u00dfe', re.IGNORECASE)
>>> s = 'straße'
>>> pat.match(s) # Matches
<_sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper()) # Doesn't match
>>> s.upper() # Case folds
'STRASSE'
>>>
```

讨论¶

混合使用Unicode和正则表达式通常会让你抓狂。 如果你真的打算这样做的话，最好考虑下安装第三方正则式库， 它们会为Unicode的大小写转换和其他大量有趣特性提供全面的支持，包括模糊匹配。

2.11 删除字符串中不需要的字符¶

问题¶

你想去掉文本字符串开头，结尾或者中间不想要的字符，比如空白。

解决方案¶

`strip()` 方法能用于删除开始或结尾的字符。 `lstrip()` 和 `rstrip()` 分别从左和从右执行删除操作。 默认情况下，这些方法会去除空白字符，但是你也可以指定其他字符。比如：

```
>>> # Whitespace stripping
>>> s = ' hello world \n'
>>> s.strip()
'hello world'
>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
' hello world'
>>>
>>> # Character stripping
>>> t = '-----hello====='
>>> t.lstrip('-')
'hello====='
>>> t.strip('-=')
'hello'
>>>
```

讨论¶

这些 `strip()` 方法在读取和清理数据以备后续处理的时候是经常会被用到的。 比如，你可以用它们来去掉空格，引号和完成其他任务。

但是需要注意的是去除操作不会对字符串的中间的文本产生任何影响。比如：

```
>>> s = ' hello     world \n'
>>> s = s.strip()
>>> s
'hello     world'
>>>
```

如果你想处理中间的空格，那么你需要求助其他技术。比如使用 `replace()` 方法或者是用正则表达式替换。示例如下：

```
>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>
```

通常情况下你想将字符串 `strip` 操作和其他迭代操作相结合，比如从文件中读取多行数据。 如果是这样的话，那么生成器表达式就可以大显身手了。比如：

```
with open(filename) as f:
    lines = (line.strip() for line in f)
    for line in lines:
        print(line)
```

在这里，表达式 `lines = (line.strip() for line in f)` 执行数据转换操作。 这种方式非常高效，因为它不需要预先读取所有数据放到一个临时的列表中去。 它仅仅只是创建一个生成器，并且每次返回行之前会先执行 `strip` 操作。

对于更高阶的strip，你可能需要使用 `translate()` 方法。请参阅下一节了解更多关于字符串清理的内容。

2.12 审查清理文本字符串¶

问题¶

一些无聊的幼稚黑客在你的网站页面表单中输入文本”pýt̂hön̈”，然后你想将这些字符清理掉。

解决方案¶

文本清理问题会涉及到包括文本解析与数据处理等一系列问题。 在非常简单的情形下，你可能会选择使用字符串函数(比如 `str.upper()` 和 `str.lower()`)将文本转为标准格式。 使用 `str.replace()` 或者 `re.sub()` 的简单替换操作能删除或者改变指定的字符序列。 你同样还可以使用2.9小节的 `unicodedata.normalize()` 函数将unicode文本标准化。

然后，有时候你可能还想在清理操作上更进一步。比如，你可能想消除整个区间上的字符或者去除变音符。 为了这样做，你可以使用经常会被忽视的 `str.translate()` 方法。 为了演示，假设你现在有下面这个凌乱的字符串：

```
>>> s = 'pýt̂höñ\fis\tawesome\r\n'
>>> s
'pýt̂höñ\x0cis\tawesome\r\n'
>>>
```

第一步是清理空白字符。为了这样做，先创建一个小的转换表格然后使用 `translate()` 方法：

```
>>> remap = {
...     ord('\t') : ' ',
...     ord('\f') : ' ',
...     ord('\r') : None # Deleted
... }
>>> a = s.translate(remap)
>>> a
'pýt̂höñ is awesome\n'
>>>
```

正如你看的那样，空白字符 `\t` 和 `\f` 已经被重新映射到一个空格。回车字符r直接被删除。

你可以以这个表格为基础进一步构建更大的表格。比如，让我们删除所有的和音符：

```
>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...                          if unicodedata.combining(chr(c)))
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pýt̂höñ is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>
```

上面例子中，通过使用 `dict.fromkeys()` 方法构造一个字典，每个Unicode和音符作为键，对应的值全部为 `None` 。

然后使用 `unicodedata.normalize()` 将原始输入标准化为分解形式字符。 然后再调用 `translate` 函数删除所有重音符。 同样的技术也可以被用来删除其他类型的字符(比如控制字符等)。

作为另一个例子，这里构造一个将所有Unicode数字字符映射到对应的ASCII字符上的表格：

```
>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...          for c in range(sys.maxunicode)
...          if unicodedata.category(chr(c)) == 'Nd' }
...
>>> len(digitmap)
460
>>> # Arabic digits
>>> x = '\u0661\u0662\u0663'
>>> x.translate(digitmap)
```

```
'123'
>>>
```

另一种清理文本的技术涉及到I/O解码与编码函数。这里的思路是先对文本做一些初步的清理， 然后再结合 `encode()` 或者 `decode()` 操作来清除或修改它。比如：

```
>>> a
'pýt̂öñ is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>
```

这里的标准化操作将原来的文本分解为单独的和音符。接下来的ASCII编码/解码只是简单的一下子丢弃掉那些字符。当然，这种方法仅仅只在最后的目标就是获取到文本对应ACSII表示的时候生效。

讨论¶

文本字符清理一个最主要的问题应该是运行的性能。一般来讲，代码越简单运行越快。 对于简单的替换操作， `str.replace()` 方法通常是最快的，甚至在你需要多次调用的时候。 比如，为了清理空白字符，你可以这样做：

```
def clean_spaces(s):
    s = s.replace('\r', '')
    s = s.replace('\t', ' ')
    s = s.replace('\f', ' ')
    return s
```

如果你去测试的话，你就会发现这种方式会比使用 `translate()` 或者正则表达式要快很多。

另一方面，如果你需要执行任何复杂字符对字符的重新映射或者删除操作的话， `translate()` 方法会非常的快。

从大的方面来讲，对于你的应用程序来说性能是你不得不去自己研究的东西。 不幸的是，我们不可能给你建议一个特定的技术，使它能够适应所有的情况。 因此实际情况中需要你自己去尝试不同的方法并评估它。

尽管这一节集中讨论的是文本，但是类似的技术也可以适用于字节，包括简单的替换，转换和正则表达式。

2.13 字符串对齐¶

问题¶

你想通过某种对齐方式来格式化字符串

解决方案¶

对于基本的字符串对齐操作，可以使用字符串的 `ljust()` , `rjust()` 和 `center()` 方法。比如：

