Charming Python: Text processing in Python

Tips for beginners

David Mertz
President
Gnosis Software, Inc.

01 September 2000

Along with several other popular scripting languages, Python is an excellent tool for scanning and manipulating textual data. This article summarizes Python's text processing facilities for the programmer new to Python. The article explains some general concepts of regular expressions and offers advice on when to use (or not use) regular expressions while processing text.

What is Python?

Python is a freely available, very high-level, interpreted language developed by Guido van Rossum. It combines a clear syntax with powerful (but optional) object-oriented semantics. Python is widely available and highly portable.

Strings -- immutable sequences

As in most higher-level programming languages, variable length strings are a basic type in Python. Python allocates memory to hold strings (or other values) "behind the scenes" where the programmer does not need to give much thought to it. Python also has several string-handling features that do not exist in other high-level languages.

In Python, strings are "immutable sequences." A program can refer to elements or subsequences of strings just as with any sequence, although strings, like tuples, cannot be modified "in place." Python refers to subsequences with the flexible "slice" operation, whose format resembles a range of rows or columns in a spreadsheet. The following interactive session illustrates the use of strings and slices:

Strings and slices

```
>>> s = "mary had a little lamb"
>>> s[0]          # index is zero-based
'm'
>>> s[3] = 'x'    # changing element in-place fails
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> s[11:18]      # 'slice' a subsequence
'little '
>>> s[:4]         # empty slice-begin assumes zero
'mary'
>>> s[4]          # index 4 is not included in slice [:4]
' '
>>> s[5:-5]       # can use "from end" index with negatives
'had a little'
>>> s[:5]+s[5:]   # slice-begin & slice-end are complimentary
'mary had a little lamb'
```

Another powerful string operation is the simple in keyword. This offers two intuitive and useful constructs:

The in keyword

```
>>> s = "mary had a little lamb"
>>> for c in s[11:18]: print c,  # print each char in slice
...
l i t t l e
>>> if'x'in s: print'got x'   # test for char occurrence
...
>>> if'y'in s: print'got y'   # test for char occurrence
...
got y
```

There are several ways to compose string literals in Python. You can use single or double quotes as long as opening and closing tokens match, and there are other variations on quoting that are frequently useful. If your string contains line breaks or embedded quotes, triple-quoting is an easy way to define a string, as in the following example:

The use of triple quotes

```
>>> s2 = """Mary had a little lamb
... its fleece was white as snow
... and everywhere that Mary went
... the lamb was sure to go"""
>>> print s2
Mary had a little lamb
its fleece was white as snow
and everywhere that Mary went
the lamb was sure to go
```

Both single-quoted or triple-quoted strings may be preceded by the letter "r" to indicate that regular expression special characters should not be interpreted by Python. For example:

Using "r-strings"

```
>>> s3 = "this \n and \n that"
>>> print s3
this
 and
 that
>>> s4 = r"this \n and \n that"
>>> print s4
this \n and \n that
```

In "r-strings," the backslash that might otherwise compose an escaped character is treated as a regular backslash. This is further explained in a later discussion of regular expressions.

Files and string variables

.readline() and .readlines() are very similar. They are both used in constructs like:

```
fh = open('c:\\autoexec.bat')
for line in fh.readlines():
    print line
```

The difference between .readline() and .readlines() is that the latter, like .read() , reads an entire file at once. .readlines() automatically parses the file's contents into a list of lines that can be processed by Python's for ... in ... construct. .readline() , on the other hand, reads just a single line at a time, and is generally much slower than .readlines() . .readline() should be used only if there might not be enough memory to read the entire file at once.

If you are using a standard module that deals with files, you can turn a string into a "virtual file" by using the cStringIO module (the StringIO module can be used instead in cases where subclassing the module is required, but a beginner is unlikely to need to do this). For example:

The cStringIO module

```
>>> import cStringIO
>>> fh = cStringIO.StringIO()
>>> fh.write("mary had a little lamb")
>>> fh.getvalue()
'mary had a little lamb'
>>> fh.seek(5)
>>> fh.write('ATE')
>>> fh.getvalue()
'mary ATE a little lamb'
```

Keep in mind, however, that unlike a real file, a cStringIO "virtual file" is not persistent. It will be gone when the program finishes if you do not take steps to save it (such as writing it to a real file, or using the shelve module or a database).

Standard module: string

A general rule-of-thumb is that if you *can* do a task using the string module, that is the *right* way to do it. In contrast to re (regular expressions), string functions are generally much faster, and in most cases they are easier to understand and maintain. Third-party Python modules, including some fast ones written in C, are available for specialized tasks, but portability and familiarity still suggest

sticking with string whenever possible. There are exceptions, but not as many as you might think if you are used to other languages.

