**第七章**

**字符串**

根据各种形式编程中字符串的基本性质，Python中的字符串可以占满整个章节就不足为奇了。用户交互，键盘输入，通过网络发送内容，分析大数据，或者参加图灵测试，字符串有很多用途。

由于所有这些对字符串的强调，Python确保包含支持他们的各种特性。其中一些特性直接内置在字符串对象中，其他的由标准库的模块和第三方数据库提供，如Boost 提供。这一章，将主要介绍python内置字符串函数而不是第三方函数库。

想要学习python字符串首先要了解它的两种不同类型：字节和Unicode字符串。让我们首先来学习字节。

**字节**

一个字符串在非常基础的层面上只是一个单独字节序。一般来说，在电脑计算的每一个数据片段都要使用字节。数字、字符串和稍微复杂的项目都可能储存为字节，一些结构更复杂的构建在字节序列之上。字节字符串，以字节对象的形式体现在Python中，每个字符只表示一个字节，所以易于与其他文件和外界接口进行交互。

标准字符串——在下一章中介绍——仅仅只用一对单引号确定文字，字节字符串第一个引号前有一个b。这是在源代码以及这些值的repr（）输出时使用的：

*>>> b'example' #the keyword print is assumed since it is a command prompt*

*statement versus a script.*

*b'example'*

字节的基本作用是表达非文本信息，例如，数字，数据，设置标志和许多别的信息。即使Python无法直接处理这些特殊的值，字节对象能确保它们可以原封不动的被处理，保证你的代码可以准确无误。没有任何关于数据意图的设想，字节拥有最大的灵活性，但你将需要一些把数据在字节和一些对你的应用有更多意义的信息之间来回转换的方法。

简单的转换：*chr（）和ord（）*

在非常基础的层面上，字节只是用某种字符表示的数字。在Python中，数字和字符是两种不同的东西，但它们的值是相等的，所以他们的转换是非常容易的。给你一个单独的字节，你可以把它传递到能够把它还原成整数值的内置函数中。

*>>> ord(b'A')*

*65*

*>>> ord(b'!')*

*33*

*>>> list(b'Example')*

*[69, 120, 97, 109, 112, 108, 101]*

注意遍历字节序列迭代时会发生什么。而不是字符的字节字符串，你可以立即获得原始整数，完全不需要ord（）。这样单个字节值从字节转换成数字很有效，但反向转换就需要内置chr（）函数了。与ord（）相反，它根据传入的整数值返回单个字符：

*>>> chr(65)*

*'A'*

*>>> chr(33)*

*'!'*

*>>> [chr(o) for o in [69, 120, 97, 109, 112, 108, 101]]*

*['E', 'x', 'a', 'm', 'p', 'l', 'e']*

*>>> “.join(chr(o) for o in [69, 120, 97, 109, 112, 108, 101])*

*'Example'*

有一个要点需要注意：缺少一个 b 前缀，说明用*chr（）*函数转换的字符串是常规字符串，而不是字节字符串。你将在本章后面的文本一节中看到，标准字符串和字节字符串运算的不同。然而，对于我们来说，最大的问题是一个标准字符串不能等同于单个字节，因此很容易出错。为了运算准确和获取额外功能，我们可以使用*struct*模块。

复杂的转换：*struct*模块

除了*chr()*返回标准字符串的问题之外，*ord（）/chr（）*组合的一个大问题是，它只能在处理单个字节时可靠地使用。在将数字转换为字节时，值限制在0到255之间。

为了支持更大范围的值和其他一些有趣的特性，Python提供了*struct*模块。与*chr（）*和*ord（）*在字节字符串和本机Python值之间转换值的方式类似。*Pack（）*写字节字符串，而*struct.Unpack（）*将这些值读入Python。但是，与那些简单的函数不同，*struct*模块使用格式字符串来限定应该如何转换值。这种格式有自己的一种简单程序来控制使用什么类型的值以及它们如何工作。因为我们使用struct是为了克服*chr（）的*一些困难，所以我们将从如何*struct*开始。*Pack（）*可以提供预期的功能。单个无符号字节的格式是*B*，在实践中你可以这样使用：

