3

Текст

Практически каждому системному администратору приходится иметь дело с текстом в той или иной форме, например, с файлами журналов, данными приложений, с XML-, HTML- и конфигурационными файлами или с выводом некоторых команд. Обычно для работы вполне хватает таких утилит, как `grep` и `awk`, но иногда для решения сложных задач необходим более элегантный и выразительный инструмент. Когда возникает потребность создать файл с данными, извлеченными из других файлов, часто бывает достаточно перенаправить вывод процесса обработки (здесь опять приходят на ум `grep` и `awk`) в файл. Но иногда складываются ситуации, когда для выполнения задания требуется инструмент с более широкими возможностями.

Как мы уже говорили во «Введении», наш опыт показывает, что язык Python можно рассматривать как более элегантный, выразительный и расширяемый, чем Perl, Bash или другие языки программирования, которые мы использовали в своей практике. Подробное описание причин, почему мы оцениваем Python более высоко, чем Perl или Bash (то же самое относится к `sed` и `awk`), приводится в главе 1. Стандартная библиотека языка Python, особенности языка и встроенные типы представляют собой мощные средства чтения текстовых файлов, манипулирования текстом и извлечения информации из текстовых файлов. Язык Python и стандартная библиотека обладают богатыми и гибкими функциональными возможностями обработки текста с помощью строкового типа, файлового типа и модуля регулярных выражений. Недавнее пополнение стандартной библиотеки – модуль ElementTree – чрезвычайно удобен при работе с данными в формате XML. В этой главе мы покажем, как эффективно использовать стандартную библиотеку и встроенные компоненты при работе с текстовой информацией.

Встроенные компоненты Python и модули

str

Строка – это просто последовательность символов. При любой работе с текстовой информацией вы почти наверняка вынуждены будете работать с ней как со строковым объектом или с последовательностью строковых объектов. Строковый тип, `str`, – это мощное, гибкое средство манипулирования строковыми данными. В этом разделе показывается, как создавать строки и какие операции можно выполнять над ними после их создания.

Создание строк

Обычно строка создается путем заключения некоторого текста в кавычки:

```
In [1]: string1 = 'This is a string'

In [2]: string2 = "This is another string"

In [3]: string3 = '''This is still another string'''

In [4]: string4 = """And one more string"""

In [5]: type(string1), type(string2), type(string3), type(string4)
Out[5]: (<type 'str'>, <type 'str'>, <type 'str'>, <type 'str'>)
```

Апострофы и кавычки, обычные и тройные, обозначают одно и то же: все они создают объект типа `str`. Апострофы и кавычки идентичны по своему действию и являются взаимозаменяемыми. Этим язык Python отличается от командных оболочек UNIX, где апострофы и кавычки не являются взаимозаменяемыми. Например:

```
jmjones@dink:~$ FOO=sometext
jmjones@dink:~$ echo "Here is $FOO"
Here is sometext
jmjones@dink:~$ echo 'Here is $FOO'
Here is $FOO
```

В языке Perl апострофы и кавычки также не могут замещать друг друга при создании строк. Ниже приводится похожий пример на языке Perl.

```
#!/usr/bin/perl

$FOO = "some_text";
print "-- $FOO --\n";
print '-- $FOO --\n';
```

И вот какие результаты дает этот небольшой сценарий на языке Perl:

```
jmjones@dinkgutsy:code$ ./quotes.pl
-- some_text --
-- $FOO --\njmjones@dinkgutsy:code$
```

Это различие отсутствует в языке Python. Право определять различия Python оставляет за программистом. Например, вы можете использовать апострофы, когда внутри строки должны находиться кавычки и вам не хотелось бы экранировать их (символом обратного слеша). Точно так же вы можете использовать кавычки, когда внутри строки должны присутствовать апострофы и вам не хотелось бы экранировать их, как показано в примере 3.1.

Пример 3.1. Кавычки и апострофы в языке Python

```
In [1]: s = "This is a string with 'quotes' in it"

In [2]: s
Out[2]: "This is a string with 'quotes' in it"

In [3]: s = 'This is a string with \'quotes\' in it'

In [4]: s
Out[4]: "This is a string with 'quotes' in it"

In [5]: s = 'This is a string with "quotes" in it'

In [6]: s
Out[6]: 'This is a string with "quotes" in it'

In [7]: s = "This is a string with \"quotes\" in it"

In [8]: s
Out[8]: 'This is a string with "quotes" in it'
```

Обратите внимание, что во 2-й и 4-й строках вывода (`Out [4]` и `Out [8]`) включение в строку экранированных кавычек того же типа, что и окружающие строку, привело при выводе к изменению типа наружных кавычек. (В действительности это приводит отображение строки к «правильному» применению кавычек разных типов.)

Иногда бывает необходимо, чтобы в одной строке объединялось несколько строк. Иногда эту проблему можно решить, вставляя символ `\n` там, где необходимо создать разрыв строки, но это довольно неудобный способ. Другая, более ясная альтернатива заключается в использовании тройных кавычек, которые позволяют определять многострочный текст. В примере 3.2 демонстрируется неудачная попытка использовать апострофы для определения многострочного текста и успешная попытка использовать тройные апострофы.

Пример 3.2. Тройные кавычки

```
In [6]: s = 'this is
------------------------------------------------------------
  File "<ipython console>", line 1
    s = 'this is
                ^
SyntaxError: EOL while scanning single-quoted string
(SyntaxError: обнаружен конец строки при интерпретации строки в апострофах)
```

```
In [7]: s = '''this is a
   ...: multiline string'''

In [8]: s
Out[8]: 'this is a\nmultiline string'
```

Помимо этого существует еще один способ обозначения строк, которые в языке Python называются «сырыми» строками. Сырые строки создаются добавлением символа `r` непосредственно перед открывающей кавычкой. По сути дела, сырые строки отличаются от обычных строк тем, что в сырых строках Python не интерпретирует экранированные последовательности символов, тогда как в обычных строках они интерпретируются. При интерпретации экранированных последовательностей в языке Python соблюдается практически тот же набор правил, который описывается стандартом языка C. Например, в обычных строках последовательность `\t` интерпретируется как символ табуляции, `\n` – как символ новой строки и `\r` – как перевод строки. В табл. 3.1 приводится список экранированных последовательностей в языке Python.

Таблица 3.1. Экранированные последовательности в языке Python

Последовательность	Интерпретируется как
`\newline`	Игнорируется
`\\`	Символ обратного слеша
`\'`	Апостроф
`\"`	Кавычка
`\a`	ASCII-символ звукового сигнала
`\b`	ASCII-символ забоя
`\f`	ASCII-символ перевода формата (страницы)
`\n`	ASCII-символ новой строки
`\N{имя}`	Именованный символ Юникода (только для строк в кодировке Юникод)
`\r`	ASCII-символ возврата каретки
`\t`	ASCII-символ горизонтальной табуляции
`\uxxxx`	Шестнадцатеричный код 16-битового символа (только для строк в кодировке Юникод)
`\Uxxxxxxxx`	Шестнадцатеричный код 32-битового символа (только для строк в кодировке Юникод)
`\v`	ASCII-символ вертикальной табуляции
`\ooo`	Восьмеричный код символа
`\xhh`	Шестнадцатеричный код символа

Об экранированных последовательностях и сырых строках стоит помнить, особенно, когда приходится иметь дело с регулярными выражениями, к которым мы подойдем далее в этой главе. В примере 3.3 демонстрируется использование экранированных последовательностей и неформатированных строк.

Пример 3.3. Экранированные последовательности и сырые строки

```
In [1]: s = '\t'

In [2]: s
Out[2]: '\t'

In [3]: print s


In [4]: s = r'\t'

In [5]: s
Out[5]: '\\t'

In [6]: print s
\t

In [7]: s = '''\t'''

In [8]: s
Out[8]: '\t'

In [9]: print s

In [10]: s = r'''\t'''

In [11]: s
Out[11]: '\\t'

In [12]: print s
\t

In [13]: s = r'\''

In [14]: s
Out[14]: "\\'"

In [15]: print s
\'
```

Когда выполняется интерпретация экранированных последовательностей, `\t` превращается в символ табуляции. Когда интерпретация не выполняется, экранированная последовательность `\t` воспринимается, как строка из двух символов, `\` и `t`. Строки, окруженные кавычками или апострофами, обычными или тройными, подразумевают, что последовательность `\t` будет интерпретироваться как символ табуляции. Если те же самые строки предваряются символом `r`, последовательность `\t` интерпретируется как два символа, `\` и `t`.