The string module contains several types of things, such as functions, methods, and classes; it also contains strings of common constants. For example:

Example 1 of string use

```
>>> import string
>>> string.whitespace
'\011\012\013\014\015 '
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Although you could write these constants by hand, the string versions more or less ensure that your constants will be correct for the national language and platform your Python script is running on.

string also includes functions that transform strings in common ways (which you can combine to form several uncommon transformations). For example:

Example 2 of string use

```
>>> import string
>>> s = "mary had a little lamb"
>>> string.capwords(s)
'Mary Had A Little Lamb'
>>> string.replace(s, 'little', 'ferocious')
'mary had a ferocious lamb'
```

There are many other transformations that are not specifically illustrated here; you can find the details in a Python manual.

You can also use string functions to report string attributes such as length or position of a sub-string, for example:

Example 3 of string use

```
>>> import string
>>> s = "mary had a little lamb"
>>> string.find(s, 'had')
5
>>> string.count(s, 'a')
4
```

Finally, string provides a very Pythonic oddball. The pair .split() and .join() provide a quick way to convert between strings and tuples, something you will find remarkably useful. Usage is straightforward:

Example 4 of string use

```
>>> import string
>>> s = "mary had a little lamb"
>>> L = string.split(s)
>>> L
['mary', 'had', 'a', 'little', 'lamb']
>>> string.join(L, "-")
'mary-had-a-little-lamb'
```

Of course, in real life, you would be likely to do something else with a list besides .join() it right back together (probably something involving our familiar for ... in ... construct).

Standard module: re

The re module obsoletes the regex and regsub modules that you may see used in some older Python code. While there are still a few, limited advantages to regex , they are minor and it is not worth using in new code. The obsolete modules are likely to be dropped from future Python releases, and 1.6 will probably have an improved interface-compatible re module. So stick with re for regular expressions.

Regular expressions are complicated. One could write a book on this a topic, and in fact, a number of people have! This article will try to capture the "gestalt" of regular expressions, and let the reader take it from there.

A regular expression is a concise way of describing a pattern that might occur in a text. Do certain characters occur? In a particular order? Are sub-patterns repeated a given number of times? Do other sub-patterns exclude a match? Conceptually, this is not unlike the way you would intuitively describe a pattern in a natural language. The trick is encoding this description in the compact syntax of regular expressions.

When approaching a regular expression, treat it as its own programming problem, even though only one or two lines of code may be involved; those lines effectively form a small program.

Start with the smallest pieces. At its lowest level, any regular expression will involve matching particular "character classes." The simplest character class is a single character, which is just included in the pattern as a literal. Frequently, you will want to match a class of characters. You can indicate a class by surrounding it in square braces; within the braces, you can have both a set of characters and character ranges indicated with a dash. You can also use a number of named character classes that will be accurate for your platform and national language. Some examples:

Character classes

```
>>> import re
>>> s = "mary had a little lamb"
>>> if re.search("m", s): print"Match!"      # char literal
...
Match!
>>> if re.search("[@A-Z]", s): print"Match!"# char class
...     # match either at-sign or capital letter
...
>>> if re.search("\d", s): print"Match!"     # digits class
...
```

You can think of character classes as the "atoms" of regular expressions, and you will usually want to group those atoms into "molecules." You can do this with a combination of *grouping* and *repetition*. Grouping is indicated by parentheses: any sub-expression contained in parentheses is treated as if it were atomic for the purpose of further grouping or repetition. Repetition is indicated by one of several operators: "*" means "zero or more"; "+" means "one or more"; "?" means "zero or one". For example, look at the expression:

```
ABC([d-w]*\d\d?)+XYZ
```

For a string to match this expression, it must contain something that starts with "ABC" and ends with "XYZ" -- but what must it have in the middle? The middle sub-expression is ([d-w]*\d\d?) followed by the "one or more" operator. So the middle of the string must consist of one (or two, or one thousand) things matching the sub-expression in parentheses. The string "ABCXYZ" will not match, because it does not have the requisite stuff in the middle.

Just what is this inner sub-expression? It starts with *zero or more* letters in the range d-w . It is important to notice that zero letters is a valid match, which may be counterintuitive if you use the English word "some" to describe it. Next the string must have *exactly one* digit; then *zero or one* additional digits. (The first digit character class has no repetition operator, so it simply occurs once. The second digit character class has the "?" operator.) In short, this translates to "either one or two digits." Some strings matched by the regular expression are:

```
ABC1234567890XYZ
ABCd12e1f37g3XYZ
ABC1XYZ
```

These are a few expressions *not* matched by the regular expression (try to think through why these do not match):

```
ABC123456789dXYZ
ABCdefghijklmnopqrstuvwXYZ
ABcd12e1f37g3XYZ
ABC12345%67890XYZ
ABCD12E1F37G3XYZ
```

It takes a bit of practice to get used to creating and understanding regular expressions. Once you have mastered regular expressions, however, you will have a great deal of expressive power at your disposal. That said, it is often easy to jump into using a regular expression to solve a problem that could actually be solved using simpler (and faster) tools, such as string.

Resources

- Read the previous installments of *Charming Python*.
- *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly and Associates, 1997) is a standard and definitive reference on regular expressions.
- For a good introduction to some early text-processing tools that are still widely used and extremely useful, look at *Sed & Awk* by Dale Dougherty and Arnold Robbins (O'Reilly and Associates, 1997).
- Read about mxTextTools, fast text-manipulation tools for Python.
- More on regular expressions:
  - A regular expressions how-to document from Python.org

About the author

David Mertz

Since conceptions without intuitions are empty, and intuitions without conceptions, blind, David Mertz wants a cast sculpture of Milton for his office. Start planning for his birthday. David may be reached at mertz@gnosis.cx; his life pored over at http://gnosis.cx/dW/. Suggestions and recommendations on this, past, or future, columns are welcomed.