*>>> import struct*

*>>> struct.pack(b'B', 65)*

*b'A'*

*>>> struct.pack(b'B', 33)*

*b'!'*

*>>> struct.pack(b'BBBBBBB', 69, 120, 97, 109, 112, 108, 101)*

*b'Example'*

可以看到，第一个参数是格式字符串本身，每个参数都有一个字符，应该转换为字节字符串。所有附加参数都用于提供应该转换的值。因此，对于每个格式说明符，前面提到的每个等效位置包含一个参数，*B*指定了一个无符号值，这意味着不能有负值。使用它，你可以输入0到255之间的值，但不能输入低于0的值。相反，带符号的值可以使用字节中8位中的一位来识别值是正还是负。仍然有256个唯一的值 ，但范围发生了变化，所以一半的值在符号的两边。因为0是一个正数，一个带符号的字节可以包含从-128到127的值。为了补充无符号字节，有符号字节的格式说明符是*b*

*>>> struct.pack(b'b', 65)*

*b'A'*

*>>> struct.pack(b'Bb', 65, -23)*

*b'A\xe9'*

*>>> struct.pack(b'B', 130)*

*b'\x82'*

*>>> struct.pack(b'b', 130)*

*Traceback (most recent call last):*

*...*

*struct.error: byte format requires -128 <= number <= 127*

当然*B*和*b*只对单字节值有效，上限为256个值。为了范围能够更大，你可以使用*H*和*h*来表示两个字节的数字，上限为65536个值。与单字节相同，大写格式采用无符号值，而小写符号采用有符号值。

*>>> struct.pack(b'Hh', 42, -137)*

*b'\*\x00w\xff'*

既然一个值可以对应多个字节，就出现了哪个字节先出现的问题。两个字节中的一个包含256个最小的值，而另一个包含0到256的值，但是需要乘以256。将两者混在一起会极大地影响存储或检索。只要使用逆函数struct.unpack（），就能快速地分清楚。

*>>> struct.unpack(b'H', b'\*\x00')*

*(42,)*

*>>> struct.unpack(b'H', b'\x00\*')*

*(10752,)*

如你所见，*struct.unpack（）*函数的使用与*struct.pack（）*十分相似，但也有几个明显差别。首先，*unpack（）*总是只有两个参数，因为第二个参数是原始字节字符串。这个字符串可以包含多个需要提取的值，但是与*pack（）*不同，它仍然只作为一个参数传递。

相反，返回值是一个元组，它可以包含多个值。因此可以说*struct.Unpack（）*是*struct.pack（）*的真逆，也就是说，你可以将一个结果传递到另一个调用中，并获得与第一次的值相同的值。你需要确保在每个单独函数调用中使用相同格式：

*>>> struct.unpack(b'Hh', struct.pack(b'Hh', 42, -42))*

*(42, -42)*

*>>> struct.pack(b'Hh', \*struct.unpack(b'Hh', b'\*\x00\x00\*'))*

*b'\*\x00\x00\*'*

那么，跨多个字节的值有什么问题呢?毕竟，这些示例表明可以将值转换为字符串并返回，而不必担心如何创建或解析这些字符串。不幸的是，这很容易，因为我们目前只在Python中工作，它有一个与自身一致的实现。如果必须处理，需要与其他应用程序一起使用的字符串，如文件内容，则需要确保与这些应用程序预期相匹配。

因此，*struct*格式还允许你详细说明其字节顺序。字节顺序是指一个值的字节是如何排序的;在big-endian值中，最重要的字节(提供数字最大部分的字节)首先存储。对于little-endian字节值，首先存储最小有效字节。

为了区分这两者，格式规范可以使用一个前缀。如果在格式前面放置一个<，则可以指定其为little-endian。相反，使用>将标记为big-endian。如果这两个选项都没有提供，就像前面的示例一样，那么默认的行为是使用与Python执行的系统相同的字节顺序，在现代系统中通常是little-endian。这允许你控制对pack()和unpack()的值的处理方式，如下所示两种不同的过程：