Еще один фокус этого примера – различия между `__repr__` и `__str__`. Когда имя переменной вводится в строке приглашения оболочки IPython

и нажимается клавиша Enter, значение переменной отображается вызовом метода __repr__. Когда вводится инструкция print, которой передается имя переменной, и нажимается клавиша Enter, переменная отображается вызовом метода __str__. Инструкция print интерпретирует экранированные последовательности в строке и отображает их соответствующим образом. Подробнее о __repr__ и __str__ рассказывается в главе 2, в разделе «Базовые понятия».

Встроенные методы извлечения строковых данных

Строки в языке Python – это объекты, поэтому они имеют методы, которые могут вызываться для выполнения определенных операций. Однако под «методами» мы подразумеваем не только методы, которыми обладает тип str, но и любые другие способы, позволяющие извлекать данные из объектов типа str. Сюда входят все методы типа str, а также операторы in и not in, приведенные в примере, следующем ниже.

С технической точки зрения, операторы проверки условия in и not in вызывают метод __contains__() объекта str. За дополнительной информацией о том, как работают эти операторы, обращайтесь к приложению. Операторы in и not in могут использоваться для проверки, является ли некоторая строка частью другой строки, как показано в примере 3.4.

Пример 3.4. Операторы in и not in

```
In [1]: import subprocess

In [2]: res = subprocess.Popen(['uname', '-sv'], stdout=subprocess.PIPE)

In [3]: uname = res.stdout.read().strip()

In [4]: uname
Out[4]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'

In [5]: 'Linux' in uname
Out[5]: True

In [6]: 'Darwin' in uname
Out[6]: False

In [7]: 'Linux' not in uname
Out[7]: False

In [8]: 'Darwin' not in uname
Out[8]: True
```

Если строка string2 содержит строку string1, то выражение string1 in string2 вернет значение True, в противном случае – значение False. Поэтому проверка вхождения строки "Linux" в строку uname в нашем случае дает в результате значение True, а проверка вхождения строки "Darwin" в строку uname дает значение False. Применение оператора not in мы привели «для комплекта».

Иногда бывает достаточно узнать, что некоторая строка является подстрокой другой строки. А иногда требуется узнать, в какой позиции находится искомая подстрока. Выяснить это можно с помощью методов `find()` и `index()`, как показано в примере 3.5.

Пример 3.5. Методы find( ) и index( )

```
In [9]: uname.index('Linux')
Out[9]: 0

In [10]: uname.find('Linux')
Out[10]: 0

In [11]: uname.index('Darwin')
---------------------------------------------------------------------------
<type 'exceptions.ValueError'>            Traceback (most recent call last)

/home/jmjones/code/<ipython console> in <module>()

<type 'exceptions.ValueError'>: substring not found

In [12]: uname.find('Darwin')
Out[12]: -1
```

Если строка `string1` присутствует в строке `string2` (как в данном примере), метод `string2.find(string1)` вернет индекс первого символа `string1` в строке `string2`, в противном случае он вернет `-1`. (Не беспокойтесь, к индексам мы перейдем через мгновение). Точно так же, если строка `string1` присутствует в строке `string2`, метод `string2.index(string1)` вернет индекс первого символа `string1` в строке `string2`, в противном случае он возбудит исключение `ValueError`. В данном примере метод `find()` обнаружил подстроку `"Linux"` в начале строки, поэтому он вернул значение `0`. Однако метод `find()` не смог обнаружить подстроку `"Darwin"` в этой строке, поэтому он вернул значение `-1`. Когда в операционной системе Linux была выполнена попытка отыскать подстроку `"Linux"` с помощью метода `index()`, был получен тот же результат, что и в случае применения метода `find()`. Но при попытке отыскать подстроку `"Darwin"` метод `index()` возбудил исключение `ValueError`, показывая, что не смог отыскать эту подстроку.

Итак, что можно делать с этими числовыми «индексами»? Зачем они нам нужны? Строки интерпретируются как списки символов. «Индекс», который возвращается методами `find()` и `index()`, просто показывает, начиная с какого символа в большей строке было обнаружено совпадение, как показано в примере 3.6.

Пример 3.6. Срез строки

```
In [13]: smp_index = uname.index('SMP')

In [14]: smp_index

Out[14]: 9

In [15]: uname[smp_index:]
```

```
Out[15]: 'SMP Tue Feb 12 02:46:46 UTC 2008'

In [16]: uname[:smp_index]

Out[16]: 'Linux #1 '

In [17]: uname

Out[17]: 'Linux #1 SMP Tue Feb 12 02:46:46 UTC 2008'
```

Мы оказались в состоянии увидеть все символы, начиная с символа, индекс которого был получен в результате поиска подстроки `"SMP"`, и до конца строки, воспользовавшись синтаксической конструкцией извлечения среза `string[index:]`. Мы также смогли увидеть все символы от начала строки `uname` до индекса, который был получен в результате поиска подстроки `"SMP"`, применив синтаксическую конструкцию извлечения среза `string[:index]`. Все различия между этими двумя конструкциями заключаются в местоположении символа двоеточия (`:`) относительно индекса.

Цель примеров на извлечение среза строки и применения операторов `in` и `not in` состоит в том, чтобы показать вам, что строки являются последовательностями и поэтому обладают теми же особенностями, что и другие последовательности, такие как списки. Более полно последовательности обсуждаются в разделе «Sequence Operations» в главе 4 книги «Python in a Nutshell» (издательство O'Reilly) Алекса Мартелли (Alex Martelli) (этот раздел доступен в Интернете на сайте издательства: *http://safari.oreilly.com/0596100469/pythonian-CHP-4-SECT-6*).

Еще два строковых метода, `startswith()` и `endswith()`, как следует из их названий, помогут определить, «начинается» ли или «заканчивается» ли строка определенной подстрокой, как показано в примере 3.7.

Пример 3.7. Методы startswith( ) и endswith( )

```
In [1]: some_string = "Raymond Luxury-Yacht"

In [2]: some_string.startswith("Raymond")
Out[2]: True

In [3]: some_string.startswith("Throatwarbler")
Out[3]: False

In [4]: some_string.endswith("Luxury-Yacht")
Out[4]: True

In [5]: some_string.endswith("Mangrove")
Out[5]: False
```

Как видите, интерпретатор Python возвращает информацию, которая говорит о том, что строка «Raymond Luxury-Yacht» начинается с подстроки «Raymond» и заканчивается подстрокой «Luxury-Yacht». Она не начинается с подстроки «Throatwarbler» и не заканчивается подстрокой «Mangrove». Достаточно просто те же результаты можно получить

с помощью операции извлечения среза, но такой подход к решению выглядит менее наглядно и может показаться несколько утомительным в реализации, как показано в примере 3.8:

Пример 3.8. Имитация методов startswith( ) и endswith( )

```
In [6]: some_string[:len("Raymond")] == "Raymond"
Out[6]: True

In [7]: some_string[:len("Throatwarbler")] == "Throatwarbler"
Out[7]: False

In [8]: some_string[-len("Luxury-Yacht"):] == "Luxury-Yacht"
Out[8]: True

In [9]: some_string[-len("Mangrove"):] == "Mangrove"
Out[9]: False
```

Операция извлечения среза создает и возвращает новый строковый объект, а не изменяет саму строку. Если операции извлечения среза часто используются в сценарии, они могут оказывать существенное влияние на потребление памяти и на производительность. Даже если заметного влияния на производительность не ощущается, тем не менее, лучше воздержаться от использования операций извлечения среза в случаях, когда достаточно применения методов `startswith()` и `endswith()`.

Мы сумели убедиться, что первые символы в строке `some_string`, число которых равно длине строки «Raymond», соответствуют строке «Raymond». Другими словами, мы сумели убедиться, что строка `some_string` начинается с подстроки «Raymond», без использования метода `startswith()`. Точно так же мы смогли убедиться, что строка заканчивается подстрокой «Luxury-Yacht».

Методы `lstrip()`, `rstrip()` и `strip()` без аргументов удаляют ведущие, заключительные, и ведущие и заключительные пробельные символы, соответственно. В качестве таких пробельных символов можно назвать символы табуляции, пробелы, символы возврата каретки и новой строки. Метод `lstrip()` без аргументов удаляет любые пробельные символы, которые находятся в начале строки, и возвращает новую строку. Метод `rstrip()` без аргументов удаляет любые пробельные символы, которые находятся в конце строки, и возвращает новую строку. Метод `strip()` без аргументов удаляет любые пробельные символы, которые находятся в начале и в конце строки, и возвращает новую строку, как показано в примере 3.9.

