String manipulation in R

Often the most time intensive aspect of data processing and analysis is correcting errors and homogenizing formats in your data sets. This is especially true if you are attempting to synthesize multiple data sets from different sources. R has a rich set of functions designed for manipulation of strings or the character data type. The topics we cover this week are intended to increase your efficiency in data cleanup, reformatting, and processing of text information.

Regular expressions

What is a regular expression? According to Linux help, a regular expression is a pattern that describes a set of strings. Simply speaking, a 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 in string matching and manipulation. Useful string manipulating functions that use regular expressions are: `grep(), grepl(), regexpr(), gregexpr(), sub(), gsub(), strsplit()`.

Two types of regular expressions are used in **R**, *extended* regular expressions (the default) and *Perl-like* regular expressions used by perl = TRUE. There is also fixed = TRUE which interprets the pattern explicitly as what is given. We will focus on extended regular expressions here.

Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. The whole expression matches zero or more characters. The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Other characters (*metacharacters*; see below) have special meanings and can be used to generalize regular expressions.

By default regular expression matching is “greedy”, so the maximal possible number of repeats is used. This can be changed to ‘minimal’ by appending ? to the quantifier.

The backreference \N, where N = 1 ... 9, matches the substring previously matched by the Nth parenthesized subexpression of the regular expression. This allows a portion of a string matching a regular expression to be temporarily saved for future use.

Metacharacters

Metacharacters are characters with special meaning in regular expressions. You can think of these as “wildcards”. The metacharacters in extended regular expressions include:

.	matches everything except for the empty string “”
?	the preceding item is optional and will be matched at most once.
+	the preceding item will be matched one or more times
*	the preceding item will be matched zero or more times
{n}	the preceding item will be matched n times
{n,}	the preceding item is matched n or more times.
{n,m}	the preceding item is matched at least n times, but not more than m times.
^	matches the beginning of a line
$	matches the end of a line
|	OR

( ) brackets for grouping
[ ] character class brackets
\ the escape character

To explicitly match a metacharacter, rather than have the function interpret its "special meaning", a metacharacter must be *escaped*. To escape a character you must precede it with double forward slashes "\\"; to escape a "\" you need four forward slashes in total.

Character classes
A *character class* is a list of characters enclosed between [and] which matches any single character in that list; unless the first character of the list is the caret ^, then the character class matches any character *not* in the list. For example, the regular expression [0123456789] matches any single digit, and [^abc] matches anything except the characters a, b or c. A range of characters may be specified by giving the first and last characters, separated by a hyphen.

Certain named classes of characters are predefined. These predefined character classes include:

[0-9] or [:digit:]	digits
[a-z] or [:lower:]	lower-case letters
[A-Z] or [:upper:]	upper-case letters
[a-zA-Z] or [:alpha:]	alphabetic characters
[a-zA-Z0-9] or [:alnum:]	alphanumeric characters
[^a-zA-Z] or [^[:alnum:]]	non-alphabetic characters
[\t\n\r\f\v] or [:space:]	whitespace characters
[:punct:]	punctuation characters

In the previous section we learned that in order to match a metacharacter as a regular character we have to precede it with a double backslash. This rule does not hold inside a character class.

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.

Functions for string manipulation and implementation of regular expressions
Regular expressions and the associated metacharacters and character classes exist in R so that they can be used with many of the following functions.

I. A couple notes on loading character data into R from a text file
   a. Strings vs. Factors
      As you have seen in class previously, R does its best to interpret the data type of each column of data read in as a data frame by the functions `read.csv()` or `read.table()`. Unfortunately, when one wants to work with character data these functions have a default setting that will interfere with that goal. To avoid strings being interpreted as *factors* (an additional R data type we haven't covered), the argument `stringsAsFactors` must be set to `FALSE` when using `read.table()` or `read.csv()`.

b. scan

An additional way to avoid misinterpretation of character data types is to use the function `scan()` to load string data. Scan reads a file into a vector with the default delimiter being any white space. You must tell scan to expect character data with the argument `what`. You can force each row into an element of the vector by setting `sep` equal to "\n" and `quote` equal to "'\"".