*>>> struct.pack(b'<H', 42)*

*b'\*\x00'*

*>>> struct.pack(b'>H', 42)*

*b'\x00\*'*

*>>> struct.unpack(b'<H', b'\*\x00')*

*(42,)*

*>>> struct.unpack(b'>H', b'\*\x00')*

*(10752,)*

既然可以控制多字节数的顺序了，那么处理较大的值就更容易了。除了前面讨论的一个和两个字节整数之外，struct还支持使用I和i的四字节值，而使用Q和q可以指定八字节值。与其他整数一样，大写字母表示无符号值，而小写字母表示有符号值。

然而，struct模块不仅仅是转换整数。你还可以使用f (甚至可能是b)转换浮点值，以获得更高的精度。实际上，you can use struct to work with strings inside strings as well，这提供了一些额外的灵活性。使用s格式代码并结合数字前缀，表示要读写的字符串的大小:

*>>> struct.pack(b'7s', b'example')*

*b'example'*

*>>> struct.unpack(b'7s', b'example')*

*(b'example',)*

*>>> struct.pack(b'10s', b'example')*

*b'example\x00\x00\x00'*

可以看到，pack()将添加空字节，根据格式符，转换为字符串。但为什么要使用struct转换为字符串呢?这样做的好处是可以一次打包多个值，所以字符串可能只是结构的一部分。考虑一个包含联系人信息的简单字节字符串:

*>>>import struct*

*>>>first\_name = 'Marty'*

*>>> last\_name = 'Alchin'*

*>>> age = 28*

*>>> struct.pack(b'10s10sB', bytes(first\_name, 'utf8'), bytes(last\_name, 'utf8'), age)*

*>>> data*

*b'Alchin\x00\x00\x00\x00Marty\x00\x00\x00\x00\x00\x1c'*

但是，如果希望以这种方式处理字符串，则更可能处理文本，在文本中，字符串作为一个整体具有意义，而不是其字符，是其他类型值的转换。在Python 3.2中，实现方法的格式发生了一点变化，所以现在必须将字符串(str)文本编码为字节字符串，通常是utf8编码。

**文本**

从概念上讲，文本是书面文字的集合。这是一个早在计算机出现之前就存在的语言概念，但当计算机需要处理文本时，就有必要研究如何为数字设计的系统中表示文本。在编程初期，文本仅限于一组字符，即美国信息交换标准代码(ASCII)或EBCDIC或其他字符。

注意这里提到的“American”;这一组127个字符——其中只有95个是可打印的——旨在满足英语语言的需要。ASCII只包含每个字节的7位，因此有未来任有可扩展空间，但即使有另外128个值也不够。一些应用程序使用特殊的技巧，通过着重和其他标记来传递额外的字母，但是标准的范围仍然非常有限。

**Unicode**

为了解决这个限制，Unicode标准成为了一种替代标准，它可以包含世界上绝大多数语言中使用的大多数字符。为了使Unicode支持尽可能多的代码点，每个代码点占用一个字节以上，这与ASCII不同。在内存中加载时，这不成问题，因为它只在Python中使用，Python只有一种管理多字节值的方法。

、

注意：Unicode标准实际上是由多个单独的“代码点”组成的，而不是字符。代码点是表示书面文本某个方面的数字，可以是正则字符、符号或修饰符。有些字符甚至出现在多个代码点，以便与Unicode引入之前使用的系统兼容。

默认情况下，Python中的所有标准字符串都是Unicode，在这个过程中支持多种语言。上一节中显示的字节字符串都需要使用b前缀来区别于标准的Unicode字符串。

当将这些值写入可以被其他系统读取的字符串时，问题就来了，因为不是所有的系统都使用相同的Unicode字符串内部表示。相反，有几种不同的编码可以用来将一个Unicode字符串折叠成一系列字节以便存储或分布。

**Encodings**

就像可以使用多个字节来存储大于一个字节的数字一样，Unicode文本可以存储为多字节格式。然而，与数字不同的是，文本通常包含大量的单个字符，因此将每个字符最多存储4个字节，意味着一段很长的文本可能比看起来要大得多。

为了尽可能有效地支持文本，很快发现并不是所有文本都需要全部可用字符。例如，is written in English，这意味着它的大部分内容都在ASCII范围内。因此，它可以从每个字符四个字节减少到一个字节。

ASCII是文本编码的一种。在本例中，将一组小的可用字符被映射到从0到127的特定值。所选择的字符旨在支持英语，因此它包含所有可用的大写和小写字母变体、所有10个数字和各种标点选项。任何包含这些值的文本都可以使用ASCII编码转换为字节。

