Introduction to String Matching and Modification in R Using Regular Expressions

Svetlana Eden

March 6, 2007

1 Do We Really Need Them ?

Working with statistical data in **R** involves a great deal of text data or ***character strings*** processing, including adjusting exported variable names to the **R** variable name format, changing categorical variable levels, processing text data for using in LaTeX. Such tasks can be performed by using functions for string search (or ***matching***) and modification.

R provides several such functions. The most commonly used ones are **grep(), gsub(), strsplit()**. When using them, it is important to know that some of their arguments are interpreted by **R** as ***regular expressions***.

What is a *regular expression* ? According to Linux help [3], *regular expression* is a pattern that describes a set of strings. Simply speaking, *regular expression* is an "instruction" given to a function on what and how to match or replace *strings*. Using *regular expression* may solve complicated problems (not all the problems) in string matching and manipulation, and may reduce the time spent on **R** code writing and maintanence.

The purpose of this presentation is to introduce *regular expressions* by showing several examples, inspired by real problems. The examples are deliberately simple to attract potential users and not to scare them off.

The concept of *regular expressions* is implemented in several programing lanquages (for example **Perl**, **Python**). *Regular expressions* in **R** are usually restricted and help [1] is not very informative; it does not cover many topics and not all examples work. To compensate for this, **R** functions are written to "understand" **Perl** *regular expression* syntax if you specify argument "perl"=TRUE. You can also read Perl documentation, which is more detailed and widely available. For a quick introduction, you can read more user friendly **Python** help[2] as well, since its *regular expressions* syntax is close to **R**.

2 How NOT to Use Regular Expressions: Beware of Metacharacters

As mentioned before, **R** string matching and modification functions interpret some of their arguments as *regular expressions*. For example, the argument **pattern** of function **gsub()** is a character string interperted as a *regular expression*. If a user is not aware of that he/she may get an error or fail to achieve his/her task and not noticing it. For example, we want to substitute a "$" with a period in string **s** using function **gsub()**

```
> s = "gsub$uses$regular$expressions"
```

Our final result should be string **s1**

```
> s1 = "gsub.uses.regular.expressions"
```

If we do not know that **gsub()** treats argument "pattern" as a *regular expression* we try to do the following.

```
> s1 = gsub(pattern = "$", replacement = ".", "gsub$uses$regular$expressions")
> s1
```

```
[1] "gsub$uses$regular$expressions."
```

String **s1** is not what we wanted because **gsub()** interpreted the character "$" as a *regular expressions* special character. To get the correct result we have to tell **gsub()** to interpret "$" as a regular character. This can be done by preceding "$" with a double backslash. The correct solution is

```
> s1 = gsub(pattern = "\\$", replacement = ".", "gsub$uses$regular$expressions")
> s1
```

```
[1] "gsub.uses.regular.expressions"
```

In *regular expressions*, characters

```
$   *   +   .   ?   [   ]   ^   {   }   |   (   )   \
```

are called ***metacharacters***. When matching any *metacharacter* as a regular character, precede it with a double backslash. When matching a backslash as a regular character, write four backslashes (see examples below).

2.1 Examples of unintentional usage of regular expressions resulting in errors

```
metaChar = c("$","*","+",".","?","[","^","{","|","(","\\")
grep(pattern="$", x=metaChar, value=TRUE)
grep(pattern="\\", x=metaChar, value=TRUE)
grep(pattern="(", x=metaChar, value=TRUE)
gsub(pattern="|", replacement=".", "gsub|uses|regular|expressions")
strsplit(x="strsplit.aslo.uses.regular.expressions", split=".")
```

2.2 Examples of correct ways of avoiding regular expression usage

```
metaChar = c("$","*","+",".","?","[","^","{","|","(","\\")
grep(pattern="\\$", x=metaChar, value=TRUE)
grep(pattern="\\\\", x=metaChar, value=TRUE)
grep(pattern="\\(", x=metaChar, value=TRUE)
gsub(pattern="\\|", replacement=".", "gsub|uses|regular|expressions")
strsplit(x="strsplit.aslo.uses.regular.expressions", split="\\.")
strsplit(x="strsplit.aslo.uses.regular.expressions", split=".", fixed=TRUE)
```

3 Here They Come...

The examples in this section show the importance of using regular expressions for efficient **R** programming.