II. substr

This function can be used to index a portion of a string. This works similarly to indexing a vector, but the square bracket notation doesn't work because the entire string is contained in a single vector element. Instead we use `substr()` to extract an indexed portion of a string. This function can be vectorized too. We can also use the assignment operator (->) to replace a portion of a string with another string. The new string must be the same length as the string being replaced or it will be truncated.

```
> word="apple"
> substr(word,start=2,stop=5)
[1] "pple"
> substr(word,start=1,stop=1)<-"sna"
> word
[1] "spple"
```

III. nchar

This function returns the length of a string. Again this is analogous to the function `length()` for a vector.

```
> word="apple"
> nchar(word)
[1] 5
```

IV. tolower

Converts characters in a character vector to lower case.

```
> word="Apple"
> tolower(word)
[1] "apple"
```

V. toupper

Converts characters in a character vector to upper case.

```
> word="Apple"
> toupper(word)
[1] "APPLE"
```

VI. chartr

Translates a list of characters (`old`) to the corresponding character in `new`. This is a vectorized function and therefore can act on a character vector of any length

```
> word="Apple"
> chartr(old="A",new="a",word)
[1] "apple"
> chart(old="Ae",new="aE",word)
[1] "apple"
```

VII. strsplit

This function splits a string into a list containing multiple strings, based upon a defined delimiter. The delimiter can be defined as a fixed character string or as a regular expression.

```
> word="apple|banana|grape"
> v=strsplit(word,split="|",fixed=TRUE)
> v
[[1]]
[1] "apple" "banana" "grape"
> v[[1]][1]
[1] "apple"
```

VIII. paste

This function combines multiple strings into a single string. A delimiter or separator is defined (default = " ") by the argument `sep` and separates each combined string. Note that paste will not combined a vector into a single character string because it is a vectorized function and looks to combine each element of the vector with another vector of strings.

```
> word="apple|banana|grape"
> v=strsplit(word,split="|",fixed=TRUE)
> v
[[1]]
[1] "apple" "banana" "grape"
> word2=paste(v[[1]],sep="|")
> word2
[1] "apple" "banana" "grape"
> word3=paste(v[[1]],3,sep="")
> word3
[1] "apple3" "banana3" "grape3"
> word4=paste(v[[1]],v[[1]],sep="|")
> word4
[1] "apple|apple" "banana|banana" "grape|grape"
```

IX. grep

This function searches for matches to `pattern` within each element of a character vector and returns an integer vector of the elements of the vector that contain a match (if `value` is set to `FALSE`, which is the default). If `value` is set to `TRUE`, the contents of the matching elements of the character vector are returned. `pattern` can be treated as an explicit string if `fixed` is set to `TRUE` or a regular expression by default. Finally, the non-matching elements can be returned if `invert` is set to `TRUE`.

```
> l=c("apple","banana","grape","10","green.pepper")
> grep("a",l)
[1] 1 2 3
> grep("a",l,value=TRUE)
[1] "apple" "banana" "grape"
> grep("[[:digit:]]",l,value=TRUE)
[1] "10"
> grep(".",l)
[1] 1 2 3 4 5
> grep(".",l,fixed=TRUE
```

```
[1] 5
> grep("\\.",l,invert=TRUE)
[1] 1 2 3 4
> grep("ap|pe",l,value=TRUE
[1] "apple" "grape" "green.pepper"
```

X. grepl

This function works similar to as described above for `grep()`, but returns a logical vector of the same length as the character vector provided as an argument.

```
> l=c("apple","banana","grape","10","green.pepper")
> grepl("a",l)
[1] TRUE TRUE TRUE TRUE FALSE FALSE
```

XI. sub

This function finds patterns within strings in a similar manner to `grep()`, but then substitutes the first instance of a match with a specified string.

```
> l=c("apple","banana","grape")
> sub("a","$",l)
[1] "$pple" "b$nana" "gr$pe"
> sub("[[:alpha:]]","$",l)
[1] "$pple" "$anana" "$rape"
```

XII. gsub

This function works in exactly the same manner as `sub()`, but replaces all matches to `pattern` rather than replacing only the first match. We refer to this as global replacements.

```
> l=c("apple","banana","grape")
> gsub("a","$",l)
[1] "$pple" "b$n$n$" "gr$pe"
> gsub("[[:alpha:]]","$",l)
[1] "$$$$$" "$$$$$$" "$$$$$"
```

XIII. regexpr

This function reports the character position in the provided string(s) where the start of the match with `pattern` occurs. The function also returns the length of the match.

```
> l=c("apple","grape","banana")
> r=regexpr("a",l)
> r
[1] 1 3 2
attr(,"match.length")
[1] 1 1 1
attr(,"useBytes")
[1] TRUE
> r[1]
[1] 1
> attributes(r)
$match.length
[1] 1 1 1

$useBytes
[1] TRUE
> attributes®$match.length[1]
```

```
[1] 1
> gsub("[[:alpha:]]","$",l)
[1] "$$$$$" "$$$$$$" "$$$$$"
```

XIV. gregexpr

This is the global version of `regexpr()`. This function reports all matches to `pattern` rather than only the first match. A list is returned in this case because if more than one match in an element of the provided character vector is possible.

```
> l=c("apple","grape","banana")
> r=gregexpr("a",l)
> r
[[1]]
[1] 1
attr(,"match.length")
[1] 1
attr(,"useBytes")
[1] TRUE

[[2]]
[1] 3
attr(,"match.length")
[1] 1
attr(,"useBytes")
[1] TRUE

[[3]]
[1] 2 4 6
attr(,"match.length")
[1] 1 1 1
attr(,"useBytes")
[1] TRUE
```

XV. regmatches

This function can retrieve the matching components of a string vector for a provided *match object* (produced by `regxpr()` or `gregexpr()`).

```
> l=c("apple","grape","banana")
> r=regexpr("a",l)
> regmatches(l,r)
[1] "a" "a" "a"
```

All of the functions described above have been reimplemented in a package `stringr`. According to the authors of this package, they have homogenized the arguments and made using the string modifying functions of R easier to use. Either I am now used to the basic R functions for string modification or these base functions have been updated since creation of the `stringr` package because the arguments required seem fairly homogeneous to me. If you are interested in the `stringr` package you can take a look at the url below. The manual for the package is also posted on the class website.
http://cran.r-project.org/web/packages/stringr/index.html

Using string manipulation functions to assign values to objects and file names
In my opinion, one of the coolest things that string manipulation affords is the ability to create variables and files based upon information in a dataset or dataframe in the R workspace. A related, useful functionality is the ability to loop over a series of files in your working directory.

The key function that allows this to work is `paste()`. We can then combine `paste()` with two other functions, `get()` and `assign`(), to retrieve are assign information contained in a variable with a name generated by `paste()`.

I. Assigning and accessing data from a variable referenced using a string

`get()` and `assign()` are useful because they allow the assignment to or retrieval of information from a variable using a string. This can be really useful if you need to repeatedly create similar dataframes or vectors with distinct names; in this case you would combine `get()` or `assign()` with the function `paste()`. The best way to explain this is with a demonstration.

```
> assign("x",c(1,2,3))
> x
[1] 1 2 3
> get("x")
[1] 1 2 3
> for(i in 1:3){
+   assign(paste("x",i,sep=""),i*3)
+ }
> x1
[1] 3
> x2
[1] 6
> x3
[1] 9
> for(i in 1:3){
+   cur=get(paste("x",i,sep=""))
+   print(cur)
+ }
[1] 3
[1] 6
[1] 9
```

II. Operating on a series of similar files with `list.files()` and a loop

Similar to accessing data in a variable with `get()` and `assign()`, we can use `paste()` to read and write file names in an automated fashion. The function `list.files()` is also useful in this realm, as it will tell us all the files present in our working directory. We can either use text-manipulating functions (e.g. `grep()`) or the `pattern` argument in `list.files()` itself to filter this list of files and conduct some process on all files that fit some requirement, such as file type.

```
> assign(paste("x",1,sep=""),1:25)
> write.table(get(paste("x",1,sep="")),paste("file",
1,".txt",sep=""),sep=" ",row.names=FALSE)
> x=read.table("file1.txt",header=TRUE,sep=" ")
> x[5,1]
[1] 5
```