Все три метода из семейства `strip()` не изменяют саму строку, а создают и возвращают новый строковый объект. Возможно, вы никогда не будете испытывать проблем с таким поведением методов, но вы должны знать о нем.

Пример 3.9. Методы lstrip( ), rstrip( ) и strip( )

```
In [1]: spacious_string = "\n\t Some Non-Spacious Text\n \t\r"

In [2]: spacious_string
Out[2]: '\n\t Some Non-Spacious Text\n \t\r'

In [3]: print spacious_string

         Some Non-Spacious Text

In [4]: spacious_string.lstrip()
Out[4]: 'Some Non-Spacious Text\n \t\r'

In [5]: print spacious_string.lstrip()
Some Non-Spacious Text


In [6]: spacious_string.rstrip()
Out[6]: '\n\t Some Non-Spacious Text'

In [7]: print spacious_string.rstrip()

         Some Non-Spacious Text

In [8]: spacious_string.strip()
Out[8]: 'Some Non-Spacious Text'

In [9]: print spacious_string.strip()
Some Non-Spacious Text
```

Все три метода, `lstrip()`, `rstrip()` и `strip()`, могут принимать единственный необязательный аргумент: строку символов, которые следует удалить из соответствующего места строки. Это означает, что методы семейства `strip()` не просто удаляют пробельные символы – они могут удалять любые символы, какие вы укажете:

```
In [1]: xml_tag = "<some_tag>"

In [2]: xml_tag.lstrip("<")

Out[2]: 'some_tag>'

In [3]: xml_tag.lstrip(">")

Out[3]: '<some_tag>'

In [4]: xml_tag.rstrip(">")

Out[4]: '<some_tag'

In [5]: xml_tag.rstrip("<")

Out[5]: '<some_tag>'
```

Здесь мы удалили из тега XML левую и правую угловые скобки, по одной за раз. А как быть, если нам потребуется удалить обе скобки одновременно? Сделать это можно следующим способом:

```
In [6]: xml_tag.strip("<").strip(">")

Out[6]: 'some_tag'
```

Методы семейства `strip()` возвращают строку, поэтому мы можем вызывать другие строковые операции прямо вслед за вызовом метода `strip()`. В этом примере мы объединили вызовы методов `strip()` в цепочку. Первый вызов метода `strip()` удаляет начальный символ (левую угловую скобку) и возвращает строку, а второй метод `strip()` удаляет завершающий символ (правую угловую скобку) и возвращает строку `"some_tag"`. Однако существует более простой способ:

```
In [7]: xml_tag.strip("<>")

Out[7]: 'some_tag'
```

Возможно, вы подумали, что методы семейства `strip()` удаляют точное вхождение указанной подстроки, но в действительности удаляются любые последовательные вхождения указанных символов с соответствующей стороны строки. В этом последнем примере методу `strip()` было предписано удалить `"<>"`. Это не означает точное соответствие подстроке `"<>"` и не означает, что должны быть удалены вхождения этих двух символов, следующих друг за другом именно в таком порядке, – это означает, что должны быть удалены символы `"<"` или `">"`, находящиеся в начале или в конце строки.

Ниже приводится, возможно, более понятный пример:

```
In [8]: gt_lt_str = "<><><>gt lt str<><><>"

In [9]: gt_lt_str.strip("<>")

Out[9]: 'gt lt str'

In [10]: gt_lt_str.strip("><")

Out[10]: 'gt lt str'
```

Здесь мы удалили все вхождения символов `"<"` или `">"` с обоих концов строки. Таким способом мы можем ликвидировать простые символы и пробелы.

Следует заметить, что такой прием может работать несколько не так, как вы ожидаете, например:

```
In [11]: foo_str = "<fooooooo>blah<foo>"

In [12]: foo_str.strip("<foo>")

Out[12]: 'blah'
```

У вас могло бы сложиться мнение, что метод `strip()` в этом примере удалит символы справа, но не слева. Однако он обнаружит и удалит любые последовательные вхождения символов `"<"`, `"f"`, `"o"` и `">"`. Это не ошибка, мы не пропустили второй символ `"o"`. Вот еще один, заключительный пример использования метода `strip()`, который прояснит это утверждение:

```
In [13]: foo_str.strip(">><of")

Out[13]: 'blah'
```

Здесь удаляются символы ">", "<", "f", "o", хотя они следуют не в этом порядке.

Методы upper() и lower() удобно использовать, когда необходимо выполнить сравнение двух строк без учета регистра символов. Метод upper() возвращает строку со всеми символами из оригинальной строки в верхнем регистре. Метод lower() возвращает строку со всеми символами из оригинальной строки в нижнем регистре, как показано в примере 3.10.

Пример 3.10. Методы upper( ) и lower( )

```
In [1]: mixed_case_string = "VOrpal BUnny"

In [2]: mixed_case_string == "vorpal bunny"
Out[2]: False

In [3]: mixed_case_string.lower() == "vorpal bunny"
Out[3]: True

In [4]: mixed_case_string == "VORPAL BUNNY"
Out[4]: False

In [5]: mixed_case_string.upper() == "VORPAL BUNNY"
Out[5]: True

In [6]: mixed_case_string.upper()
Out[6]: 'VORPAL BUNNY'

In [7]: mixed_case_string.lower()
Out[7]: 'vorpal bunny'
```

Если вам необходимо извлечь часть строки, ограниченной какими-либо символами-разделителями, метод split() предоставит вам эту возможность, как показано в примере 3.11.

Пример 3.11. Метод split( )

```
In [1]: comma_delim_string = "pos1,pos2,pos3"

In [2]: pipe_delim_string = "pipepos1|pipepos2|pipepos3"

In [3]: comma_delim_string.split(',')
Out[3]: ['pos1', 'pos2', 'pos3']

In [4]: pipe_delim_string.split('|')
Out[4]: ['pipepos1', 'pipepos2', 'pipepos3']
```

Методу split() передается строка-разделитель, по которому необходимо разбить строку на подстроки. Часто это единственный символ, такой как запятая или вертикальная черта, но это может быть строка, содержащая более одного символа. В данном примере мы разбили

строку comma_delim_string по запятым, а строку pipe_delim_string — по символу вертикальной черты (|), передавая символ запятой или вертикальной черты методу split(). Возвращаемым значением метода является список строк, каждая из которых представляет собой группу последовательных символов, находящихся между разделителями. Когда в качестве разделителя необходимо использовать не единственный символ, а некоторую строку, метод split() справится и с этим. К моменту написания этих строк в языке Python отсутствовал символьный тип, поэтому хотя в примерах метод split() получал единственный символ, он рассматривался методом как строка. Поэтому, когда методу split() передается несколько символов, он обработает и их, как показано в примере 3.12.

Пример 3.12. Строка-разделитель в методе split( )

```
In [1]: multi_delim_string = "pos1XXXpos2XXXpos3"

In [2]: multi_delim_string.split("XXX")
Out[2]: ['pos1', 'pos2', 'pos3']

In [3]: multi_delim_string.split("XX")
Out[3]: ['pos1', 'Xpos2', 'Xpos3']

In [4]: multi_delim_string.split("X")
Out[4]: ['pos1', '', '', 'pos2', '', '', 'pos3']
```

Обратите внимание, что сначала мы использовали в качестве разделителя строку "XXX" для разделения строки multi_delim_string. Как и ожидалось, в результате был получен список ['pos1', 'pos2', 'pos3']. Затем, мы использовали в качестве разделителя строку "XX" и метод split() вернул ['pos1', 'Xpos2', 'Xpos3']. Здесь метод split() выбрал все символы, находящиеся между соседними разделителями "XX". Подстрока "pos1" начинается с начала строки и простирается до первого разделителя "XX"; подстрока "Xpos2" располагается сразу за первым вхождением разделителя "XX" и простирается до второго его вхождения; и подстрока "Xpos3" располагается сразу за вторым вхождением "XX" и простирается до конца строки. Последний вызов метода split() получает в качестве разделителя единственный символ "X". Обратите внимание, что позициям между соседними символами "X" соответствуют пустые строки ("") в результирующем списке. Это означает, что между соседними символами "X" ничего нет.