3.1 Please, Meet Some of the Metacharacters

Suppose we export a dataframe from a text file using function **csv.get()**. For the sake of this example reprodusibility we will generate this data frame rather than actually exporting it. Suppose that, after this procedure, the names of the dataframe variables contain multiple periods.

```
> d1 = data.frame(id...of....patient = c(1, 2), patient....age = c(1,
+     2))
> d1
```

```
  id...of....patient patient....age
1                  1              1
2                  2              2
```

We would like to replace multiple periods with a single one. First, we get rid of 4 periods and then of 3 periods

```
> names(d1) = gsub(pattern = "\\.\\.\\.\\.", replacement = ".",
+     names(d1))
> names(d1)
```

```
[1] "id...of.patient" "patient.age"
```

```
> names(d1) = gsub(pattern = "\\.\\.\\.", replacement = ".", names(d1))
> names(d1)
```

```
[1] "id.of.patient" "patient.age"
```

The main disadvantage of this solution is when the data changes we have to rewrite the code for the new data, which is against the spirit of efficient programming. We want to minimize time spent on code maintenance and free time for the data analysis. This is when *regular expressions* come to the rescue. The following solution will always work, even if the names of the variable change.

```
> d1 = data.frame(id...of....patient = c(1, 2), patient....age = c(1,
+     2))
> d1
```

```
  id...of....patient patient....age
1                  1              1
2                  2              2
```

```
> names(d1) <- gsub(pattern = "\\.+", replacement = ".", x = names(d1))
> names(d1)
```

```
[1] "id.of.patient" "patient.age"
```

Regular expression

```
\\.+
```

tells function **gsub()** to match and replace one or more repititions of a period. *Metacharacter* "+" is the instruction to match one or more repetitions of whatever comes before "+".

Here are some *metacharacters* and their meanings.

```
"." matches everything except for the empty sting "".
"+" the preceding item will be matched one or more times.
"*" the preceding item will be matched zero or more times.
"^" matches the empty string at the at the beginning of a line.
    When used in a character class (see explanation about character classes
    in the following section) means to match any character but the following ones.
"$" matches empty string at the end of a line.
"|" infix operator: OR
"(", ")" brackets for grouping.
"[", "]" character class brackets (see next section).
```

We will see how to use some of them in the following sections.

3.2 Character Classes at Your Service

Consider a task of replacing all values that do not contain letters or digits with NA value. Here is one way to accomplish this without using *regular expressions*

```
> d = data.frame(id = c(11, 22, 33, 44, 55, 66, 77, 88), drug = c("vitamin E",
+     "vitamin ESTER-C", " vitamin E ", "vitamin E(ointment)",
+     "", "vitamin E ", "provitamin E\n", "vit E"), text = c("",
+     "  ", "3 times a day after meal", "once a day", "       ",
+     "  one per day ", "\t", "\n  "), stringsAsFactors = FALSE)
> s = d$text
> unique(s)
```

```
[1] ""                         "  "
[3] "3 times a day after meal" "once a day"
[5] "       "                  "  one per day "
[7] "\t"                       "\n  "
```

```
> s[s == "" | s == "  " | s == "       " | s == "\t" | s == "\n"] = NA
```

As in the previous case, this solution is not good because when the data change, the code has to be changed too. Moreover, the data may be so large that it is impossible (takes too long) to check it manually. Let's see how we can employ *regular expressions* in this case.

We have to explain to function **grep()** that we need only strings containing letter or digits. *Regular expressions* allow us to do that by describing a set of characters. Such set is called a ***character class*** and denoted by square brackets. For example, *regular expression* "[nco]" matches any string containing any of the characters "n", "c", "o".

```
> grep("[nco]", c("nose", "letter38", "window9", "apple0"), value = TRUE)
```

```
[1] "nose"    "window9"
```

Another example of a *character class* is regular expression ”[01234567]” or ”[0-7]” match any string containing any of the digits from ”0” to ”7”.

```
> grep("[01234567]", c("nose", "letter38", "window9", "apple0"),
+     value = TRUE)
```

```
[1] "letter38" "apple0"
```

```
> grep("[0-7]", c("nose", "letter38", "window9", "apple0"), value = TRUE)
```

```
[1] "letter38" "apple0"
```

Here are examples of *character classes*:

```
"[0-9]" - Digits
"[a-z]" - Lower-case letters
"[A-Z]" - Upper-case letters
"[a-zA-Z]" - Alphabetic characters
"[^a-zA-Z]" - Non-alphabetic characters
"[a-zA-Z0-9]" - Alphanumeric characters
"[ \t\n\r\f\v]" - Space characters
"[]$*+.?[^{|(\\#%&~_/<=>'!,:;`\")}@-]"  - Punctuation Characters
```

To solve our problem we use alphanumeric character class.

```
> s = d$text
> s
```

```
[1] ""                         "   "
[3] "3 times a day after meal" "once a day"
[5] "       "                  "  one per day "
[7] "\t"                       "\n  "
```

```
> allIndices = 1:length(s)
> letOrDigIndices = grep("[a-zA-Z0-9]", s)
> blankInd = setdiff(allIndices, letOrDigIndices)
> s[blankInd] = NA
> s
```

```
[1] NA                         NA
[3] "3 times a day after meal" "once a day"
[5] NA                         "  one per day "
[7] NA                         NA
```

This solution is better then the previous one, since, if our data change, we do not have to change our code.
Another way of doing this:

```
> s = d$text
> s = gsub("^$|^( +)$|[\t\n\r\f\v]+", NA, s)
> s
```

```
[1] NA                          NA
[3] "3 times a day after meal" "once a day"
[5] NA                          "  one per day "
[7] NA                          NA
```

If you would like to get rid of extra blank spaces the following code is helpful.

```
> s = d$text
> s = gsub("^([ \t\n\r\f\v]+)|([ \t\n\r\f\v]+)$", "", s)
> s
```

```
[1] ""                          ""
[3] "3 times a day after meal" "once a day"
[5] ""                          "one per day"
[7] ""                          ""
```

```
> s = gsub("^$", NA, s)
> s
```

```
[1] NA                          NA
[3] "3 times a day after meal" "once a day"
[5] NA                          "one per day"
[7] NA                          NA
```

3.3 Watch out for Special Characters Inside a Character Class

In section 2 we learned that in order to match a *metacharacter* as a regular character we have to preceed it with a double backslash. This rule does not hold inside a *character class*; almost all *metacharacters* loose their magic power to the following characters.

```
]   ^   -   \
```

Here is a set of rules on how to match characters as regular characters inside a *character class*

To match "]" inside a *character class* put it first.

To match "-" inside a *character class* put it first or last.

To match "^" inside a *character class* put it anywhere, but first.

To match any other character or metacharacter (but \) inside a *character class* put it anywhere.

R help [1] says that \ remains special inside a *character class*, but it does not say how to match it. I tried to use a double backslash and it worked.

Good luck with all these rules !

4 But Wait, There's More...

Here is another useful example, in which we can practice using both *character classes* and *metacharacters.*

We would like to match all strings containing ”vitamin E”. Without using *regular expressions* it is very easy to miss relevant matches or to match something we do not need.

```
> s = d$drug
> s
```

```
[1] "vitamin E"           "vitamin ESTER-C"     " vitamin E "
[4] "vitamin E(ointment)" ""                    "vitamin E "
[7] "provitamin E\n"      "vit E"
```

```
> grep("vitamin e", s, ignore.case = TRUE, value = TRUE)
```

```
[1] "vitamin E"           "vitamin ESTER-C"     " vitamin E "
[4] "vitamin E(ointment)" "vitamin E "          "provitamin E\n"
```

Using *regular expressions* helps to refine our search.

First, we exclude all strings with ”vitamin e” followed by a letter or nothing (an empty string).

```
> grep("vitamin e($|[^a-zA-Z])", s, ignore.case = TRUE, value = TRUE)
```

```
[1] "vitamin E"           " vitamin E "         "vitamin E(ointment)"
[4] "vitamin E "          "provitamin E\n"
```

Regular expressions ”vitamin e($|[^a-zA-Z])” says to find all character strings ”vitamin e” followed by a non-character (character group [^a-zA-Z]) or an empty string (metacharacter ”$”). Here we also use round brackets to make sure that operator ”or” (metacharacter ”|”) works on the right parts of our *regular expression.*

Second, we include string ”vit e” in our search by using operator ”or” (metacharacter ”|”)

```
> grep("vitamin e($|[^a-zA-Z])|vit e($|[^a-zA-Z])", s, ignore.case = TRUE,
+     value = TRUE)
```

```
[1] "vitamin E"           " vitamin E "         "vitamin E(ointment)"
[4] "vitamin E "          "provitamin E\n"      "vit E"
```