```
>>> text = 'Hello World'
>>> text.ljust(20)
'Hello World         '
>>> text.rjust(20)
'         Hello World'
>>> text.center(20)
'    Hello World     '
>>>
```

所有这些方法都能接受一个可选的填充字符。比如：

```
>>> text.rjust(20,'=')
'==========Hello World'
>>> text.center(20,'*')
'****Hello World*****'
>>>
```

函数 `format()` 同样可以用来很容易的对齐字符串。 你要做的就是使用 `<,>` 或者 `^` 字符后面紧跟一个指定的宽度。比如：

```
>>> format(text, '>20')
'         Hello World'
>>> format(text, '<20')
'Hello World         '
>>> format(text, '^20')
'    Hello World     '
>>>
```

如果你想指定一个非空格的填充字符，将它写到对齐字符的前面即可：

```
>>> format(text, '=>20s')
'==========Hello World'
>>> format(text, '*^20s')
'****Hello World*****'
>>>
```

当格式化多个值的时候，这些格式代码也可以被用在 `format()` 方法中。比如：

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'     Hello      World'
>>>
```

`format()` 函数的一个好处是它不仅适用于字符串。它可以用来格式化任何值，使得它非常的通用。 比如，你可以用它来格式化数字：

```
>>> x = 1.2345
>>> format(x, '>10')
'    1.2345'
>>> format(x, '^10.2f')
'   1.23   '
>>>
```

讨论¶

在老的代码中，你经常会看到被用来格式化文本的 `%` 操作符。比如：

```
>>> '%-20s' % text
'Hello World         '
>>> '%20s' % text
'         Hello World'
>>>
```

但是，在新版本代码中，你应该优先选择 `format()` 函数或者方法。 `format()` 要比 `%` 操作符的功能更为强大。 并且 `format()` 也比使用 `ljust()` , `rjust()` 或 `center()` 方法更通用， 因为它可以用来格式化任意对象，而不仅仅是字符串。

如果想要完全了解 `format()` 函数的有用特性， 请参考 在线Python文档

2.14 合并拼接字符串¶

问题¶

你想将几个小的字符串合并为一个大的字符串

解决方案¶

如果你想要合并的字符串是在一个序列或者 iterable 中，那么最快的方式就是使用 join() 方法。比如：

```
>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ' '.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>
```

初看起来，这种语法看上去会比较怪，但是 join() 被指定为字符串的一个方法。 这样做的部分原因是你想去连接的对象可能来自各种不同的数据序列(比如列表，元组，字典，文件，集合或生成器等)， 如果在所有这些对象上都定义一个 join() 方法明显是冗余的。 因此你只需要指定你想要的分割字符串并调用他的 join() 方法去将文本片段组合起来。

如果你仅仅只是合并少数几个字符串，使用加号(+)通常已经足够了：

```
>>> a = 'Is Chicago'
>>> b = 'Not Chicago?'
>>> a + ' ' + b
'Is Chicago Not Chicago?'
>>>
```

加号(+)操作符在作为一些复杂字符串格式化的替代方案的时候通常也工作的很好，比如：

```
>>> print('{} {}'.format(a,b))
Is Chicago Not Chicago?
>>> print(a + ' ' + b)
Is Chicago Not Chicago?
>>>
```

如果你想在源码中将两个字面字符串合并起来，你只需要简单的将它们放到一起，不需要用加号(+)。比如：

```
>>> a = 'Hello' 'World'
>>> a
'HelloWorld'
>>>
```

讨论¶

字符串合并可能看上去并不需要用一整节来讨论。 但是不应该小看这个问题，程序员通常在字符串格式化的时候因为选择不当而给应用程序带来严重性能损失。

最重要的需要引起注意的是，当我们使用加号(+)操作符去连接大量的字符串的时候是非常低效率的， 因为加号连接会引起内存复制以及垃圾回收操作。 特别的，你永远都不应像下面这样写字符串连接代码：

```
s = ''
for p in parts:
    s += p
```

这种写法会比使用 join() 方法运行的要慢一些，因为每一次执行+=操作的时候会创建一个新的字符串对象。 你最好是先收集所有的字符串片段然后再将它们连接起来。

一个相对比较聪明的技巧是利用生成器表达式(参考1.19小节)转换数据为字符串的同时合并字符串，比如：

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for d in data)
'ACME,50,91.1'
>>>
```

同样还得注意不必要的字符串连接操作。有时候程序员在没有必要做连接操作的时候仍然多此一举。比如在打印的时候：

```
print(a + ':' + b + ':' + c) # Ugly
print(':'.join([a, b, c])) # Still ugly
print(a, b, c, sep=':') # Better
```

当混合使用I/O操作和字符串连接操作的时候，有时候需要仔细研究你的程序。 比如，考虑下面的两端代码片段：

```
# Version 1 (string concatenation)
f.write(chunk1 + chunk2)

# Version 2 (separate I/O operations)
f.write(chunk1)
f.write(chunk2)
```

如果两个字符串很小，那么第一个版本性能会更好些，因为I/O系统调用天生就慢。 另外一方面，如果两个字符串很大，那么第二个版本可能会更加高效， 因为它避免了创建一个很大的临时结果并且要复制大量的内存块数据。 还是那句话，有时候是需要根据你的应用程序特点来决定应该使用哪种方案。

最后谈一下，如果你准备编写构建大量小字符串的输出代码， 你最好考虑下使用生成器函数，利用yield语句产生输出片段。比如：

```
def sample():
    yield 'Is'
    yield 'Chicago'
    yield 'Not'
    yield 'Chicago?'
```

这种方法一个有趣的方面是它并没有对输出片段到底要怎样组织做出假设。 例如，你可以简单的使用 `join()` 方法将这些片段合并起来：

```
text = ''.join(sample())
```

或者你也可以将字符串片段重定向到I/O：

```
for part in sample():
    f.write(part)
```

再或者你还可以写出一些结合I/O操作的混合方案：

```
def combine(source, maxsize):
    parts = []
    size = 0
    for part in source:
        parts.append(part)
        size += len(part)
        if size > maxsize:
            yield ''.join(parts)
            parts = []
            size = 0
    yield ''.join(parts)

# 结合文件操作
with open('filename', 'w') as f:
    for part in combine(sample(), 32768):
        f.write(part)
```

这里的关键点在于原始的生成器函数并不需要知道使用细节，它只负责生成字符串片段就行了。

2.15 字符串中插入变量¶

问题¶

你想创建一个内嵌变量的字符串，变量被它的值所表示的字符串替换掉。

解决方案¶

Python并没有对在字符串中简单替换变量值提供直接的支持。 但是通过使用字符串的 `format()` 方法来解决这个问题。比如：

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

或者，如果要被替换的变量能在变量域中找到， 那么你可以结合使用 `format_map()` 和 `vars()` 。就像下面这样：