Но как быть, если необходимо разбить строку только по первым n вхождениям указанного разделителя? Для этого методу split() следует передать второй аргумент, с именем max_split. Когда во втором аргументе max_split методу split() передается целочисленное значение, он выполнит только указанное число разбиений исходной строки:

```
In [1]: two_field_string = "8675309,This is a freeform, plain text, string"

In [2]: two_field_string.split(',', 1)
Out[2]: ['8675309', 'This is a freeform, plain text, string']
```

Здесь мы разбиваем строку по запятым и предписываем методу `split()` выполнить только одно разбиение. Несмотря на то, что в строке присутствует несколько запятых, строка была разбита только один раз.

Если необходимо разбить строку по пробелам, например, чтобы извлечь из текста отдельные слова, эту задачу легко можно решить вызовом метода `split()` без аргументов:

```
In [1]: prosaic_string = "Insert your clever little piece of text here."

In [2]: prosaic_string.split()
Out[2]: ['Insert', 'your', 'clever', 'little', 'piece', 'of', 'text', 'here.']
```

Когда метод `split()` не получает никаких аргументов, по умолчанию он выполняет разбиение строки по пробельным символам.

Чаще всего вы будете получать именно те результаты, которые ожидали получить. Однако в случае многострочного текста результат может получиться неожиданным для вас. Часто при работе с многострочным текстом бывает необходимо выполнять его обработку по одной строке за раз. Но вы можете с удивлением обнаружить, что программа разбивает такой текст на отдельные слова:

```
In [1]: multiline_string = """This
   ...: is
   ...: a multiline
   ...: piece of
   ...: text"""

In [2]: multiline_string.split()
Out[2]: ['This', 'is', 'a', 'multiline', 'piece', 'of', 'text']
```

Для таких случаев лучше подходит метод `splitlines()`:

```
In [3]: lines = multiline_string.splitlines()

In [4]: lines

Out[4]: ['This', 'is', 'a multiline', 'piece of', 'text']
```

Метод `splitlines()` возвращает список всех строк из многострочного текста и сохраняет группы «слов». После этого можно выполнить итерации по отдельным строкам текста и извлечь отдельные слова:

```
In [5]: for line in lines:
   ...:     print "START LINE::"
   ...:     print line.split()
   ...:     print "::END LINE"
   ...:
START LINE::
['This']
::END LINE
START LINE::
['is']
::END LINE
```

```
START LINE::
['a', 'multiline']
::END LINE
START LINE::
['piece', 'of']
::END LINE
START LINE::
['text']
::END LINE
```

Иногда бывает необходимо не анализировать строку или извлекать из нее информацию, а объединить в строку уже имеющиеся данные. В этом случае вам на помощь придет метод `join()`:

```
In [1]: some_list = ['one', 'two', 'three', 'four']

In [2]: ','.join(some_list)
Out[2]: 'one,two,three,four'

In [3]: ', '.join(some_list)
Out[3]: 'one, two, three, four'

In [4]: '\t'.join(some_list)
Out[4]: 'one\ttwo\tthree\tfour'

In [5]: ''.join(some_list)
Out[5]: 'onetwothreefour'
```

Учитывая, что исходные данные хранятся в виде списка, мы можем объединить строки `'one'`, `'two'`, `'three'` и `'four'` несколькими способами. Мы объединяем элементы списка `some_list` с помощью запятой, запятой и пробела, символа табуляции и пустой строки. Метод `join()` – это строковый метод, поэтому вызов его в качестве метода литерала, такого как `','`, является корректным. Метод `join()` принимает в качестве аргумента последовательность строк и объединяет их в одну строку так, чтобы элементы последовательности располагались в исходном порядке и отделялись строкой, для которой вызывается метод `join()`.

Мы должны предупредить вас об особенностях поведения метода `join()` и об аргументе, который он ожидает получить. Обратите внимание: метод `join()` ожидает получить последовательность строк. А что произойдет, если ему передать последовательность целых чисел? Взгляните!

```
In [1]: some_list = range(10)

In [2]: some_list
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: ",".join(some_list)
---------------------------------------------------------------------------
exceptions.TypeError                      Traceback (most recent call last)

/Users/jmjones/<ipython console>
```

```
TypeError: sequence item 0: expected string, int found
(TypeError: элемент 0 последовательности: ожидается строка, обнаружено целое)
```

Диагностическая информация, приложенная к исключению, возбужденному методом `join()`, достаточно ясно объясняет происшедшее, но так как это довольно распространенная ошибка, в этом стоит разобраться. Вы легко сможете избежать этой ловушки с помощью простого генератора списков. Ниже мы прибегли к помощи генератора списков, чтобы преобразовать все элементы списка `some_list`, которые содержат целые числа, в строки:

```
In [4]: ",".join([str(i) for i in some_list])
Out[4]: '0,1,2,3,4,5,6,7,8,9'
```

Или можно использовать выражение-генератор:

```
In [5]: ",".join(str(i) for i in some_list)

Out[5]: '0,1,2,3,4,5,6,7,8,9'
```

За дополнительной информацией об использовании генераторов списков обращайтесь к разделу «Control Flow Statements» в главе 4 книги «Python in a Nutshell» (этот раздел доступен в Интернете, на сайте издательства: *http://safari.oreilly.com/0596100469/pythonian-CHP-4-SECT-10*).

Последний метод, используемый для создания и изменения текстовых строк, – это метод `replace()`. Метод `replace()` принимает два аргумента: строку, которую требуется заменить, и строку замены, соответственно. Ниже приводится простой пример использования метода `replace()`:

```
In [1]: replacable_string = "trancendental hibernational nation"

In [2]: replacable_string.replace("nation", "natty")
Out[2]: 'trancendental hibernattyal natty'
```

Обратите внимание, что метод `replace()` никак не проверяет, является замещаемая строка частью слова или отдельным словом. Поэтому метод `replace()` может использоваться только в случаях, когда просто требуется заменить определенную последовательность символов другой определенной последовательностью символов.

Иногда требуется более тонкое управление операцией замены, когда вариант простой замены одной последовательности символов на другую не подходит. В таких случаях обычно бывает необходимо иметь возможность определить шаблон последовательности символов, которую требуется найти и заменить. Применение шаблонов также может помочь с поиском требуемого текста для последующего извлечения из него данных. В тех случаях, когда предпочтительнее использовать шаблоны, вам помогут регулярные выражения. Регулярные выражения мы рассмотрим далее.

Так же, как операция извлечения среза и метод `strip()`, метод `replace()` не изменяет существующую строку, а создает новую.

Строки Юникода

До сих пор во всех примерах работы со строками, которые мы видели, использовались исключительно строковые объекты встроенного типа `str`, но в языке Python существует еще один строковый тип, с которым вам предстоит познакомиться: строки Юникода. Любые символы, которые выводятся на экран дисплея, внутри компьютера представлены числами. До появления кодировки Юникод существовало множество разнообразных наборов отображения числовых кодов в символы в зависимости от языка и платформы. Юникод – это стандарт, обеспечивающий единое отображение числовых кодов в символы, независимое от языка, платформы или даже программы, выполняющей обработку текста. В этом разделе мы рассмотрим понятие Юникода и способы работы с этой кодировкой, имеющиеся в языке Python. Подробное описание Юникода вы найдете в превосходном учебнике Э. М. Качлинга (A. M. Kuchling) по адресу: *http://www.amk.ca/python/howto/unicode*.