If you would like to get rid of ”provitamin” use the following:

```
> grep("([^a-z]+|^)vitamin e($|[^a-zA-Z])|([^a-z]+|^)vit e($|[^a-zA-Z])",
+     s, ignore.case = TRUE, value = TRUE)
```

```
[1] "vitamin E"           " vitamin E "         "vitamin E(ointment)"
[4] "vitamin E "          "vit E"
```

You can get really fancy:

```
> grep("([^a-z]+|^)vitamin([^a-zA-Z]+)e($|[^a-zA-Z])|([^a-z]+|^)vit([^a-zA-Z]+)e($|[^a
+     s, ignore.case = TRUE, value = TRUE)
```

```
[1] "vitamin E"           " vitamin E "         "vitamin E(ointment)"
[4] "vitamin E "          "vit E"
```

Drug ID	Drug Price
11	$36.
22	2 for $11
33	50% sale

5 Very Useful LaTeX Info and a Puzzler

This section provides motivation for using *regular expressions* and presents a challenge even to those who are familiar with the topic.

Suppose you would like to create a table

```
library(Hmisc)
d1 = data.frame(drugId=c(11,22,33),
                drugInfo=c("$36.", "2 for $11", "50% sale"),
                stringsAsFactors=FALSE)
latex(d1, rowname=NULL, colheads=c("Drug ID", "Drug Price"), file="")
```

When you try to compile your file, LaTeX will give you an error, because one of the variables contains special LaTeX characters ”$” and ”%”. If you want to display those characters in LaTeX you have to escape them (to precede them with a backslash in a LaTeX file or double slash, if you write a LaTeX from R). Let’s do that.

```
> d1 = data.frame(drugId = c(11, 22, 33), drugInfo = c("$36.",
+     "2 for $11", "50% sale"), stringsAsFactors = FALSE)
> d1$drugInfo = gsub("\\$", "\\\\$", d1$drugInfo)
> d1$drugInfo = gsub("\\%", "\\\\%", d1$drugInfo)
> d1

  drugId    drugInfo
1     11      \\$36.
2     22 2 for \\$11
3     33  50\\% sale

> library(Hmisc)
> latex(d1, rowname = NULL, colheads = c("Drug ID", "Drug Price"),
+     file = "")
```

Well, that was easy! Now, imagine that your data contain other LaTeX special characters and you have to make sure that all of them are converted to LaTeX text characters. Here is what you have to convert

```
-------------------------------------------------
-------------VERY USEFUL LATEX INFO--------------
-------------------------------------------------
special      how to write it in LaTeX
LaTeX        to make it look like
character    regular character
#                  \#
```

```
$                 \$
%                 \%
_                 \_
{                 \{
}                 \}
&                 \&
~                 \verb*+~+
^                 \verb*+^+
<                 $<$
>                 $>$
|                 $|$
\                 $\backslash$
```

This task is difficult and time consuming without using *regular expressions*. Using *regular expressions* makes it easier, but it is still not trivial and requires using *regular expression* tools not covered in this presentation. I believe that some of you will find the following challanges interesting.

Puzzler 1
How to convert all LaTeX special characters to LaTeX regular characters without using *regular expressions*.

Puzzler 2
How to convert all LaTeX special characters to LaTeX regular characters using *regular expressions*.

6 Annoying strsplit() Feature

Since we talk about **R** string manipulation, as a side note, I would like to mention an annoying "feature" of **strsplit()** .

If we split a string by a character and this character is at the beginning of the string, **strsplit()** splits the string into an empty string and the rest. If this character is at the end of the string, **strsplit()** does not split it into an empty string and the rest.

For example, this is what **R** does

```
> s = "k000k00k"
> strsplit(s, "k")
[[1]]
[1] ""    "000" "00"
```

This is what **Python** does and **R** should have done

```
>>> s = "k000k00k"
>>> s.split("k")
['', '000', '00', '']
>>>
```

This "feature" sometimes gets in the way and **R** users should watch out for it.

7 Aknowledgements

Cathy, thanks so much for correcting the text. Rafe, thanks for suggesting the phrase ”But wait, there’s more...”. Peggy, I do not even show my writing to anyone, before I show it to you - thanks.

References

[1] http://www.maths.lth.se/help/R/.R/library/base/html/regex.html

[2] http://www.amk.ca/python/howto/regex.html

[3] Linux help (type ”man grep” from a command line)