```
>>> name = 'Guido'
>>> n = 37
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

`vars()` 还有一个有意思的特性就是它也适用于对象实例。比如：

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Guido',37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

`format` 和 `format_map()` 的一个缺陷就是它们并不能很好的处理变量缺失的情况，比如：

```
>>> s.format(name='Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

一种避免这种错误的方法是另外定义一个含有 `__missing__()` 方法的字典对象，就像下面这样：

```
class safesub(dict):
"""防止key找不到"""
def __missing__(self, key):
    return '{' + key + '}'
```

现在你可以利用这个类包装输入后传递给 `format_map()` ：

```
>>> del n # Make sure n is undefined
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

如果你发现自己在代码中频繁的执行这些步骤，你可以将变量替换步骤用一个工具函数封装起来。就像下面这样：

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

现在你可以像下面这样写了：

```
>>> name = 'Guido'
>>> n = 37
>>> print(sub('Hello {name}'))
Hello Guido
>>> print(sub('You have {n} messages.'))
You have 37 messages.
>>> print(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>
```

讨论¶

多年以来由于Python缺乏对变量替换的内置支持而导致了各种不同的解决方案。 作为本节中展示的一个可能的解决方案，你可以有时候会看到像下面这样的字符串格式化代码：

```
>>> name = 'Guido'
>>> n = 37
>>> '%(name) has %(n) messages.' % vars()
'Guido has 37 messages.'
>>>
```

你可能还会看到字符串模板的使用：

```
>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>
```

然而， `format()` 和 `format_map()` 相比较上面这些方案而已更加先进，因此应该被优先选择。 使用 `format()` 方法还有一个好处就是你可以获得对字符串格式化的所有支持(对齐，填充，数字格式化等待)， 而这些特性是使用像模板字符串之类的方案不可能获得的。

本机还部分介绍了一些高级特性。映射或者字典类中鲜为人知的 `__missing__()` 方法可以让你定义如何处理缺失的值。 在 `SafeSub` 类中，这个方法被定义为对缺失的值返回一个占位符。 你可以发现缺失的值会出现在结果字符串中(在调试的时候可能很有用)，而不是产生一个 `KeyError` 异常。

`sub()` 函数使用 `sys._getframe(1)` 返回调用者的栈帧。可以从中访问属性 `f_locals` 来获得局部变量。 毫无疑问绝大部分情况下在代码中去直接操作栈帧应该是不推荐的。 但是，对于像字符串替换工具函数而言它是非常有用的。 另外，值得注意的是 `f_locals` 是一个复制调用函数的本地变量的字典。 尽管你可以改变 `f_locals` 的内容，但是这个修改对于后面的变量访问没有任何影响。 所以，虽说访问一个栈帧看上去很邪恶，但是对它的任何操作不会覆盖和改变调用者本地变量的值。

2.16 以指定列宽格式化字符串¶

问题¶

你有一些长字符串，想以指定的列宽将它们重新格式化。

解决方案¶

使用 `textwrap` 模块来格式化字符串的输出。比如，假如你有下列的长字符串：

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \
the eyes, not around the eyes, don't look around the eyes, \
look into my eyes, you're under."
```

下面演示使用 `textwrap` 格式化字符串的多种方式：

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes, don't look around the eyes, look into my eyes,
you're under.

>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, initial_indent='    '))
    Look into my eyes, look into my
eyes, the eyes, the eyes, the eyes, not
around the eyes, don't look around the
eyes, look into my eyes, you're under.

>>> print(textwrap.fill(s, 40, subsequent_indent='    '))
Look into my eyes, look into my eyes,
    the eyes, the eyes, the eyes, not
    around the eyes, don't look around
    the eyes, look into my eyes, you're
    under.
```

讨论¶

`textwrap` 模块对于字符串打印是非常有用的，特别是当你希望输出自动匹配终端大小的时候。 你可以使用 `os.get_terminal_size()` 方法来获取终端的大小尺寸。比如：

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

`fill()` 方法接受一些其他可选参数来控制tab，语句结尾等。 参阅 textwrap.TextWrapper文档 获取更多内容。

2.17 在字符串中处理html和xml¶

问题¶

你想将HTML或者XML实体如 `&entity;` 或 `&#code;` 替换为对应的文本。 再者，你需要转换文本中特定的字符(比如<, >, 或 &)。

解决方案¶

如果你想替换文本字符串中的 '<' 或者 '>'，使用 `html.escape()` 函数可以很容易的完成。比如：

```
>>> s = 'Elements are written as "<tag>text</tag>".'
>>> import html
>>> print(s)
Elements are written as "<tag>text</tag>".
>>> print(html.escape(s))
Elements are written as &quot;&lt;tag&gt;text&lt;/tag&gt;&quot;.

>>> # Disable escaping of quotes
>>> print(html.escape(s, quote=False))
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;".
>>>
```

如果你正在处理的是ASCII文本，并且想将非ASCII文本对应的编码实体嵌入进去， 可以给某些I/O函数传递参数 `errors='xmlcharrefreplace'` 来达到这个目。比如：

```
>>> s = 'Spicy Jalapeño'
>>> s.encode('ascii', errors='xmlcharrefreplace')
b'Spicy Jalape&#241;o'
>>>
```

为了替换文本中的编码实体，你需要使用另外一种方法。 如果你正在处理HTML或者XML文本，试着先使用一个合适的HTML或者XML解析器。 通常情况下，这些工具会自动替换这些编码值，你无需担心。

有时候，如果你接收到了一些含有编码值的原始文本，需要手动去做替换， 通常你只需要使用HTML或者XML解析器的一些相关工具函数/方法即可。比如：

```
>>> s = 'Spicy &quot;Jalape&#241;o&quot.'
>>> from html.parser import HTMLParser
>>> p = HTMLParser()
>>> p.unescape(s)
'Spicy "Jalapeño".'
>>>
>>> t = 'The prompt is &gt;&gt;&gt;'
>>> from xml.sax.saxutils import unescape
>>> unescape(t)
'The prompt is >>>'
>>>
```

讨论¶

在生成HTML或者XML文本的时候，如果正确的转换特殊标记字符是一个很容易被忽视的细节。 特别是当你使用 `print()` 函数或者其他字符串格式化来产生输出的时候。 使用像 `html.escape()` 的工具函数可以很容易的解决这类问题。

如果你想以其他方式处理文本，还有一些其他的工具函数比如 `xml.sax.saxutils.unescapge()` 可以帮助你。 然而，你应该先调研清楚怎样使用一个合适的解析器。 比如，如果你在处理HTML或XML文本， 使用某个解析模块比如 `html.parse` 或 `xml.etree.ElementTree` 已经帮你自动处理了相关的替换细节。