编码过程本身使用字符串的encode()方法进行管理。只需传入编码的名称，它将返回一个字节字符串，表示给定编码中的文本。在ASCII的情况下，字节字符串的表示形式看起来就像输入文本，因为每个字节正好映射到一个字符：

*>>> 'This is an example, with punctuation and UPPERCASE.'.encode('ascii')*

*b'This is an example, with punctuation and UPPERCASE.'*

通过将每个字节映射到单个字符，ASCII非常有效，但它仅在源文本包含编码中指定的那些字符时才有效。对于哪些角色重要到足以在如此小的范围内出现，必须做出一定的假设。其他语言将有它们自己的优先字符，因此它们使用不同的编码，以达到与ASCII (UTF-8是最流行的)一样的效率。

一些语言，包括中文和日文，有太多的字符，一个字节根本无法表示它们。这些语言的一些编码为每个字符使用两个字节，进一步强调了各种文本编码的不同之处。因此，为特定语言设计的编码通常不能用于该语言之外的文本。

为了解决这个问题，有一些更通用的以Unicode为中心的编码。由于可用字符的数量太多，这些编码使用了变长的方法。在UTF-8中最常见，一定范围内的字符可以用一个字节表示。其他字符需要两个字节，还有一些字符可以使用三个甚至四个字节。UTF-8是可取的，因为它表现出一些特殊的特点:

它可以支持任何可用的Unicode编码点，即使在实际文本中并不常见。该特性并不是UTF-8独有的，但它确实将UTF-8与其他特定于语言的编码(如ASCII)区分开来。

字符在实际使用中越常见，其代码点占用的空间就越少。例如，在大多数英文文档中，UTF-8几乎与ASCII一样重要。即使在编码非英语文本时，大多数语言也共享某些公共字符，如空格和标点符号，这些字符可以用单个字节进行编码。当它必须使用两个字节时，它仍然比内存中的Unicode对象更有效。

单字节范围与ASCII标准完全一致，使UTF-8与ASCII文本完全兼容。所有ASCII文本都可以被读取为UTF-8，无需修改。同样，只包含ASCII字符的文本可以使用UTF-8进行编码，并且仍然可以被只理解ASCII的应用程序访问。

由于这些原因，UTF-8已经成为一种非常常见的编码(自2008年以来)，用于需要支持多种语言的应用程序，或者应用程序的语言在设计时并不为人所知。这似乎是一种奇怪的情况，但是在研究框架、库和其他大型应用程序时，这种情况经常出现。它们可以部署在地球上的任何环境中，因此它们应该尽可能多地支持其他语言。第8章将更详细地描述应用程序支持多种语言的步骤。

根据应用程序的需要、使用的编码和传入的文本，使用错误编码或解码的结果可能会有所不同。例如，可以使用UTF-8对ASCII文本进行解码，而不会产生任何问题，从而生成完全有效的Unicode字符串。反转这个过程并不总是那么容易，因为Unicode字符串可以包含有效ASCII范围之外的代码点:

*>>> ascii = 'This is a test'.encode('ascii')*

*>>> ascii b'This is a test'*

*>>> ascii.decode('utf-8')*

*'This is a test'*

*>>> unicode = 'This is a test: \u20ac' # A manually encoded Euro symbol*

*>>> unicode.encode('utf-8')*

*b'This is a test: \xe2\x82\xac'*

*>>> unicode.encode('ascii')*

*Traceback (most recent call last):*

*...*

*UnicodeEncodeError: 'ascii' codec can't encode character '\u20ac' in position 16*

*: ordinal not in range(128)*

在其他时候，文本似乎可以被正确地编码或解码，但结果文本却是胡言乱语。从头开始构建Unicode应用程序并不能完全消除这些问题的可能性，但它极大地帮助避免了这些问题。

**Simple Substitution**

有许多不同的方法可以生成只在运行时可用的信息。也许最明显的是使用 +操作符将多个字符串连接在一起，但是只有当所有的值都是字符串时才有效。Python不会隐式地将其他值转换为要连接的字符串，因此必须显式地转换它们，例如，首先将它们传递到str()函数中。