Создание строк Юникода выглядит ничуть не сложнее, чем создание обычных строк:

```
In [1]: unicode_string = u'this is a unicode string'

In [2]: unicode_string
Out[2]: u'this is a unicode string'

In [3]: print unicode_string
this is a unicode string
```

Или можно воспользоваться встроенной функцией `unicode()`:

```
In [4]: unicode('this is a unicode string')
Out[4]: u'this is a unicode string'
```

На первый взгляд, в этом нет ничего примечательного, особенно если учесть, что здесь мы имеем дело с символами одного языка. Но как быть, когда приходится работать с символами из нескольких языков? Здесь вам на помощь придет Юникод. Чтобы внутри строки Юникода создать символ с определенным числовым кодом, можно воспользоваться нотацией `\uXXXX` или `\uXXXXXXXX`. Например, ниже приводится строка Юникода, содержащая символы латиницы, греческого алфавита и кириллицы:

```
In [1]: unicode_string = u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042F'

In [2]: unicode_string
Out[2]: u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f'
```

Интерпретатор генерирует строку (`str`) в зависимости от используемой кодировки. В версии Python, которая поставляется вместе с компьютерами Mac, если попытаться вывести строку из предыдущего примера с помощью инструкции `print`, будет получено сообщение об ошибке:

```
In [3]: print unicode_string
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)

/Users/jmjones/<ipython console> in <module>()

UnicodeEncodeError: 'ascii' codec can't encode characters in position 4-6:
  ordinal not in range(128)
(UnicodeEncodeError: кодек 'ascii' не в состоянии кодировать символы
в позиции 4-6: числовые значения находятся вне диапазона range(128))
```

Мы должны определить другой кодек, который знает, как обрабатывать все символы в строке:

```
In [4]: print unicode_string.encode('utf-8')
abc_ΠΣΩ_ДФЯ
```

Здесь мы выполнили кодирование строки, содержащей символы латиницы, греческого алфавита и кириллицы, в кодировку UTF-8, которая наиболее часто используется для кодирования данных Юникода.

Строки Юникода обладают теми же возможностями, такими как возможность выполнения проверки с помощью оператора `in`, и методами, что и обычные строки, о которых мы уже говорили:

```
In [5]: u'abc' in unicode_string
Out[5]: True

In [6]: u'foo' in unicode_string
Out[6]: False

In [7]: unicode_string.split()
Out[7]: [u'abc_\u03a0\u03a3\u03a9_\u0414\u0424\u042f']

In [8]: unicode_string.
unicode_string.__add__           unicode_string.expandtabs
unicode_string.__class__         unicode_string.find
unicode_string.__contains__      unicode_string.index
unicode_string.__delattr__       unicode_string.isalnum
unicode_string.__doc__           unicode_string.isalpha
unicode_string.__eq__            unicode_string.isdecimal
unicode_string.__ge__            unicode_string.isdigit
unicode_string.__getattribute__  unicode_string.islower
unicode_string.__getitem__       unicode_string.isnumeric
unicode_string.__getnewargs__    unicode_string.isspace
unicode_string.__getslice__      unicode_string.istitle
unicode_string.__gt__            unicode_string.isupper
unicode_string.__hash__          unicode_string.join
unicode_string.__init__          unicode_string.ljust
unicode_string.__le__            unicode_string.lower
```

```
unicode_string.__len__          unicode_string.lstrip
unicode_string.__lt__           unicode_string.partition
unicode_string.__mod__          unicode_string.replace
unicode_string.__mul__          unicode_string.rfind
unicode_string.__ne__           unicode_string.rindex
unicode_string.__new__          unicode_string.rjust
unicode_string.__reduce__       unicode_string.rpartition
unicode_string.__reduce_ex__    unicode_string.rsplit
unicode_string.__repr__         unicode_string.rstrip
unicode_string.__rmod__         unicode_string.split
unicode_string.__rmul__         unicode_string.splitlines
unicode_string.__setattr__      unicode_string.startswith
unicode_string.__str__          unicode_string.strip
unicode_string.capitalize       unicode_string.swapcase
unicode_string.center           unicode_string.title
unicode_string.count            unicode_string.translate
unicode_string.decode           unicode_string.upper
unicode_string.encode           unicode_string.zfill
unicode_string.endswith
```

Возможно, строки Юникода не потребуются вам немедленно. Но важно знать об их существовании, если вы собираетесь продолжать программировать на языке Python.

re

Раз поставка языка Python комплектуется в соответствии с принципом «батарейки включены», можно было бы ожидать, что в состав стандартной библиотеки будут включены модули для работы с регулярными выражениями. Так оно и есть. Акцент в этом разделе сделан на использовании в языке Python регулярных выражений, а не на подробностях их синтаксиса. Поэтому, если вы не знакомы с регулярными выражениями, рекомендуем вам приобрести книгу «Mastering Regular Expressions» (O'Reilly) Джеффри Е. Ф. Фридла (Jeffrey E. F. Friedl) (доступна также в Интернете на сайте издательства по адресу: *http://safari.oreilly.com/0596528124*).[1] Далее мы предполагаем, что вы достаточно уверенно оперируете регулярными выражениями, в противном случае рекомендуем держать книгу Фридла под рукой.

Если вы знакомы с языком Perl, то, возможно, вы уже использовали регулярные выражения с оператором `=~`. В языке Python поддержка регулярных выражений реализована на уровне библиотеки, а не на уровне синтаксических особенностей языка. Поэтому для работы с регулярными выражениями необходимо импортировать модуль `re`. Ни-

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-плюс, 2008. В Интернете можно ознакомиться с 5-й главой этой книги по адресу *http://www.books.ru/chapter?id= 592346&num=1*. – *Прим. перев.*

же приводится простой пример создания и использования регулярного выражения, как показано в примере 3.13.

Пример 3.13. Простой пример использования регулярного выражения

```
In [1]: import re

In [2]: re_string = "{{(.*?)}}"

In [3]: some_string = "this is a string with {{words}} embedded in\
   ...: {{curly brackets}} to show an {{example}} of {{regular expressions}}"

In [4]: for match in re.findall(re_string, some_string):
   ...:     print "MATCH->", match
   ...:
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

Первое, что мы сделали в этом примере, – импортировали модуль `re`. Как вы уже наверняка поняли, имя `re` происходит от «regular expressions» (регулярные выражения). Затем мы создали строку `re_string`, которая будет играть роль шаблона для поиска. Этому шаблону будут соответствовать две открывающие фигурные скобки (`{{`), вслед за которыми может следовать текст, завершающийся двумя закрывающими фигурными скобками (`}}`). Затем мы создали строку `some_string`, которая содержит группы слов, окруженные фигурными скобками. И в конце мы выполнили обход результатов поиска в строке `some_string` по шаблону `re_string`, полученных от функции `findall()` из модуля `re`. Как видите, пример вывел строки `words`, `curly brackets`, `example` и `regular expressions`, которые представляют все группы слов, заключенные в двойные фигурные скобки.

В языке Python существует два способа работы с регулярными выражениями. Первый заключается в непосредственном использовании функций из модуля `re`, как в предыдущем примере. Второй способ состоит в том, чтобы создать объект скомпилированного регулярного выражения и затем использовать методы этого объекта.

Итак, что же такое скомпилированное регулярное выражение? Это просто объект, созданный вызовом функции `re.compile()`, которой передается шаблон. Этот объект, созданный за счет передачи шаблона функции `re.compile()`, содержит множество методов для работы с регулярным выражением. Между скомпилированным и нескомпилированным регулярными выражениями имеются два основных отличия. Во-первых, вместо ссылки на шаблон регулярного выражения `"{{.*?}}"` создается объект скомпилированного выражения на основе шаблона. Во-вторых, вместо функции `findall()` из модуля `re` следует вызывать метод `findall()` объекта скомпилированного выражения.

За дополнительной информацией о всех функциях, имеющихся в модуле re, обращайтесь к разделу «Module Contents» в справочнике «Python Library Reference», *http://docs.python.org/lib/node46.html*. За дополнительной информацией об объектах скомпилированных регулярных выражений обращайтесь к разделу «Regular Expression Objects» в справочнике «Python Library Reference», *http://docs.python.org/lib/re-objects.html*.

В примере 3.14 представлена реализация предыдущего примера с двойными фигурными скобками, выполненная на основе использования объекта скомпилированного регулярного выражения.

Пример 3.14. Простое регулярное выражение, скомпилированный шаблон

```
In [1]: import re

In [2]: re_obj = re.compile("{{(.*?)}}")

In [3]: some_string = "this is a string with {{words}} embedded in\
   ...: {{curly brackets}} to show an {{example}} of {{regular expressions}}"

In [4]: for match in re_obj.findall(some_string):
   ...:     print "MATCH->", match
   ...:
MATCH-> words
MATCH-> curly brackets
MATCH-> example
MATCH-> regular expressions
```

Выбор метода работы с регулярными выражениями в языке Python зависит отчасти от личных предпочтений и от самого регулярного выражения. Следует заметить, что метод, основанный на использовании функций модуля re, уступает в производительности методу, основанному на использовании объектов скомпилированных регулярных выражений. Проблема производительности особенно остро может вставать, например, когда регулярное выражение применяется в цикле к каждой строке текстового файла, содержащего сотни и тысячи строк. В примерах ниже представлены реализации простых сценариев, использующих скомпилированые и нескомпилированные регулярные выражения, которые применяются к файлу, содержащему 500 000 строк текста. Если воспользоваться специальной функцией timeit, можно увидеть разницу в производительности между этими двумя сценариями. Смотрите пример 3.15.

Пример 3.15. Тест производительности нескомпилированного регулярного выражения

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

import re

def run_re():
    pattern = 'pDq'
```

```
    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re.search(pattern, line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES::', lines
    print 'MATCHES::', match_count
```

Функция `timeit` выполняет программный код несколько раз и возвращает время самого лучшего варианта. Ниже показаны результаты запуска утилиты `timeit` для этого сценария в оболочке IPython:

```
In [1]: import re_loop_nocompile

In [2]: timeit -n 5 re_loop_nocompile.run_re()
5 loops, best of 3: 1.93 s per loop
```

В этом примере функция `run_re()` была вызвана 5 раз, и было вычислено среднее из 3 самых лучших показателей, которое составило 1,93 секунды. Специальная функция `timeit` выполняется с исследуемым программным кодом несколько раз, чтобы уменьшить погрешность, вызванную влиянием других процессов, исполняющихся в системе.

Ниже приводятся результаты измерения времени выполнения того же самого программного кода с помощью утилиты `time` операционной системы UNIX:

```
jmjones@dink:~/code$ time python re_loop_nocompile.py
LINES:: 500000 MATCHES:: 242

real    0m2.113s
user    0m1.888s
sys     0m0.163s
```

Пример 3.16 – это тот же пример с регулярным выражением за исключением того, что мы используем `re.compile()` для создания объекта скомпилированного шаблона.

Пример 3.16. Тест производительности скомпилированного регулярного выражения

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

import re

def run_re():
    pattern = 'pDq'
    re_obj = re.compile(pattern)
```

```
    infile = open('large_re_file.txt', 'r')
    match_count = 0
    lines = 0
    for line in infile:
        match = re_obj.search(line)
        if match:
            match_count += 1
        lines += 1
    return (lines, match_count)

if __name__ == "__main__":
    lines, match_count = run_re()
    print 'LINES::', lines
    print 'MATCHES::', match_count
```

Испытания с помощью специальной функции `timeit` в оболочке IPython дали следующие результаты:

```
In [3]: import re_loop_compile

In [4]: timeit -n 5 re_loop_compile.run_re()
5 loops, best of 3: 860 ms per loop
```

А испытания того же самого сценария с помощью утилиты `time` операционной системы UNIX дали следующие результаты:

```
jmjones@dink:~/code$ time python
re_loop_compile.py LINES:: 500000 MATCHES:: 242

real    0m0.996s
user    0m0.836s
sys     0m0.154s
```

Версия со скомпилированным регулярным выражением одержала чистую победу. Время работы этой версии оказалось в два раза меньше как по данным утилиты UNIX `time`, так и по данным функции `timeit` оболочки IPython. Поэтому мы настоятельно рекомендуем взять в привычку использовать объекты скомпилированных регулярных выражений.

Как уже говорилось ранее в этой главе, для определения строк, в которых не интерпретируются экранированные последовательности, можно использовать сырые (неформатированные) строки. В примере 3.17 показано применение неформатированных строк для использования в регулярных выражениях.

Пример 3.17. Неформатированные строки и регулярные выражения

```
In [1]: import re

In [2]: raw_pattern = r'\b[a-z]+\b'

In [3]: non_raw_pattern = '\b[a-z]+\b'

In [4]: some_string = 'a few little words'
```

```
In [5]: re.findall(raw_pattern, some_string)

Out[5]: ['a', 'few', 'little', 'words']

In [6]: re.findall(non_raw_pattern, some_string)

Out[6]: []
```

Шаблонный символ `\b` в регулярных выражениях соответствует границе слова. То есть, как в случае применения сырой строки, так и в случае применения обычной строки, мы предполагаем отыскать отдельные слова, состоящие из символов нижнего регистра. Обратите внимание, что при использовании `raw_pattern` были обнаружены отдельные слова в `some_string`, а при использовании `non_raw_pattern` вообще ничего не было найдено. В строке `raw_pattern` комбинация `\b` интерпретируется как два отдельных символа, в то время как в строке `non_raw_pattern` она интерпретируется как символ забоя (backspace). В результате функция `findall()` сумела отыскать отдельные слова с помощью неформатированной строки шаблона. Однако при использовании шаблона в виде обычной строки функция `findall()` не отыскала ни одного символа забоя (backspace).

Чтобы с помощью шаблона `non_raw_pattern` можно было отыскать соответствие в строке, необходимо окружить требуемое слово символами `\b`, как показано ниже:

```
In [7]: some_other_string = 'a few \blittle\b words'

In [8]: re.findall(non_raw_pattern, some_other_string)
Out[8]: ['\x08little\x08']
```

Обратите внимание на шестнадцатеричную форму записи символа `"\x08"` в соответствии, найденном функцией `findall()`. Эта шестнадцатеричная форма записи соответствует символам забоя (backspace), которые были добавлены с помощью экранированной последовательности `\b`.

Как видите, неформатированные строки могут пригодиться, когда предполагается использовать специальные последовательности, такие как `"\b"`, обозначающую границу слова, `"\d"`, обозначающую цифру, или `"\w"`, обозначающую алфавитно-цифровой символ. Полный перечень специальных последовательностей, начинающихся с символа обратного слеша, вы найдете в разделе «Regular Expression Syntax» в справочнике «Python Library Reference», *http://docs.python.org/lib/re-syntax.html*.

Примеры с 3.14 по 3.17 были очень простыми. В них во всех использовались регулярные выражения и различные методы, применяемые к ним. Иногда такого ограниченного использования регулярных выражений вполне достаточно. Иногда бывает необходимо нечто более мощное, чем имеется в библиотеке регулярных выражений.

К основным методам (или функциям) регулярных выражений, которые используются наиболее часто, относятся `findall()`, `finditer()`, `match()` и `search()`. Вам также могут потребоваться методы `split()` и `sub()`, но, вероятно, не так часто, как другие методы.

Метод `findall()` отыскивает все вхождения указанного шаблона в строке. Если метод `findall()` найдет соответствия шаблону, тип возвращаемой структуры данных будет зависеть от наличия групп в шаблоне.

Краткое напоминание: группировка в регулярных выражениях позволяет указывать текст внутри регулярного выражения, который следует извлечь из результата. За дополнительной информацией обращайтесь к разделу «Common Metacharacters and Fields» в книге Фридла (Friedl) «Mastering Regular Expressions»[1] или в Интернете по адресу: *http://safari.oreilly.com/0596528124/regex3-CHP-3-SECT-5?imagepage=137*.

Если в регулярном выражении отсутствуют группы, а совпадение найдено, тогда `findall()` вернет список строк. Например:

```
In [1]: import re

In [2]: re_obj = re.compile(r'\bt.*?e\b')

In [3]: re_obj.findall("time tame tune tint tire")
Out[3]: ['time', 'tame', 'tune', 'tint tire']
```

В этом шаблоне отсутствуют группы, поэтому `findall()` возвращает список строк. Здесь можно наблюдать интересный побочный эффект – последний элемент списка содержит два слова, tint и tire. Используемое здесь регулярное выражение соответствует словам, начинающимся с символа «t» и заканчивающимся символом «e». Но часть выражения `.*?` соответствует любым символам, включая пробелы. Метод `findall()` отыскал все, что предполагалось. Он отыскал слово, начинающееся с символа «t» (tint), и продолжил просмотр строки, пока не обнаружил слово, завершающееся символом «e» (tire). Поэтому соответствие «tint tire» вполне согласуется с шаблоном. Чтобы исключить пробел, можно было бы использовать регулярное выражение `r'\bt\w*e\b'`:

```
In [4]: re_obj = re.compile(r'\bt\w*e\b')

In [5]: re_obj.findall("time tame tune tint tire")
Out[5]: ['time', 'tame', 'tune', 'tire']
```

Второй тип структуры данных, который может быть получен, – это список кортежей. Если группы присутствуют в выражении и было найдено совпадение, то `findall()` вернет список кортежей. Подобный шаблон и строка показаны в примере 3.18.

---

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-плюс, 2008. Глава 3. Раздел «Стандартные метасимволы и возможности». – *Прим. перев.*

Пример 3.18. Простая группировка и метод findall( )

```
In [1]: import re

In [2]: re_obj = re.compile(r"""(A\W+\b(big|small)\b\W+\b
   ...: (brown|purple)\b\W+\b(cow|dog)\b\W+\b(ran|jumped)\b\W+\b
   ...: (to|down)\b\W+\b(the)\b\W+\b(street|moon).*?\.)""",
   ...: re.VERBOSE)

In [3]: re_obj.findall('A big brown dog ran down the street. \
   ...: A small purple cow jumped to the moon.')