2.18 字符串令牌解析¶

问题¶

你有一个字符串，想从左至右将其解析为一个令牌流。

解决方案¶

假如你有下面这样一个文本字符串：

```
text = 'foo = 23 + 42 * 10'
```

为了令牌化字符串，你不仅需要匹配模式，还得指定模式的类型。 比如，你可能想将字符串像下面这样转换为序列对：

```
tokens = [('NAME', 'foo'), ('EQ','='), ('NUM', '23'), ('PLUS','+'),
          ('NUM', '42'), ('TIMES', '*'), ('NUM', '10')]
```

为了执行这样的切分，第一步就是像下面这样利用命名捕获组的正则表达式来定义所有可能的令牌，包括空格：

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ = r'(?P<EQ>=)'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

在上面的模式中， `?P<TOKENNAME>` 用于给一个模式命名，供后面使用。

下一步，为了令牌化，使用模式对象很少被人知道的 `scanner()` 方法。 这个方法会创建一个 `scanner` 对象， 在这个对象上不断的调用 `match()` 方法会一步步的扫描目标文本，每步一个匹配。 下面是演示一个 `scanner` 对象如何工作的交互式例子：

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

实际使用这种技术的时候，可以很容易的像下面这样将上述代码打包到一个生成器中：

```
def generate_tokens(pat, text):
    Token = namedtuple('Token', ['type', 'value'])
    scanner = pat.scanner(text)
```

```
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use
for tok in generate_tokens(master_pat, 'foo = 42'):
    print(tok)
# Produces output
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

如果你想过滤令牌流，你可以定义更多的生成器函数或者使用一个生成器表达式。 比如，下面演示怎样过滤所有的空白令牌：

```
tokens = (tok for tok in generate_tokens(master_pat, text)
          if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

讨论¶

通常来讲令牌化是很多高级文本解析与处理的第一步。 为了使用上面的扫描方法，你需要记住这里一些重要的几点。第一点就是你必须确认你使用正则表达式指定了所有输入中可能出现的文本序列。 如果有任何不可匹配的文本出现了，扫描就会直接停止。这也是为什么上面例子中必须指定空白字符令牌的原因。

令牌的顺序也是有影响的。 `re` 模块会按照指定好的顺序去做匹配。 因此，如果一个模式恰好是另一个更长模式的子字符串，那么你需要确定长模式写在前面。比如：

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Correct
# master_pat = re.compile('|'.join([LT, LE, EQ])) # Incorrect
```

第二个模式是错的，因为它会将文本<=匹配为令牌LT紧跟着EQ，而不是单独的令牌LE，这个并不是我们想要的结果。

最后，你需要留意下子字符串形式的模式。比如，假设你有如下两个模式：

```
PRINT = r'(?P<PRINT>print)'
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'

master_pat = re.compile('|'.join([PRINT, NAME]))

for tok in generate_tokens(master_pat, 'printer'):
    print(tok)

# Outputs :
# Token(type='PRINT', value='print')
# Token(type='NAME', value='er')
```

关于更高阶的令牌化技术，你可能需要查看 PyParsing 或者 PLY 包。 一个调用PLY的例子在下一节会有演示。

2.19 实现一个简单的递归下降分析器¶

问题¶

你想根据一组语法规则解析文本并执行命令，或者构造一个代表输入的抽象语法树。 如果语法非常简单，你可以不去使用一些框架，而是自己写这个解析器。

解决方案¶

在这个问题中，我们集中讨论根据特殊语法去解析文本的问题。 为了这样做，你首先要以BNF或者EBNF形式指定一个标准语法。 比如，一个简单数学表达式语法可能像下面这样：

```
expr ::= expr + term
    |   expr - term
    |   term

term ::= term * factor
    |   term / factor
    |   factor

factor ::= ( expr )
    |   NUM
```

或者，以EBNF形式：

```
expr ::= term { (+|-) term }*

term ::= factor { (*|/) factor }*

factor ::= ( expr )
    |   NUM
```

在EBNF中，被包含在 `{...}*` 中的规则是可选的。*代表0次或多次重复(跟正则表达式中意义是一样的)。

现在，如果你对BNF的工作机制还不是很明白的话，就把它当做是一组左右符号可相互替换的规则。 一般来讲，解析的原理就是你利用BNF完成多个替换和扩展以匹配输入文本和语法规则。 为了演示，假设你正在解析形如 `3 + 4 * 5` 的表达式。 这个表达式先要通过使用2.18节中介绍的技术分解为一组令牌流。 结果可能是像下列这样的令牌序列：

```
NUM + NUM * NUM
```

在此基础上， 解析动作会试着去通过替换操作匹配语法到输入令牌：

```
expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor}* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM
```

下面所有的解析步骤可能需要花点时间弄明白，但是它们原理都是查找输入并试着去匹配语法规则。 第一个输入令牌是NUM，因此替换首先会匹配那个部分。 一旦匹配成功，就会进入下一个令牌+，以此类推。 当已经确定不能匹配下一个令牌的时候，右边的部分(比如 `{ (*/) factor }*`)就会被清理掉。 在一个成功的解析中，整个右边部分会完全展开来匹配输入令牌流。

有了前面的知识背景，下面我们举一个简单示例来展示如何构建一个递归下降表达式求值程序：

```
#!/usr/bin/env python
```

```
# -*- encoding: utf-8 -*-
"""
Topic: 下降解析器
Desc :
"""
import re
import collections

# Token specification
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
MINUS = r'(?P<MINUS>-)'
TIMES = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\()'
RPAREN = r'(?P<RPAREN>\))'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                  DIVIDE, LPAREN, RPAREN, WS]))
# Tokenizer
Token = collections.namedtuple('Token', ['type', 'value'])


def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok


# Parser
class ExpressionEvaluator:
    '''
    Implementation of a recursive descent parser. Each method
    implements a single grammar rule. Use the ._accept() method
    to test and accept the current lookahead token. Use the ._expect()
    method to exactly match and discard the next token on on the input
    (or raise a SyntaxError if it doesn't match).
    '''

    def parse(self, text):
        self.tokens = generate_tokens(text)
        self.tok = None  # Last symbol consumed
        self.nexttok = None  # Next symbol tokenized
        self._advance()  # Load first lookahead token
        return self.expr()

    def _advance(self):
        'Advance one token ahead'
        self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

    def _accept(self, toktype):
        'Test and consume the next token if it matches toktype'
        if self.nexttok and self.nexttok.type == toktype:
            self._advance()
            return True
        else:
            return False

    def _expect(self, toktype):
        'Consume next token if it matches toktype or raise SyntaxError'
        if not self._accept(toktype):
            raise SyntaxError('Expected ' + toktype)