作为一种选择，Python字符串还支持一种向字符串中注入对象的方法。它在字符串中使用占位符来指示对象的位置，以及应该将他们放入的对象集合。这称为字符串替换，使用%操作符，使用自定义的\_\_mod\_\_()方法来执行，如第5章所述。

占位符由百分号和转换格式组成，可以选择在它们之间添加一些修饰符来指定转换是如何发生的。该方案允许字符串指定如何转换对象，而不必显式地调用单独的函数。这些格式中最常见的是%s，它相当于直接使用str()函数:

*>>> 'This object is %s' % 1*

*'This object is 1'*

*>>> 'This object is %s' % object()*

*'This object is <object object at 0x...>'*

因为这相当于直接调用str()，所以放在字符串中的值是调用对象的str\_\_()方法的结果。类似地，如果在替换字符串中使用%r占位符，Python将调用对象的repr\_\_()方法。例如，这对于记录函数的参数非常有用。尝试下一个例子作为脚本:

*def func(\*args):*

*for i, arg in enumerate(args):*

*print('Argument %s: %r' % (i, arg))*

*func('example', {}, [1, 2, 3], object())*

*Your output will look like the following:*

*Argument 0: 'example'*

*Argument 1: {}*

*Argument 2: [1, 2, 3]*

*Argument 3: <object object at 0x...>*

这个示例还演示了如何通过将多个值包装在一个tuple中，将它们同时放在字符串中。它们根据位置与字符串中的对应对象进行匹配，因此第一个对象放在第一个占位符中，等等。不幸的是，如果你不小心的话，这个特性有时也会成为你的绊脚石。最常见的错误发生在试图将一个元组注入替换字符串时:

*>>> def log(\*args):*

*... print('Logging arguments: %r' % args)*

*...*

*>>> log('test') "Logging arguments: 'test'"*

*>>> log('test', 'ing')*

*Traceback (most recent call last):*

*...*

*TypeError: not all arguments converted during string formatting*

这里所发生的是，Python没有区分在源代码中写成这样的元组和仅仅从其他地方传递过来的元组。因此，字符串替换无法知道你的意图是什么。在本例中，只要传入一个参数，替换就可以正常工作，因为字符串中只有一个占位符。一旦你传递了不止一个参数，它就会中断。

为了解决这个问题，你需要构建一个单项元组来包含你想放在字符串中的元组。通过这种方式，字符串替换总是得到一个元组，其中包含一个要放在单个占位符中的元组:

*>>> def log(\*args):*

*... print('Logging arguments: %r' % (args,))*

*...*

*>>> log('test')*

*"Logging arguments: ('test',)"*

*>>> log('test', 'ing')*

*"Logging arguments: ('test', 'ing')"*

在解决了元组问题之后，值得注意的是，对象也可以通过关键字插入。这样做需要替换字符串，包含括号中的关键字，立即省略百分号。然后，要传入要注入的值，只需传入对象的字典，而不是元组:

*>>> def log(\*args):*

*... for i, arg in enumerate(args):*

*... print('Argument %(i)s: %(arg)r' % {'i': i, 'arg': arg}) ...*

*>>> log('test') Argument 0: 'test'*

*>>> log('test', 'ing')*

*Argument 0: 'test'*

*Argument 1: 'ing'*

除了能够更容易地重新排列替换字符串中的占位符之外，该特性还允许你只包含那些重要的值。如果字符串中的值比需要的多，则只能引用需要的值。Python将简单地忽略字符串中没有提到名字的任何值。这与位置选项相反，在位置选项中，提供的值比你在字符串中标记的值多，将导致类型错误。

**格式化**

对于上一节中描述的简单字符串替换的更强大的替代方法，Python还包括一个强大的字符串格式化系统。字符串格式化不是依赖于一个不太明显的操作符，而是对字符串使用显式的format()方法。此外，用于格式化字符串的语法与以前简单替换中使用的语法有很大的不同。

format()不使用百分号和格式代码，而是希望占位符被花括号包围。这些大括号内的内容取决于你打算如何传递值以及它们应该如何格式化。占位符的第一部分决定它应该查找位置参数还是关键字参数。对于位置参数，内容是数字，表示要处理的值的索引，而对于关键字参数，则提供引用适当值的键:

