Programmer's Niche

Little Bits of String

by Thomas Lumley

Character strings occupy an unusual position in R. They are treated as atomic types — little bits of string — by all the subscripting functions. On the other hand, in most cases where the programmer really is treating strings as atomic objects it would be better to use factors instead.

The decision not to treat strings as vectors of characters makes sense when you consider what sorts of operations we do on strings. The vectorised operations in R mostly deal with operating on the same entry in multiple vectors. With two strings, the 'same character' is more likely to be defined relative to patterns in the string rather than by counting characters from the beginning. The most useful operations on strings are not elementwise operations on their characters, but pattern matching and replacement operations. In the Unix world these pattern matching operations are described by *regular expressions*. R provides two implementations of regular expressions, which allow quite complex text manipulation, though for efficient processing of very large quantities of text a specialised tool such as Perl is better. I will discuss only some features of one of the R implementations. A more complete description of regular expressions in R is given by `help(regex)` and the references it cites.

A regular expression is a small program that matches one or more strings. For example, the regular expression `a regular expression` matches the single string "a regular expression". R provides functions `grep`, `sub`, `gsub`, `regexpr`, and `strsplit` that take a regular expression and a string and look for any of the possible outputs of the regular expression in the string. This can be useful even with very simple regular expressions, for example, `apropos("file")` returns all the currently defined objects that have the string `"file"` in their names.

As the example shows, most characters in a regular expression just match themselves as output. The power of regular expressions comes from the special characters that do more than this. The first two are `^` and `$`, which represent the beginning and end of the string respectively. The regular expression `^print` matches "`print`" at the beginning of the string, which is found in the strings `"print.data.frame"` and `"printer.type"` but not in `"dev.print"`. In simpler times the function `methods`, which lists the methods available for a given generic function, could just use a regular expression of this sort to find all functions whose name began with the name of the generic; things are much more complicated now.

In addition to representing a single character, we can represent a class of characters. At the extreme, the character `.` matches any character, but more restricted patterns are often more useful. For example, the regular expression `[[:alpha:]]` matches any single uppercase or lowercase letter, `[[:digit:]]` matches any digit, and `[[:space:]]` matches any of the white-space characters. Other square-bracket expressions specify other character classes. You may see `[A-Z]` for specifying ranges of characters (between A and Z in this example), but this should be used with care (for example, in Danish the letters Æ, Ø, and Å are not between A and Z).

The special characters present a problem when you need an actual `.` or `$`, say, in a string. To prevent their being interpreted as special they must be preceded by a backslash and, given the familiar problems with C strings, this means typing a double backslash. To match ".sgml" at the end of a string (identifying S help files) we need the regular expression `\\.sgml$` (as a convenience, the R functions that accept regular expressions mostly accept an argument `fixed=TRUE` to specify that all special characters should be interpreted as ordinary characters).

The real power of regular expressions comes from specifying repetitions of characters or sets of characters. The qualifiers `?`, `+`, and `*` specify that the previous character should be repeated at most once, at least once, or any number of times. For example, `^[[:space:]]*` matches any amount of empty space at the start of a string, so

```
sub("^[[:space:]]*", "", string)
```

will remove any leading whitespace characters from the strings (that is, replace it with nothing). Parentheses create groups that can be repeated together, so that `(".+",[[:space:]]*)*` matches any number of terms in quotation marks, separated by commas and optionally by space. Even trickier things are possible with parentheses:

```
 ([[:alpha:]]+)[[:space:]]\\1
```

matches repeated words. The backreference `\\1` means 'the string that the first set of parentheses matched'. We could remove repeated words from a string with

```
> gsub("([[:alpha:]]+)[[:space:]]*\\1", "\\1",
             "the the end is is nigh")
[1] "the end is nigh"
```

The Sweave tools (Leisch, 2002) provide a nice example of the integration of regular expressions into R programs. The underlying R code can process input files in multiple formats, with the necessary descriptions of the formats given by regular expressions (*e.g.* `tools::SweaveSyntaxNoweb`[1])

[1]in future versions of R this will be `utils::SweaveSyntaxNoweb`

Despite this power, the main use I make of regular expressions is in simple script processing. For example, in porting a package written for S-PLUS recently I wrote a function to find all the SGML documentation files in the package, strip the version control comments from the beginning of each file, and feed them to `R CMD Sd2Rd` to make Rd files. This could have been done in Perl or probably with a simple shell script, but it seemed easier just to use R.

An interesting exercise for the reader: Creating a namespace for an existing package that uses S3 methods requires registering the methods in the `NAMESPACE` file. That is, for a function such as `print.coxph`, the `NAMESPACE` file should contain `S3method(print, coxph)`. How would you write a function that looked for methods for a list of common generic functions and created the necessary `S3method` calls? You would probably need to use the functions either `strsplit` and `paste` or `regexpr` and `substr`, and would need to remember that the generic name need not be a single word (`is.na.Surv` springs to mind). Some hand editing would still be required, for example to determine whether `t.test.formula` was a method for `t` or `t.test` or neither.

References

Leisch, F. (2002) Sweave, Part I: Mixing R and LaTeX. *R News* 2(3): 28–31.

Thomas Lumley
Biostatistics, University of Washington
tlumley@u.washington.edu

Recent Events

R at the Statistics Canada Symposium

R was one of the software packages demonstrated at the 2003 Statistics Canada Symposium, an annual event organized by Statistics Canada and presented near Ottawa, Ontario. The event is attended largely by the methodologists (statisticians) who work here and at other national statistical agencies, such as the US Census Bureau. This year's symposium was the 20th edition of the conference. The four day event had two days of software demonstrations with different packages being shown each day. Out of the 10 different packages being demonstrated, 7 were developed at Statistics Canada for very specific tasks.

Although the software demonstrations were hidden in a room out of sight from the main proceedings, there were still several curious visitors to the R demo computer. We showed some basic demos of R at work in sampling and in simulation problems, emphasizing both the simplicity of the code and the results that can be produced. We also used the built-in graphics demo, which impressed many people. As most people working for Statistics Canada use SAS, many of our discussions were about why one would use R instead of SAS (besides the fact that R is free). We had many discussions on programming structure, creating plots, dealing with large amounts of data and general ease of use. All our visitors, when shown the software, seemed to like it and were not adverse to the idea of using it instead of SAS. In fact, they couldn't understand why we were not already using it more widely.

Krisztina Filep
Social Survey Methods Division
Statistics Canada
Ottawa, Ontario, Canada
Krisztina.Filep@statcan.ca