    # Grammar rules follow
    def expr(self):
        "expression ::= term { ('+'|'-') term }*"
        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
```

```
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval += right
            elif op == 'MINUS':
                exprval -= right
        return exprval

    def term(self):
        "term ::= factor { ('*'|'/') factor }*"
        termval = self.factor()
        while self._accept('TIMES') or self._accept('DIVIDE'):
            op = self.tok.type
            right = self.factor()
            if op == 'TIMES':
                termval *= right
            elif op == 'DIVIDE':
                termval /= right
        return termval

    def factor(self):
        "factor ::= NUM | ( expr )"
        if self._accept('NUM'):
            return int(self.tok.value)
        elif self._accept('LPAREN'):
            exprval = self.expr()
            self._expect('RPAREN')
            return exprval
        else:
            raise SyntaxError('Expected NUMBER or LPAREN')


def descent_parser():
    e = ExpressionEvaluator()
    print(e.parse('2'))
    print(e.parse('2 + 3'))
    print(e.parse('2 + 3 * 4'))
    print(e.parse('2 + (3 + 4) * 5'))
    # print(e.parse('2 + (3 + * 4)'))
    # Traceback (most recent call last):
    #    File "<stdin>", line 1, in <module>
    #    File "exprparse.py", line 40, in parse
    #    return self.expr()
    #    File "exprparse.py", line 67, in expr
    #    right = self.term()
    #    File "exprparse.py", line 77, in term
    #    termval = self.factor()
    #    File "exprparse.py", line 93, in factor
    #    exprval = self.expr()
    #    File "exprparse.py", line 67, in expr
    #    right = self.term()
    #    File "exprparse.py", line 77, in term
    #    termval = self.factor()
    #    File "exprparse.py", line 97, in factor
    #    raise SyntaxError("Expected NUMBER or LPAREN")
    #    SyntaxError: Expected NUMBER or LPAREN


if __name__ == '__main__':
    descent_parser()
```

讨论¶

文本解析是一个很大的主题， 一般会占用学生学习编译课程时刚开始的三周时间。 如果你在找寻关于语法，解析算法等相关的背景知识的话，你应该去看一下编译器书籍。 很显然，关于这方面的内容太多，不可能在这里全部展开。

尽管如此，编写一个递归下降解析器的整体思路是比较简单的。 开始的时候，你先获得所有的语法规则，然后将其转换为一个函数或者方法。 因此如果你的语法类似这样：

```
expr ::= term { ('+'|'-') term }*

term ::= factor { ('*'|'/') factor }*

factor ::= '(' expr ')'
    | NUM
```

你应该首先将它们转换成一组像下面这样的方法：

```
class ExpressionEvaluator:
    ...
    def expr(self):
    ...
    def term(self):
    ...
    def factor(self):
    ...
```

每个方法要完成的任务很简单 - 它必须从左至右遍历语法规则的每一部分，处理每个令牌。 从某种意义上讲，方法的目的就是要么处理完语法规则，要么产生一个语法错误。 为了这样做，需采用下面的这些实现方法：

- 如果规则中的下个符号是另外一个语法规则的名字(比如term或factor)，就简单的调用同名的方法即可。 这就是该算法中"下降"的由来 - 控制下降到另一个语法规则中去。 有时候规则会调用已经执行的方法(比如，在 `factor ::= '('expr ')'` 中对expr的调用)。 这就是算法中"递归"的由来。
- 如果规则中下一个符号是个特殊符号(比如()，你得查找下一个令牌并确认是一个精确匹配)。 如果不匹配，就产生一个语法错误。这一节中的 `_expect()` 方法就是用来做这一步的。
- 如果规则中下一个符号为一些可能的选择项(比如 + 或 -)， 你必须对每一种可能情况检查下一个令牌，只有当它匹配一个的时候才能继续。 这也是本节示例中 `_accept()` 方法的目的。 它相当于_expect()方法的弱化版本，因为如果一个匹配找到了它会继续， 但是如果没找到，它不会产生错误而是回滚(允许后续的检查继续进行)。
- 对于有重复部分的规则(比如在规则表达式 `::= term { ('+'|'-') term }*` 中)， 重复动作通过一个while循环来实现。 循环主体会收集或处理所有的重复元素直到没有其他元素可以找到。
- 一旦整个语法规则处理完成，每个方法会返回某种结果给调用者。 这就是在解析过程中值是怎样累加的原理。 比如，在表达式求值程序中，返回值代表表达式解析后的部分结果。 最后所有值会在最顶层的语法规则方法中合并起来。

尽管向你演示的是一个简单的例子，递归下降解析器可以用来实现非常复杂的解析。 比如，Python语言本身就是通过一个递归下降解析器去解释的。 如果你对此感兴趣，你可以通过查看Python源码文件Grammar/Grammar来研究下底层语法机制。 看完你会发现，通过手动方式去实现一个解析器其实会有很多的局限和不足之处。

其中一个局限就是它们不能被用于包含任何左递归的语法规则中。比如，假如你需要翻译下面这样一个规则：

```
items ::= items ',' item
    | item
```

为了这样做，你可能会像下面这样使用 `items()` 方法：

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
        itemsval.append(self.item())
    else:
        itemsval = [ self.item() ]
```

唯一的问题是这个方法根本不能工作，事实上，它会产生一个无限递归错误。

关于语法规则本身你可能也会碰到一些棘手的问题。 比如，你可能想知道下面这个简单扼语法是否表述得当：

```
expr ::= factor { ('+'|'-'|'*'|'/') factor }*

factor ::= '(' expression ')'
    | NUM
```

这个语法看上去没啥问题，但是它却不能察觉到标准四则运算中的运算符优先级。 比如，表达式 `"3 + 4 * 5"` 会得到35而不是期望的23. 分开使用"expr"和"term"规则可以让它正确的工作。

对于复杂的语法，你最好是选择某个解析工具比如PyParsing或者是PLY。 下面是使用PLY来重写表达式求值程序的代码：

```
from ply.lex import lex
from ply.yacc import yacc

# Token list
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN' ]
# Ignored characters
t_ignore = ' \t\n'
# Token specifications (as regexs)
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# Token processing functions
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Error handler
def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the lexer
lexer = lex()

# Grammar rules and handler functions
def p_expr(p):
    '''
    expr : expr PLUS term
         | expr MINUS term
    '''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]


def p_expr_term(p):
    '''
    expr : term
    '''
    p[0] = p[1]


def p_term(p):
    '''
    term : term TIMES factor
    | term DIVIDE factor
    '''
    if p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]

def p_term_factor(p):
    '''
    term : factor
    '''
    p[0] = p[1]

def p_factor(p):
    '''
    factor : NUM
```

```
    '''
    p[0] = p[1]

def p_factor_group(p):
    '''
    factor : LPAREN expr RPAREN
    '''
    p[0] = p[2]

def p_error(p):
    print('Syntax error')

parser = yacc()
```