*>>> 'This is argument 0: {0}'.format('test')*

*'This is argument 0: test'*

*>>> 'This is argument key: {key}'.format(key='value')*

*'This is argument key: value'*

这可能看起来很像旧的替换技术，但是它已经有了一个主要的优点。因为格式化是通过方法调用而不是操作符来启动的，所以可以同时指定位置参数和关键字参数。这样，如果需要，可以混合和匹配格式字符串中的索引和键，以任何顺序引用它们。

另外，这也意味着不需要在字符串中引用所有的位置参数才能正常工作。如果提供的内容超出了需要，format()将忽略任何没有占位符的内容。这使得将格式字符串传递给应用程序要容易得多，该应用程序稍后将调用format()，参数可能来自其他来源。一个这样的例子是一个验证函数，它在过程中接受一个错误消息:

*>>> def exact\_match(expected, error):*

*... def validator(value):*

*... if value != expected:*

*... raise ValueError(error.format(value, expected))*

*... return validator*

*...*

*>>> validate\_zero = exact\_match(0, 'Expected {1}, got {0}')*

*>>> validate\_zero(0)*

*>>> validate\_zero(1) Traceback (most recent call last):*

*...*

*ValueError: Expected 0, got 1*

*>>> validate\_zero = exact\_match(0, '{0} != {1}')*

*>>> validate\_zero(1) Traceback (most recent call last):*

*...*

*ValueError: 1 != 0*

*>>> validate\_zero = exact\_match(0, '{0} is not the right value')*

*>>> validate\_zero(1)*

*Traceback (most recent call last): ...*

*ValueError: 1 is not the right value*

由此可见，这个特性允许validator函数调用format()使用当时可用的所有信息，而让格式字符串决定如何布置它。对于另一个字符串替换，将被迫使用关键字来实现相同的效果，因为位置参数的工作方式不同。

**查找对象中的值**

除了能够引用传入的对象之外，格式字符串语法还允许你专门引用那些对象的部分。它的语法与普通Python代码非常相似。要引用属性，请使用句点将其名称与对象引用分隔开。若要使用索引或关键字值，请在方括号内提供索引或关键字;不要用引号括住关键词:

*>>> import datetime*

*>>> def format\_time(time):*

*... return '{0.minute} past {0.hour}'.format(time)*

*...*

*>>> format\_time(datetime.time(8, 10))*

*'10 past 8'*

*>>> '{0[spam]}'.format({'spam': 'eggs'})*

*'eggs'*

**区分字符串类型**

你可能还记得，简单的替换要求你指定%s或%r，以指示是否应该使用str（）类函数或repr()类函数将对象转换为字符串，而到目前为止给出的示例还没有包含这样的提示。默认情况下，format()将使用str()，但该行为仍然可以作为格式字符串的一部分进行控制。紧接对象引用之后，简单地包括一个感叹号，后面跟着s或r:

*>>> validate\_test = exact\_match('test', 'Expected {1!r}, got {0!r}')*

*>>> validate\_test('invalid')*

*Traceback (most recent call last):*

*...*

*ValueError: Expected 'test', got 'invalid'*

**标准格式规范**

这种新的字符串格式与以前的替换特性的真正不同之处在于，可以灵活地格式化对象的输出。在前面几节提到的字段引用和字符串类型之后，可以包括一个冒号，后跟一个控制引用对象格式的字符串。这种格式规范有一个通常适用于大多数对象的标准语法。

第一个选项控制输出字符串的对齐方式，在需要指定输出的最小字符数时使用。提供左尖括号(<)将产生左对齐的值;右尖括号(>)向右对齐;插入符号(^)将值居中。总宽度可以在后面指定一个数字:

*>>> import os.path*

*>>> '{0:>20}{1}'.format(\*os.path.splitext('contents.txt'))*

*' contents.txt'*

*>>> for filename in ['contents.txt', 'chapter.txt', 'index.txt']:*

*... print('{0:<10}{1}'.format(\*os.path.splitext(filename)))*

*...*

*contents .txt*

*chapter .txt*

*index .txt*

需要注意的是，长度规范的默认行为是用空格填充输出以达到必要的长度。这也可以通过在对齐说明符之前插入不同的字符来控制。例如，一些纯文本文档格式希望标题位于等号或连字符长度的中间。使用字符串格式可以轻易地做到这一点：