Out[3]:
[('A big brown dog ran down the street.',
  'big',
  'brown',
  'dog',
  'ran',
  'down',
  'the',
  'street'),
 ('A small purple cow jumped to the moon.',
  'small',
  'purple',
  'cow',
  'jumped',
  'to',
  'the',
  'moon')]
```

Несмотря на свою простоту, этот пример демонстрирует ряд важных моментов. Во-первых, обратите внимание, что этот простой шаблон нелепо длинен и содержит массу неалфавитно-цифровых символов, от которых начинает рябить в глазах, если смотреть слишком долго. Это обычная вещь для многих регулярных выражений. Затем, обратите внимание, что шаблон содержит явные вложенные группы. Объемлющая группа будет соответствовать любому тексту, начинающемуся с символа «A» и заканчивающемуся точкой. Символы между начальным символом «A» и завершающей точкой образуют вложенные группы, которые должны соответствовать словам «big» или «small», «brown» или «purple» и так далее. Далее, возвращаемое значение метода `findall()` представляет собой список кортежей. Элементами этих кортежей являются группы, которые были определены в регулярном выражении. Первый элемент кортежа – все предложение, потому что оно соответствует наибольшей, объемлющей группе. Последующие элементы кортежа соответствуют каждой из подгрупп. Наконец, обратите внимание на последний аргумент в вызове метода `re.compile()` – `re.VERBOSE`. Это позволило нам записать регулярное выражение в многострочном режиме, то есть мы смогли расположить регулярное выражение в нескольких строках, не оказывая влияния на поиск соответствий. Пробел, оказавшийся за пределами группировки, был проигнори-

рован. Хотя мы и не продемонстрировали здесь такую возможность, тем не менее, многострочный режим позволяет вставлять комментарии в конец каждой строки регулярного выражения, чтобы описать, что делает та или иная его часть. Одна из основных сложностей, связанных с регулярными выражениями, состоит в том, что описание шаблона часто бывает очень длинным и трудным для чтения. Цель `re.VERBOSE` состоит в том, чтобы упростить написание регулярных выражений, следовательно, это ценный инструмент, облегчающий сопровождение программного кода, содержащего регулярные выражения.

Метод `finditer()` является разновидностью метода `findall()`. Вместо того чтобы возвращать список кортежей, как это делает метод `findall()`, `finditer()` возвращает итератор, как это следует из имени метода. Каждый элемент итератора – это объект найденного совпадения, который мы обсудим далее в этой главе. Пример 3.19 реализует тот же простой пример, только в нем вместо метода `findall()` используется метод `finditer()`.

Пример 3.19. Пример использования метода finditer( )

```
In [4]: re_iter = re_obj.finditer('A big brown dog ran down the street. \
   ...: A small purple cow jumped to the moon.')

In [5]: re_iter

Out[5]: <callable-iterator object at 0xa17ad0>

In [6]: for item in re_iter:
   ...:     print item
   ...:     print item.groups()
   ...:
<_sre.SRE_Match object at 0x9ff858>
('A big brown dog ran down the street.', 'big', 'brown', 'dog', 'ran',
    'down', 'the', 'street')
<_sre.SRE_Match object at 0x9ff940>
('A small purple cow jumped to the moon.', 'small', 'purple', 'cow',
    'jumped', 'to', 'the', 'moon')
```

Если прежде вы никогда не сталкивались с итераторами, вы можете представлять их себе как списки, которые создаются в тот момент, когда они необходимы. Один из недостатков такого определения состоит в том, что вы не можете обратиться к определенному элементу итератора по его индексу, как, например, к элементу списка `some_list[3]`. Вследствие этого ограничения вы не можете получить срез итератора, как, например, в случае списка `some_list[2:6]`. Тем не менее, независимо от этих ограничений итераторы представляют собой легкое и мощное средство, особенно когда необходимо выполнить итерации через некоторую последовательность, потому что при этом последовательность не загружается целиком в память, а элементы ее возвращаются по требованию. Это позволяет итераторам занимать меньший объем

памяти, чем соответствующие им списки. Кроме того, доступ к элементам последовательности производится быстрее.

Еще одно преимущество метода `finditer()` перед `findall()` состоит в том, что каждый элемент, возвращаемый методом `finditer()`, – это объект `match`, а не простой список строк или список кортежей, соответствующих найденному тексту.

Методы `match()` и `search()` обеспечивают похожие функциональные возможности. Оба метода применяют регулярное выражение к строке; оба указывают, с какой позиции начать и в какой закончить поиск по шаблону; оба возвращают объект `match` для первого найденного соответствия заданному шаблону. Разница между этими двумя методами состоит в том, что метод `match()` пытается отыскать совпадение только от начала строки или от указанного места в строке, не переходя в другие позиции в строке, а метод `search()` будет пытаться отыскать соответствие шаблону в любом месте строки или между начальной и конечной позицией, которые вы укажете, как показано в примере 3.20.

Пример 3.20. Сравнение методов match( ) и search( )

```
In [1]: import re

In [2]: re_obj = re.compile('FOO')

In [3]: search_string = ' FOO'

In [4]: re_obj.search(search_string)

Out[4]: <_sre.SRE_Match object at 0xa22f38>

In [5]: re_obj.match(search_string)

In [6]:
```

Даже при том, что в строке `search_string` имеется соответствие шаблону, поиск по которому производит метод `match()`, тем не менее, поиск завершается неудачей, потому что подстрока в `search_string`, соответствующая шаблону, находится не в начале строки. Метод `search()`, напротив, нашел соответствие и вернул объект `match`.

Методы `search()` и `match()` принимают параметры, определяющие начальную и конечную позицию поиска в строке, как показано в примере 3.21.

Пример 3.21. Параметры начала и конца поиска в методах search( ) и match( )

```
In [6]: re_obj.search(search_string, pos=1)

Out[6]: <_sre.SRE_Match object at 0xabe030>

In [7]: re_obj.match(search_string, pos=1)

Out[7]: <_sre.SRE_Match object at 0xabe098>
```

```
In [8]: re_obj.search(search_string, pos=1, endpos=3)

In [9]: re_obj.match(search_string, pos=1, endpos=3)

In [10]:
```

Параметр `pos` – это индекс, определяющий место в строке, откуда должен начинаться поиск по шаблону. В данном примере передача параметра `pos` методу `search()` не повлияла на результат, но передача параметра `pos` методу `match()`привела к тому, что он нашел соответствие шаблону, хотя без параметра `pos` соответствие обнаружить не удалось. Установка параметра `endpos` в значение 3 привела к тому, что оба метода – и `match()`, и `search()` не нашли соответствие, потому что соответствие шаблону включает символ в третьей позиции.

Методы `findall()` и `finditer()` отвечают на вопрос: «чему соответствует мой шаблон?», а главный вопрос, на который отвечают методы `search()` и `match()`: «имеется ли соответствие моему шаблону?». Методы `search()` и `match()` отвечают также на вопрос: «каково первое соответствие моему шаблону?», но часто единственное, что требуется узнать, это: «имеется ли соответствие моему шаблону?». Например, предположим, что необходимо написать сценарий, который должен читать строки из файла журнала и обертывать каждую строку в теги HTML, чтобы обеспечить удобочитаемое отображение. При этом хотелось бы, чтобы все строки, содержащие текст «ERROR», отображались красным цветом, для чего можно было бы выполнить цикл по всем строкам в файле, проверить их с помощью регулярного выражения и, если метод `search()` обнаруживает текст «ERROR», можно было бы определить такой формат строки, чтобы она отображалась красным цветом.

Методы `search()` и `match()` удобны не только тем, что они определяют наличие соответствия, но и тем, что они возвращают объект `match`. Объекты `match` содержат различные методы извлечения данных, которые могут пригодиться при обходе полученных результатов. Особый интерес представляют такие методы объекта `match`, как `start()`, `end()`, `span()`, `groups()` и `groupdict()`.

Методы `start()`, `end()` и `span()` определяют позиции в строке поиска, где совпадение с шаблоном начинается и где заканчивается. Метод `start()` возвращает целое число, определяющее позицию в строке начала найденного соответствия. Метод `end()` возвращает целое число, определяющее позицию в строке конца найденного соответствия. А метод `span()` возвращает кортеж, содержащий позицию начала и конца совпадения.

Метод `groups()` возвращает кортеж совпадения, каждый элемент которого соответствует группе, имеющейся в шаблоне. Этот кортеж напоминает кортежи в списке, возвращаемом методом `findall()`. Метод `groupdict()` возвращает словарь именованных групп, ключи которого соответствуют именам групп, присутствующих непосредственно в регулярном выражении, например: `(?P<group_name>pattern)`.

Подводя итоги, можно сказать – чтобы эффективно использовать регулярные выражения, следует взять в привычку использовать объекты скомпилированных регулярных выражений. Используйте методы `findall()` и `finditer()`, когда необходимо получить части текста, соответствующие шаблону. Запомните, что метод `finditer()` обладает более высокой гибкостью, чем `findall()`, потому что возвращает итератор по объектам `match`. Более подробный обзор библиотеки регулярных выражений вы найдете в главе 9 книги «Python in a Nutshell» Алекса Мартелли (Alex Martelli) (O'Reilly). Чтобы познакомиться с регулярными выражениями в действии, обращайтесь к книге «Data Crunching» Грега Уилсона (Greg Wilson) (The Pragmatic Bookshelf).

Работа с конфигурационным файлом Apache

Теперь, когда вы получили представление о работе с регулярными выражениями в языке Python, попробуем поработать с конфигурационным файлом веб-сервера Apache:

```
NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
    DocumentRoot /var/www/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error.log
    LogLevel warn
    CustomLog /var/log/apache2/access.log combined
    ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
    DocumentRoot /var/www2/
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    ErrorLog /var/log/apache2/error2.log
    LogLevel warn
    CustomLog /var/log/apache2/access2.log combined
    ServerSignature On
</VirtualHost>
```

Это слегка измененный конфигурационный файл Apache в Ubuntu. Мы создали именованные виртуальные хосты для некоторых своих нужд. Мы также добавили в файл */etc/hosts* следующую строку:

```
127.0.0.1     local2
```

Она позволяет указать броузеру, что серверу с именем `local2` соответствует IP-адрес 127.0.01, то есть локальный компьютер. И в чем же здесь смысл? Если в броузере ввести адрес *http://local2*, он передаст серверу

указанное имя в заголовке HTTP. Ниже приводится HTTP-запрос, направленный серверу `local2`:

```
GET / HTTP/1.1
Host: local2
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.8.1.13)
Gecko/20080325 Ubuntu/7.10 (gutsy) Firefox/2.0.0.13
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
If-Modified-Since: Tue, 15 Apr 2008 17:25:24 GMT
If-None-Match: "ac5ea-53-44aecaf804900"
Cache-Control: max-age=0
```

Обратите внимание, что запрос начинается с заголовка `Host:`. Когда веб-сервер Apache получит такой запрос, он направит его виртуальному хосту с именем `local2`.

Теперь все, что нам предстоит сделать, – это написать сценарий, который анализирует конфигурационный файл веб-сервера Apache, такой, как показано выше, отыскивает раздел `VirtualHost` и замещает значение параметра `DocumentRoot` в этом разделе. Сам сценарий приводится ниже:

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

from cStringIO import StringIO
import re

vhost_start = re.compile(r'<VirtualHost\s+(.*?)>')
vhost_end = re.compile(r'</VirtualHost')
docroot_re = re.compile(r'(DocumentRoot\s+)(\S+)')

def replace_docroot(conf_string, vhost, new_docroot):
    '''отыскивает в файле httpd.conf строки DocumentRoot, соответствующие
       указанному vhost, и замещает их новыми строками new_docroot
    '''
    conf_file = StringIO(conf_string)
    in_vhost = False
    curr_vhost = None
    for line in conf_file:
        vhost_start_match = vhost_start.search(line)
        if vhost_start_match:
            curr_vhost = vhost_start_match.groups()[0]
            in_vhost = True
        if in_vhost and (curr_vhost == vhost):
            docroot_match = docroot_re.search(line)
            if docroot_match:
                sub_line = docroot_re.sub(r'\1%s' % new_docroot, line)
                line = sub_line
```

```
        vhost_end_match = vhost_end.search(line)
        if vhost_end_match:
            in_vhost = False
        yield line

if __name__ == '__main__':
    import sys
    conf_file = sys.argv[1]
    vhost = sys.argv[2]
    docroot = sys.argv[3]
    conf_string = open(conf_file).read()
    for line in replace_docroot(conf_string, vhost, docroot):
        print line,
```

Этот сценарий сначала создает три объекта скомпилированных регулярных выражений: один соответствует открывающему тегу `VirtualHost`, один – закрывающему тегу `VirtualHost` и один – строке с параметром `DocumentRoot`. Мы также создали функцию, которая выполняет эту утомительную работу. Функция называется `replace_docroot` и принимает в качестве аргументов тело конфигурационного файла в виде строки, имя раздела `VirtualHost`, который требуется отыскать, и значение параметра `DocumentRoot`, которое требуется назначить для данного виртуального хоста. Функция устанавливает признак состояния, который указывает, находится ли текущая анализируемая строка в разделе `VirtualHost`. Кроме того, сохраняется имя текущего виртуального хоста. При анализе строк в разделе `VirtualHost` эта функция пытается отыскать строку с параметром `DocumentRoot` и изменяет его значение. Поскольку функция `replace_docroot()` выполняет итерации по каждой строке в конфигурационном файле, она возвращает либо неизмененную исходную строку, либо измененную строку с параметром `DocumentRoot`.

Мы создали простой интерфейс командной строки к этой функции. В нем не предусматривается использование ничего особенного, такого как функция `optparse`, и не выполняется проверка на количество входных аргументов, но он работает. Теперь попробуем применить этот сценарий к конфигурационному файлу веб-сервера Apache, представленному выше, и изменим настройки `VirtualHost local2:80` так, чтобы он использовал каталог */tmp* в качестве корневого каталога документов. Предусмотренный нами интерфейс командной строки просто выводит строки, возвращаемые функцией `replace_docroot()`, а не изменяет сам файл:

```
jmjones@dinkgutsy:code$ python apache_conf_docroot_replace.py
  /etc/apache2/sites-available/psa
  local2:80 /tmp
NameVirtualHost 127.0.0.1:80
<VirtualHost localhost:80>
        DocumentRoot /var/www/
        <Directory />
                Options FollowSymLinks
```

```
                AllowOverride None
        </Directory>
        ErrorLog /var/log/apache2/error.log
        LogLevel warn
        CustomLog /var/log/apache2/access.log combined
        ServerSignature On
</VirtualHost>
<VirtualHost local2:80>
        DocumentRoot /tmp
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
        ErrorLog /var/log/apache2/error2.log
        LogLevel warn
        CustomLog /var/log/apache2/access2.log combined
        ServerSignature On
</VirtualHost>
```

Единственная строка, которая изменилась, – это строка с параметром `DocumentRoot` в разделе `VirtualHost local2:80`. Ниже приводятся различия, полученные после того, как вывод сценария был перенаправлен в файл:

```
jmjones@dinkgutsy:code$ diff apache_conf.diff /etc/apache2/sites-available/psa
20c20
< DocumentRoot /tmp
---
> DocumentRoot /var/www2/
```

Изменение значения параметра `DocumentRoot` в конфигурационном файле веб-сервера Apache – это достаточно простая задача, но когда это приходится делать достаточно часто или когда имеется множество виртуальных хостов, которые приходится изменять, тогда есть смысл написать сценарий, подобный тому, что был показан выше. Не менее просто можно было бы изменить сценарий так, чтобы он комментировал требуемый раздел `VirtualHost`, изменял значение параметра `LogLevel` или изменял имя файла журнала для указанного виртуального хоста.

Работа с файлами

Овладение приемами работы с файлами является ключом к обработке текстовых данных. Зачастую текст, который требуется обработать, находится в текстовом файле, например, в файле журнала, в конфигурационном файле или в файле с данными приложения. Нередко результаты анализа данных требуется сохранить в виде файла отчета или просто записать их в текстовый файл для последующего изучения. К счастью, в языке Python имеется простой в использовании тип объектов с именем `file`, который в состоянии помочь выполнить все необходимые действия с файлами.