这个程序中，所有代码都位于一个比较高的层次。你只需要为令牌写正则表达式和规则匹配时的高阶处理函数即可。而实际的运行解析器，接受令牌等等底层动作已经被库函数实现了。

下面是一个怎样使用得到的解析对象的例子：

```
>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>
```

如果你想在你的编程过程中来点挑战和刺激，编写解析器和编译器是个不错的选择。 再次，一本编译器的书籍会包含很多底层的理论知识。不过很多好的资源也可以在网上找到。 Python自己的ast模块也值得去看一下。

2.2 字符串开头或结尾匹配¶

问题¶

你需要通过指定的文本模式去检查字符串的开头或者结尾，比如文件名后缀，URL Scheme等等。

解决方案¶

检查字符串开头或结尾的一个简单方法是使用 `str.startswith()` 或者是 `str.endswith()` 方法。比如：

```
>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>
```

如果你想检查多种匹配可能，只需要将所有的匹配项放入到一个元组中去， 然后传给 `startswith()` 或者 `endswith()` 方法：

```
>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('.c', '.h')) ]
['foo.c', 'spam.c', 'spam.h'
>>> any(name.endswith('.py') for name in filenames)
True
>>>
```

下面是另一个例子：

```
from urllib.request import urlopen

def read_data(name):
    if name.startswith(('http:', 'https:', 'ftp:')):
        return urlopen(name).read()
    else:
        with open(name) as f:
            return f.read()
```

奇怪的是，这个方法中必须要输入一个元组作为参数。 如果你恰巧有一个 `list` 或者 `set` 类型的选择项， 要确保传递参数前先调用 `tuple()` 将其转换为元组类型。比如：

```
>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, not list
>>> url.startswith(tuple(choices))
True
>>>
```

讨论¶

`startswith()` 和 `endswith()` 方法提供了一个非常方便的方式去做字符串开头和结尾的检查。 类似的操作也可以使用切片来实现，但是代码看起来没有那么优雅。比如：

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
```

```
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or url[:6] == 'https:' or url[:4] == 'ftp:'
True
>>>
```

你可以能还想使用正则表达式去实现，比如：

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
<_sre.SRE_Match object at 0x101253098>
>>>
```

这种方式也行得通，但是对于简单的匹配实在是有点小材大用了，本节中的方法更加简单并且运行会更快些。

最后提一下，当和其他操作比如普通数据聚合相结合的时候 `startswith()` 和 `endswith()` 方法是很不错的。 比如，下面这个语句检查某个文件夹中是否存在指定的文件类型：

```
if any(name.endswith(('.c', '.h')) for name in listdir(dirname)):
...
```

2.20 字节字符串上的字符串操作¶

问题¶

你想在字节字符串上执行普通的文本操作(比如移除，搜索和替换)。

解决方案¶

字节字符串同样也支持大部分和文本字符串一样的内置操作。比如：

```
>>> data = b'Hello World'
>>> data[0:5]
b'Hello'
>>> data.startswith(b'Hello')
True
>>> data.split()
[b'Hello', b'World']
>>> data.replace(b'Hello', b'Hello Cruel')
b'Hello Cruel World'
>>>
```

这些操作同样也适用于字节数组。比如：

```
>>> data = bytearray(b'Hello World')
>>> data[0:5]
bytearray(b'Hello')
>>> data.startswith(b'Hello')
True
>>> data.split()
[bytearray(b'Hello'), bytearray(b'World')]
>>> data.replace(b'Hello', b'Hello Cruel')
bytearray(b'Hello Cruel World')
>>>
```

你可以使用正则表达式匹配字节字符串，但是正则表达式本身必须也是字节串。比如：

```
>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re
>>> re.split('[:,]',data)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/local/lib/python3.3/re.py", line 191, in split
return _compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object
>>> re.split(b'[:,]',data) # Notice: pattern as bytes
[b'FOO', b'BAR', b'SPAM']
>>>
```

讨论¶

大多数情况下，在文本字符串上的操作均可用于字节字符串。 然而，这里也有一些需要注意的不同点。首先，字节字符串的索引操作返回整数而不是单独字符。比如：

```
>>> a = 'Hello World' # Text string
>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World' # Byte string
>>> b[0]
72
>>> b[1]
101
>>>
```

这种语义上的区别会对于处理面向字节的字符数据有影响。

第二点，字节字符串不会提供一个美观的字符串表示，也不能很好的打印出来，除非它们先被解码为一个文本字符串。比如：

```
>>> s = b'Hello World'
>>> print(s)
b'Hello World' # Observe b'...'
>>> print(s.decode('ascii'))
Hello World
>>>
```

类似的，也不存在任何适用于字节字符串的格式化操作：

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'
>>> b'{} {} {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

如果你想格式化字节字符串，你得先使用标准的文本字符串，然后将其编码为字节字符串。比如：

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')
b'ACME 100 490.10'
>>>
```

最后需要注意的是，使用字节字符串可能会改变一些操作的语义，特别是那些跟文件系统有关的操作。 比如，如果你使用一个编码为字节的文件名，而不是一个普通的文本字符串，会禁用文件名的编码/解码。比如：

```
>>> # Write a UTF-8 filename
>>> with open('jalape\xf1o.txt', 'w') as f:
...     f.write('spicy')
...
>>> # Get a directory listing
>>> import os
>>> os.listdir('.') # Text string (names are decoded)
['jalapeño.txt']
>>> os.listdir(b'.') # Byte string (names left as bytes)
[b'jalapen\xcc\x83o.txt']
>>>
```

注意例子中的最后部分给目录名传递一个字节字符串是怎样导致结果中文件名以未解码字节返回的。 在目录中的文件名包含原始的UTF-8编码。 参考5.15小节获取更多文件名相关的内容。

最后提一点，一些程序员为了提升程序执行的速度会倾向于使用字节字符串而不是文本字符串。 尽管操作字节字符串确实会比文本更加高效(因为处理文本固有的Unicode相关开销)。 这样做通常会导致非常杂乱的代码。你会经常发现字节字符串并不能和Python的其他部分工作的很好， 并且你还得手动处理所有的编码/解码操作。 坦白讲，如果你在处理文本的话，就直接在程序中使用普通的文本字符串而不是字节字符串。不做死就不会死！

2.3 用Shell通配符匹配字符串¶

问题¶

你想使用 **Unix Shell** 中常用的通配符(比如 `*.py`, `Dat[0-9]*.csv` 等)去匹配文本字符串

解决方案¶

`fnmatch` 模块提供了两个函数——`fnmatch()` 和 `fnmatchcase()` ，可以用来实现这样的匹配。用法如下：

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

`fnmatch()` 函数使用底层操作系统的大小写敏感规则(不同的系统是不一样的)来匹配模式。比如：

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False
>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

如果你对这个区别很在意，可以使用 `fnmatchcase()` 来代替。它完全使用你的模式大小写匹配。比如：

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
>>>
```