*>>> def heading(text):*

*... return '{0:=^40}'.format(text)*

*...*

*>>> heading('Standard Format Specification')*

*'=====Standard Format Specification======'*

*>>> heading('This is a longer heading, beyond 40 characters')*

*'This is a longer heading, beyond 40 characters'*

现在标题将总是至少有40个字符宽，在文本的每一边也总是有至少一个等号，即使它运行得很长。麻烦的是，现在这种做法需要在格式字符串中写入三次等号，一旦填充字符是一个连字符，就会成为维护的麻烦。

解决这个问题的一部分很简单:因为我们显式地对占位符进行了编号，所以我们可以将

填充字符作为参数传入，然后在格式字符串中仅引用两次该参数;然而，仅仅在开头和结尾各一次并不能真正解决问题，因为它没有触及核心问题:如何替换文本的部分参数引用。

为了解决这个问题，格式规范还允许嵌套参数引用。在文本部分的占位符内部，我们可以在为填充字符保留的位置添加另一个占位符;Python会先计算这个，然后再计算另一个。在此过程中，我们还可以控制输出将填充多少字符：

*>>> def heading(text, padding='=', width=40):*

*... return '{1}{0:{1}^{2}}{1}'.format(text, padding, width - 2)*

*...*

*>>> heading('Standard Format Specification')*

*'=====Standard Format Specification======'*

*>>> heading('This is a longer heading, beyond 40 characters')*

*'=This is a longer heading, beyond 40 characters='*

*>>> heading('Standard Format Specification', padding='-', width=60)*

*'---------------Standard Format Specification----------------'*

尽管文档有许多形式，纯文本可能是最常见的一种，因为它不需要任何额外的软件来查看。然而，由于缺少链接或页码，浏览大量文档可能很困难。浏览目录，可以使用行号代替页码，但是正确格式的内容表仍然可能难以维护。

考虑一个典型的目录，其中一个部分的标题是左对齐的，页面或行号是右对齐的，这两个部分之间用一行点号连接起来，以将视线从一个点转到另一个点。从这种格式中添加或删除行很简单，但是每次更改一个节的名称或位置时，你不仅要更改相关信息;还需要更新中间的周期线;这种方法不是很方便。

在这时，字符串格式化就可以派上用场，因为你可以为字符串中的多个值指定对齐和填充选项。有了它，你可以设置一个简单的脚本来自动格式化目录。做到这一点的关键是你要理解到你在做什么。

从表面上看，它的目标似乎就像上面提到的那样:左对齐部分标题，右对齐行号，并在中间放置一行句号。然而，没有完全那样做的方法，所以我们需要以不同的方式来处理它。通过让字符串的每个部分负责填充的一部分就能很容易达到预期的效果：

*>>> '{0:.<50}'.format('Example')*

*'Example...........................................'*

*>>> '{0:.<50}'.format('Longer Example')*

*'Longer Example....................................'*

*>>> '{0:.>10}'.format(20)*

*'........20'*

*>>> '{0:.>10}'.format(1138)*

*'......1138'*

有了这两个部分，只需将它们组合起来就可以在目录中创建完整的行。许多纯文本文档在一行中限制为80个字符，因此我们可以稍微扩展一下，为更长的标题留出一些空间。此外，即使在非常长的文档中,行号的10位数也有点高，因此可以减少行号，从而为标题提供更多的空间。

*>>> def contents\_line(title, line\_number=1):*

*... return '{0:.<70}{1:.>5}'.format(title, line\_number)*

*...*

*>>> contents\_line('Installation', 20)*

*'Installation...........................................................20'*

*>>> contents\_line('Usage', 112)*

*'Usage.................................................................112'*

但是，从长远来看，一次一行地调用这个函数并不是一个现实的解决方案，因此我们将创建一个新的函数来建造一个更有用的数据结构。它不需要很复杂，所以我们将使用一个二元组序列，每个二元组由一个章节标题和相应的行号组成:

*>>> contents = (('Installation', 20), ('Usage', 112))*

*>>> def format\_contents(contents):*

*... for title, line\_number in contents:*

*... yield '{0:.<70}{1:.>5}'.format(title, line\_number)*

*...*

*>>> for line in format\_contents(contents):*

*... print(line)*

*...*

*Installation.............................................................20*

*Usage...................................................................112*

**自定义格式规范**

新格式化系统的真正优点是format（）实际上无法控制上一节所描述的格式化语法。与第四章中描述的特性一样，它将控制权委托给作为参数传入的对象上的类函数。

该类函数format（）可以接受一个包含输入对象所在的格式字符串中格式规范的参数。然而，它无法得到整个括号内的表达式，只能得到冒号后面的部分。你可以通过直接在一个全新的对象实例上使用它来看到这适用于所有的对象。从Python3.3及更高版本开始，这个格式已经更改，请在尝试接下来的示例前前确保你的版本为Python3.3及以上：

>>> object().\_\_format\_\_(")

'=====<object object at 0x0209F158>======'

因此，前一节中描述的标准格式规范选项并不是完成任务的唯一方法。如果你有一个自定义需求，你可以用你正在使用的方法来覆盖它。你可以扩展现有的行为，或者编写一个全新的。

例如，你可以用一个类来表示一个动词，它可以有现在时或过去时。这个动词类可以用一个词来表示每一个时态，然后用在表达式中，形成一个完整的句子：

*>>> class Verb:*

*... def \_\_init\_\_(self, present, past=None):*

*... self.present = present*

*... self.past = past*

*... def \_\_format\_\_(self, tense):*

*... if tense == 'past':*

*... return self.past*

*... else:*

*... return self.present*

*...*

*>>> format = Verb('format', past='formatted')*

*>>> message = 'You can {0:present} strings with {0:past} objects.'*

*>>> message.format(format)*

*'You can format strings with formatted objects.'*

*>>> save = Verb('save', past='saved')*

*>>> message.format(save)*

*'You can save strings with saved objects.'*

在本例中，占位符字符串不会格式化过去时动词，因此它将责任委托给传入的动词。只要输入一次字符串，就可以与不同的动词多次搭配，而不会有所遗漏。

**Python扩展**

**实现Feedparser**

RSS提要（）是针对博客、新闻和媒体等信息发布的提要。也称为提要、web提要或频道，它们可以包括摘要信息或标题。可以说，它们是在信息超载的世界中保持数据更新的首要步骤。Python feedparser库可以处理包括Atom、RDF和RSS在内的格式。从我们已经学过的知识来看，与Beautiful Soup或其他库一起使用，访问这些数据信息将非常方便：

**如何安装**

使用pip安装库：

pip安装feedparser(输入)

确保你处于Windows的升级命令中。Linux和Mac的安装过程相似。安装过程中没有任何错误，你就可以设置为使用Feedparser。

**如何使用**

对于本例，我们将从“Anytime Fitness blog”获取数据。示例代码将提取标题副标题、RSS条目的数量及其名称。它还有更多的功能，你可以将数据写入文件，供其他库日后提取关键数据使用。试试它有多简单：

*#feedparser example*

*import feedparser*

*# main site is: http://blog.anytimefitness.com/*

*c = feedparser.parse('http://feeds.feedburner.com/anytimefitnessofficial')*

*#all elements of the channel are now in container c*

*#print the title and subtitle and list # of elements of the feed*

*print (c['feed']['title'])*

*print (c['feed']['subtitle'])*

*print ("There are this many entries: ", len(c['entries']))*

*print()*

*for item in c['entries']:*

*title = item.title*

*print (title)*

*# try others such as item.summary, item.description, item.link, etc.*

*# write the data to a file for use with BeautifulSoup, etc.*

在本例中c 是一个已命名条目（如标题、副标题等）的容器。并且，这个容器中的时间是整数（len）

**学习须知**

因为字符串在所有类型的编程中都很常见，所以你会发现自己在各种各样的情况都需要它。本章所展示的字符串的特性将帮助你更好地使用它，但是无法为你编写符合需要的技术组合。当你编写代码时，你需要根据你需要的技术来选择你需要对哪些技术进行更深入的了解。

到目前为止，以上章节主要关注如何利用Python的各个功能里执行一些复杂和有用的任务，从而使你的程序更加强大。下一章将向你展示如何确保这些任务在正确地执行。