这两个函数通常会被忽略的一个特性是在处理非文件名的字符串时候它们也是很有用的。 比如，假设你有一个街道地址的列表数据：

```
addresses = [
    '5412 N CLARK ST',
    '1060 W ADDISON ST',
    '1039 W GRANVILLE AVE',
    '2122 N CLARK ST',
    '4802 N BROADWAY',
]
```

你可以像这样写列表推导：

```
>>> from fnmatch import fnmatchcase
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] *CLARK*')]
['5412 N CLARK ST']
>>>
```

讨论¶

`fnmatch()` 函数匹配能力介于简单的字符串方法和强大的正则表达式之间。 如果在数据处理操作中只需要简单的通配符就能完成的时候，这通常是一个比较合理的方案。

如果你的代码需要做文件名的匹配，最好使用 `glob` 模块。参考5.13小节。

2.4 字符串匹配和搜索¶

问题¶

你想匹配或者搜索特定模式的文本

解决方案¶

如果你想匹配的是字面字符串，那么你通常只需要调用基本字符串方法就行， 比如 `str.find()`,`str.endswith()`, `str.startswith()` 或者类似的方法：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> # Exact match
>>> text == 'yeah'
False
>>> # Match at start or end
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False
>>> # Search for the location of the first occurrence
>>> text.find('no')
10
>>>
```

对于复杂的匹配需要使用正则表达式和 `re` 模块。 为了解释正则表达式的基本原理，假设你想匹配数字格式的日期字符串比如 `11/27/2012`，你可以这样做：

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
>>> # Simple matching: \d+ means match one or more digits
>>> if re.match(r'\d+/\d+/\d+', text1):
... print('yes')
... else:
... print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
... print('yes')
... else:
... print('no')
...
no
>>>
```

如果你想使用同一个模式去做多次匹配，你应该先将模式字符串预编译为模式对象。比如：

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
... print('yes')
... else:
... print('no')
...
yes
>>> if datepat.match(text2):
... print('yes')
... else:
... print('no')
...
no
>>>
```

`match()` 总是从字符串开始去匹配，如果你想查找字符串任意部分的模式出现位置， 使用 `findall()` 方法去代替。比

如：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

在定义正则式的时候，通常会利用括号去捕获分组。比如：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

捕获分组可以使得后面的处理更加简单，因为可以分别将每个组的内容提取出来。比如：

```
>>> m = datepat.match('11/27/2012')
>>> m
<_sre.SRE_Match object at 0x1005d2750>
>>> # Extract the contents of each group
>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
('11', '27', '2012')
>>> month, day, year = m.groups()
>>>
>>> # Find all matches (notice splitting into tuples)
>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for month, day, year in datepat.findall(text):
...     print('{}-{}-{}'.format(year, month, day))
...
2012-11-27
2013-3-13
>>>
```

`findall()` 方法会搜索文本并以列表形式返回所有的匹配。 如果你想以迭代方式返回匹配，可以使用 `finditer()` 方法来代替，比如：

```
>>> for m in datepat.finditer(text):
...     print(m.groups())
...
('11', '27', '2012')
('3', '13', '2013')
>>>
```

讨论¶

关于正则表达式理论的教程已经超出了本书的范围。 不过，这一节阐述了使用re模块进行匹配和搜索文本的最基本方法。 核心步骤就是先使用 `re.compile()` 编译正则表达式字符串， 然后使用 `match()` , `findall()` 或者 `finditer()` 等方法。

当写正则式字符串的时候，相对普遍的做法是使用原始字符串比如 `r'(\d+)/(\d+)/(\d+)'` 。 这种字符串将不去解析反斜杠，这在正则表达式中是很有用的。 如果不这样做的话，你必须使用两个反斜杠，类似 `'(\\d+)/(\\d+)/(\\d+)'` 。

需要注意的是 `match()` 方法仅仅检查字符串的开始部分。它的匹配结果有可能并不是你期望的那样。比如：

```
>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
```

```
>>> m.group()
'11/27/2012'
>>>
```

如果你想精确匹配，确保你的正则表达式以$结尾，就像这么这样：

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

最后，如果你仅仅是做一次简单的文本匹配/搜索操作的话，可以略过编译部分，直接使用 `re` 模块级别的函数。比如：

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

但是需要注意的是，如果你打算做大量的匹配和搜索操作的话，最好先编译正则表达式，然后再重复使用它。 模块级别的函数会将最近编译过的模式缓存起来，因此并不会消耗太多的性能， 但是如果使用预编译模式的话，你将会减少查找和一些额外的处理损耗。

2.5 字符串搜索和替换¶

问题¶

你想在字符串中搜索和匹配指定的文本模式

解决方案¶

对于简单的字面模式，直接使用 `str.replace()` 方法即可，比如：

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

对于复杂的模式，请使用 `re` 模块中的 `sub()` 函数。 为了说明这个，假设你想将形式为 `11/27/2012` 的日期字符串改成 `2012-11-27` 。示例如下：

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

`sub()` 函数中的第一个参数是被匹配的模式，第二个参数是替换模式。反斜杠数字比如 `\3` 指向前面模式的捕获组号。

如果你打算用相同的模式做多次替换，考虑先编译它来提升性能。比如：

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

如果你使用了命名分组，那么第二个参数请使用 `\g<group_name>` ,如下

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(?P<month>\d+)/(?P<day>\d+)/(?P<year>\d+)', r'\g<year>-\g<month>-\g<day>', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

对于更加复杂的替换，可以传递一个替换回调函数来代替，比如：

```
>>> from calendar import month_abbr
>>> def change_date(m):
... mon_name = month_abbr[int(m.group(1))]
... return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

一个替换回调函数的参数是一个 `match` 对象，也就是 `match()` 或者 `find()` 返回的对象。 使用 `group()` 方法来提取特定的匹配部分。回调函数最后返回替换字符串。

如果除了替换后的结果外，你还想知道有多少替换发生了，可以使用 `re.subn()` 来代替。比如：

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

讨论¶

关于正则表达式搜索和替换，上面演示的 `sub()` 方法基本已经涵盖了所有。 其实最难的部分就是编写正则表达式模式，这个最好是留给读者自己去练习了。

2.6 字符串忽略大小写的搜索替换¶

问题¶

你需要以忽略大小写的方式搜索与替换文本字符串

解决方案¶

为了在文本操作时忽略大小写，你需要在使用 `re` 模块的时候给这些操作提供 `re.IGNORECASE` 标志参数。比如：

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

最后的那个例子揭示了一个小缺陷，替换字符串并不会自动跟被匹配字符串的大小写保持一致。 为了修复这个，你可能需要一个辅助函数，就像下面的这样：

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
        else:
            return word
    return replace
```

下面是使用上述函数的方法：

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

译者注： `matchcase('snake')` 返回了一个回调函数(参数必须是 `match` 对象)，前面一节提到过， `sub()` 函数除了接受替换字符串外，还能接受一个回调函数。

讨论¶

对于一般的忽略大小写的匹配操作，简单的传递一个 `re.IGNORECASE` 标志参数就已经足够了。 但是需要注意的是，这个对于某些需要大小写转换的Unicode匹配可能还不够， 参考2.10小节了解更多细节。

2.7 最短匹配模式¶

问题¶

你正在试着用正则表达式匹配某个文本模式，但是它找到的是模式的最长可能匹配。 而你想修改它变成查找最短的可能匹配。

解决方案¶

这个问题一般出现在需要匹配一对分隔符之间的文本的时候(比如引号包含的字符串)。 为了说明清楚，考虑如下的例子：

```
>>> str_pat = re.compile(r'"(.*)"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

在这个例子中，模式 `r'\"(.*)\"'` 的意图是匹配被双引号包含的文本。 但是在正则表达式中*操作符是贪婪的，因此匹配操作会查找最长的可能匹配。 于是在第二个例子中搜索 `text2` 的时候返回结果并不是我们想要的。

为了修正这个问题，可以在模式中的*操作符后面加上?修饰符，就像这样：

```
>>> str_pat = re.compile(r'"(.*?)"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

这样就使得匹配变成非贪婪模式，从而得到最短的匹配，也就是我们想要的结果。

讨论¶

这一节展示了在写包含点(.)字符的正则表达式的时候遇到的一些常见问题。 在一个模式字符串中，点(.)匹配除了换行外的任何字符。 然而，如果你将点(.)号放在开始与结束符(比如引号)之间的时候，那么匹配操作会查找符合模式的最长可能匹配。 这样通常会导致很多中间的被开始与结束符包含的文本被忽略掉，并最终被包含在匹配结果字符串中返回。 通过在 `*` 或者 `+` 这样的操作符后面添加一个 `?` 可以强制匹配算法改成寻找最短的可能匹配。

2.8 多行匹配模式¶

问题¶

你正在试着使用正则表达式去匹配一大块的文本，而你需要跨越多行去匹配。

解决方案¶

这个问题很典型的出现在当你用点(.)去匹配任意字符的时候，忘记了点(.)不能匹配换行符的事实。 比如，假设你想试着去匹配C语言分割的注释：

```
>>> comment = re.compile(r'/\*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
... multiline comment */
... '''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

为了修正这个问题，你可以修改模式字符串，增加对换行的支持。比如：

```
>>> comment = re.compile(r'/\*((?:.|\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\n multiline comment ']
>>>
```

在这个模式中， `(?:.|\n)` 指定了一个非捕获组 (也就是它定义了一个仅仅用来做匹配，而不能通过单独捕获或者编号的组)。

讨论¶

`re.compile()` 函数接受一个标志参数叫 `re.DOTALL` ，在这里非常有用。 它可以让正则表达式中的点(.)匹配包括换行符在内的任意字符。比如：

```
>>> comment = re.compile(r'/\*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n multiline comment ']
```

对于简单的情况使用 `re.DOTALL` 标记参数工作的很好， 但是如果模式非常复杂或者是为了构造字符串令牌而将多个模式合并起来(2.18节有详细描述)， 这时候使用这个标记参数就可能出现一些问题。 如果让你选择的话，最好还是定义自己的正则表达式模式，这样它可以在不需要额外的标记参数下也能工作的很好。

2.9 将Unicode文本标准化¶

问题¶

你正在处理Unicode字符串，需要确保所有字符串在底层有相同的表示。

解决方案¶

在Unicode中，某些字符能够用多个合法的编码表示。为了说明，考虑下面的这个例子：

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalapeño'
>>> s2
'Spicy Jalapeño'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

这里的文本”Spicy Jalapeño”使用了两种形式来表示。 第一种使用整体字符”ñ”(U+00F1)，第二种使用拉丁字母”n”后面跟一个”~”的组合字符(U+0303)。

在需要比较字符串的程序中使用字符的多种表示会产生问题。 为了修正这个问题，你可以使用unicodedata模块先将文本标准化：

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'
>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

`normalize()` 第一个参数指定字符串标准化的方式。 NFC表示字符应该是整体组成(比如可能的话就使用单一编码)，而NFD表示字符应该分解为多个组合字符表示。

Python同样支持扩展的标准化形式NFKC和NFKD，它们在处理某些字符的时候增加了额外的兼容特性。比如：

```
>>> s = '\ufb01' # A single character
>>> s
'fi'
>>> unicodedata.normalize('NFD', s)
'fi'
# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

讨论¶

标准化对于任何需要以一致的方式处理Unicode文本的程序都是非常重要的。 当处理来自用户输入的字符串而你很难去控制编码的时候尤其如此。

在清理和过滤文本的时候字符的标准化也是很重要的。 比如，假设你想清除掉一些文本上面的变音符的时候(可能是为了搜索和匹配)：

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

最后一个例子展示了 `unicodedata` 模块的另一个重要方面，也就是测试字符类的工具函数。 `combining()` 函数可以测试一个字符是否为和音字符。 在这个模块中还有其他函数用于查找字符类别，测试是否为数字字符等等。

Unicode显然是一个很大的主题。如果想更深入的了解关于标准化方面的信息， 请看考 Unicode官网中关于这部分的说明 Ned Batchelder在 他的网站 上对Python的Unicode处理问题也有一个很好的介绍。

第二章：字符串和文本¶

几乎所有有用的程序都会涉及到某些文本处理，不管是解析数据还是产生输出。 这一章将重点关注文本的操作处理，比如提取字符串，搜索，替换以及解析等。 大部分的问题都能简单的调用字符串的内建方法完成。 但是，一些更为复杂的操作可能需要正则表达式或者强大的解析器，所有这些主题我们都会详细讲解。 并且在操作Unicode时候碰到的一些棘手的问题在这里也会被提及到。

Contents: