Text Processing basics - Tutorial

Sowmya Vajjala

January 5, 2018

The purpose of this file is to provide you with some useful basic text processing commands in R which you will be using in the rest of this course. The lines starting with ## are the outputs of the R commands in the line above it.

Note: This document is adapted from the original lessons that were seen in on an older version of the swirl() class on R Programming. These introduce the basics of working with text data in R. They don't seem to be there in the latest version of the course. So, I am adapting them from the old version to our class. Most of the text is copy-pasted. I rewrote a few parts suiting to this class, sometimes changing the examples, sometimes changing the text itself.

Here are the details of the original class and author:

Course: The R Programming Environment

Lesson: Regular Expressions

Author: Sean Kross

Organization: The Johns Hopkins Data Science Lab

Version: 2.4.2

Generally "text" format for a computer represents any data that is encoded in a plain text format. It can even be a .csv file full of numbers which can be read from notepad or any such text editor. So, learning to manipulate text files is a useful skill in general. Since we will deal with texts written in human language throughout this class and any file format you take (html, pdf etc) essentially requires you to "somehow" get plain text representation of the file, learning to work with text files is an essential component for us too. R has a lot of built-in tools and many more installable libraries for text based analysis. In this tutorial, you will learn about some of them.

1. Strings in R - basics

Text in R is represented as a String object surrounded by single or double quotes. Let us define two string objects below.

```
string1 <- "Hello students,"
string2 <- "this is 410X!"
```

You can use is.character() to tell whether a variable is a string or not. Here are some examples:

```
is.character(string1)
```

```
## [1] TRUE
```

```
is.character(string1)
```

```
## [1] TRUE
```

Paste and Paste0 functions

You can combine several strings using paste() function. Below, I am passing string1 and string2 into paste function as arguments.

```
paste(string1, string2)
```

```
## [1] "Hello students, this is 410X!"
```

By default the paste() function inserts a space between each string you pass into it. You can insert a different string between the arguments by specifying the sep argument. The below command will insert a ___ between string1 and string2 instead of a space.

```
paste(string1, string2, sep = "__")
```

```
## [1] "Hello students,__this is 410X!"
```

If you want no seperation between the strings you pass into paste, you should use paste0. Here is an example:

```
paste0(string1, string2)
```

```
## [1] "Hello students,this is 410X!"
```

Note: You can pass any number of strings into paste and paste0, not just 2!

You can also pass a vector as an argument into paste() function. Let us define an example vector and have a look at this phenomenon:

```
countries <- c("India", "China", "Indonesia", "Sri Lanka", "Japan")
string3 <- "One Asian country I know is: "
paste(string3, countries)
```

```
## [1] "One Asian country I know is:  India"
## [2] "One Asian country I know is:  China"
## [3] "One Asian country I know is:  Indonesia"
## [4] "One Asian country I know is:  Sri Lanka"
## [5] "One Asian country I know is:  Japan"
```

You can also collapse all of the elements of a vector of strings into a single string by specifying the collapse argument. The below line of code creates a single string from countries vector, separating all names by a comma.

```
paste(countries, collapse = ",")
```

```
## [1] "India,China,Indonesia,Sri Lanka,Japan"
```

nchar function

The nchar() function counts the number of characters in a string. Let us start with recalling what string1, string2 and string 3 are in this tutorial.

```
string1
```

```
## [1] "Hello students,"
```

```
string2
```

```
## [1] "this is 410X!"
```

```
string3
```

```
## [1] "One Asian country I know is: "
```

Let us see how many characters each of these strings have:

```
nchar(string1)
```

```
## [1] 15
```

```
nchar(string2)
```

```
## [1] 13
```

```
nchar(string3)
```

```
## [1] 29
```

toupper, tolower functions

As the names indicate, toupper and tolower are used to convert text into upper or lower case. They can be used with both strings and string vectors. Let me take string3 and countries as examples of string and vector below:

```
upperString3 <- toupper(string3)
upperString3
```

```
## [1] "ONE ASIAN COUNTRY I KNOW IS: "
```

```
lowerString3 <- tolower(string3)
lowerString3
```

```
## [1] "one asian country i know is: "
```

```
toupper(countries)
```

```
## [1] "INDIA"     "CHINA"     "INDONESIA" "SRI LANKA" "JAPAN"
```

```
lowerCountries <- tolower(countries)
lowerCountries
```

```
## [1] "india"     "china"     "indonesia" "sri lanka" "japan"
```

```
countries
```

```
## [1] "India"     "China"     "Indonesia" "Sri Lanka" "Japan"
```

2. Regular Expressions for Strings

A regular expression is a string that is written in a specific syntax and is typically used to search for patterns in strings, replacing one string with another etc. It is like a small code language in itself where normal symbols can sometimes have a different meaning. Although this may sound intimidating, they are one of the most useful tools to learn when you are processing text. I will refer to them as regex in the below tutorial. There are two main functions in R that will make regex search possible. grepl() and grep(). Let me start with grepl().

grepl() function

grepl() takes two arguments. The first argument is a regular expression and the second argument is a string to be searched. If the string contains the specified regular expression then grepl() will return TRUE, otherwise it will return FALSE. Let us take a simple regex and look at what this function does:

```
regular_expression <- "a"
string <- "language"

grepl(regular_expression, string)
```

```
## [1] TRUE
```

```
grepl("an", string)
```

```
## [1] TRUE
```

```
grepl("An", string)
```

```
## [1] FALSE
```

The first and second grepl returned TRUE because the strings "a" and "an" are found in "language". The third grepl returned FALSE because the string "An" is not found in "language" (note: regex is case-sensitive!). This is an example of the simplest form of regular expression - check for the existence of a specific string. However, the real purpose of regex is to search for patterns, not exact strings.

There's a dataset that comes with R called state.name which is a vector of strings, one for each state in the United States of America. We're going to use this vector in several of the following examples.

```
state.name
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

Let's build a regular expression for identifying several strings in this vector, specifically a regular expression that will match names of states that both start and end with a vowel. The state name could start and end with any vowel, so we won't be able to match exact sub-strings like in the previous examples. Thankfully we can use metacharacters to look for vowels and other parts of strings. The metacharacters used in regex language are seen in another file regex-description.pdf in the Blackboard assignments folder. Open that file and start browsing through the examples below:

The first metacharacter that we'll discuss is ".". The metacharacter that only consists of a period represents any character other than a new line (we'll discuss new lines soon). Enter grepl(".", "Maryland") into the R console to see if there is one instance of any character present.

```
grepl(".", "Maryland")
```

```
## [1] TRUE
```

```
grepl(".", "")
```

```
## [1] FALSE
```

Since a period matches any character, the first line returns TRUE. In the second line, we just have an empty string, it will return FALSE. Now, look at this:

```
grepl("a.b", c("aaa", "aab", "abb", "acadb"))
```

```
## [1] FALSE  TRUE  TRUE  TRUE
```

This is where the period regex character is most useful. It will find all strings that match the description "a.b" which means "any string that starts with a, ends with b, with anything in between. This is true for the last 3 strings of the vector above. Hence, grepl returns FALSE TRUE TRUE TRUE.

The "+" metacharacter indicates that one or more of the preceding expression should be present and "*" indicates that zero or more of the preceding expression is present. Let us see an example with our countries vector:

```
countries
```

```
## [1] "India"     "China"     "Indonesia" "Sri Lanka" "Japan"
```

```
grepl("d+", countries)
```

```
## [1]  TRUE FALSE  TRUE FALSE FALSE
```

```
grepl("d*", countries)
```

```
## [1] TRUE TRUE TRUE TRUE TRUE
```

In the first grepl, it looks for the presence of one or more consecutive d's in all the strings of the vector countries. In the second grepl, it looks for 0 or more consecutive d's in each string. So, in second case, all strings get a TRUE.

You can also specify exact numbers of expressions using curly brackets {}. For example "a{5}" specifies "a exactly five times," "a{2,5}" specifies "a between 2 and 5 times," and "a{2,}" specifies "a at least 2 times". For example, to see if the string "Mississippi" contains exactly two adjacent "s" (using curly brackets), write:

```
grepl("s{2}", "Mississippi")
```

```
## [1] TRUE
```

To see if the string "Mississippi" contains two or three adjacent "i" (using curly brackets), write:

```
grepl("i{2,3}", "Mississippi")
```

```
## [1] FALSE
```

In addition to curly brackets you can use parentheses "()" to create a capturing group. A capturing group allows you to use quantifiers on other regular expressions. Let's use a capturing group to see if the string "Mississippi" contains the substring "iss" twice.

```
grepl("(iss){2}", "Mississippi")
```

```
## [1] TRUE
```

Notice that the regular expression in the previous example is essentially the same as "ississ"

Now, which regular expression roughtly translates to - Does "Mississippi" contain the pattern of an "i" followed by 2 of any character, with that pattern repeated three times adjacently? The complex looking question is expressed in regex language as:

```
grepl("(i.{2}){3}", "Mississippi")
```

```
## [1] TRUE
```

Let us look at some other metacharacters: words ("\w"), digits ("\d"), and whitespace characters ("\s"). Words specify any letter, digit, or a underscore, digits specify the digits 0 through 9, and whitespace specifies line breaks, tabs, or spaces. Each of these character sets have their own compliments - not words ("\W"), not

digits ("\D"), and not whitespace characters ("\S"). Each specifies all of the characters not included in their corresponding character sets.

To see if a string "0123456789" contains a digit or not.

```
grepl("\\d", "0123456789")
```

```
## [1] TRUE
```

```
grepl("\\D", "0123456789")
```

```
## [1] FALSE
```

First one is true, and consequently, second one is false. Here are some other expressions - look at the outputs too and check if you got them right.

```
grepl("\\s", "abcdefghijklmnopqrstuvwxyz0123456789")
```

```
## [1] FALSE
```

```
grepl("\\w", "abcdefghijklmnopqrstuvwxyz0123456789")
```

```
## [1] TRUE
```

```
grepl("\\d", "abcdefghijklmnopqrstuvwxyz0123456789")
```

```
## [1] TRUE
```

You can also specify specific character sets using straight brackets []. For example a character set of just the vowels would look like - "[aeiou]". You can find the complement to a specific character by putting a carrot ^ after the first bracket. For example "[^aeiou]" matches all characters except the lowercase vowels. You can also specify ranges of characters using a hyphen - inside of the brackets. For example "[a-m]" matches all of the lowercase characters between a and m, while "[5-8]" matches any digit between 5 and 8 inclusive. Look at some examples below:

```
grepl("[aeiou]", "rhythms")
```

```
## [1] FALSE
```

```
grepl("[a-mA-M]", "ABC")
```

```
## [1] TRUE
```

```
grepl("[a-m]", "ABC")
```

```
## [1] FALSE
```

```
grepl("[a-mA-M]", "ABC")
```

```
## [1] TRUE
```

You might be wondering how you can use regular expressions to match a particular punctuation mark since many punctuation marks are used as metacharacters! Putting two backslashes before a punctuation mark that is also a metacharacter indicates that you are looking for the symbol and not the metacharacter meaning. For example "\." indicates you are trying to match a period in a string. Example to use grepl() to see whether or not a peroid exists in the string"http://www.iastate.edu/"

```
grepl("\\.", "http://www.iastate.edu/")
```

```
## [1] TRUE
```

Some more examples:

```
grepl("\\+", "tragedy + time = humor")
```

```
## [1] TRUE
```

```
grepl("\\.", "tragedy + time = humor")
```

```
## [1] FALSE
```

```
grepl("\\*", "tragedy + time = humor")
```

```
## [1] FALSE
```

There are also metacharacters for matching the beginning and the end of a string which are "^" and "$" respectively. To see which strings in a string vector starts with a "a":

```
grepl("^a", c("bab", "aab"))
```

```
## [1] FALSE  TRUE
```

Ro see if the two strings from above end with the letter "b":

```
grepl("b$", c("bab", "aab"))
```

```
## [1] TRUE TRUE
```

The last metacharacter we'll discuss is the OR metacharacter ("|"). The OR metacharacter matches either the regex on the left or the regex on the right side of this character. For example, to see which strings contain "a" or "b":

```
grepl("a|b", c("abc", "bcd", "cde"))
```

```
## [1]  TRUE  TRUE FALSE
```

To see which strings start with "North"" or "South":

```
grepl("^(North|South)", c("South Dakota", "North Carolina", "West Virginia", "I am North", "I am South"))
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE
```

Okay, to remind you what got us started on this regex trip, we wanted to get all states from state.name that start and end in a vowel. Remember?? :) Now, we learned enough regex to do this in R! This regular expression must match the beginning of a string, then one instance of a capitalized vowel, then any characters until one instance of a lowercase vowel followed by the end of the string. Here is the regex to do that:

```
regex_we_want <- "^[AEIOU]{1}.+[aeiou]{1}$"
```

Now,use this regex to store a logical vector which indicates which strings in state.name matches the regular expression.

```
vowel_states <- grepl(regex_we_want, state.name)
vowel_states
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12]  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE FALSE FALSE
```

Now finally index state.name with the vector you created in the last question in order to see which state names start and end with a vowel!

```
state.name[vowel_states]
```

```
## [1] "Alabama"  "Alaska"   "Arizona"  "Idaho"    "Indiana"  "Iowa"
## [7] "Ohio"     "Oklahoma"
```

As you can see, grepl() is a powerful function to use regular expressions. I mentioned earlier that there is another function called grep(). While grepl() returns TRUE or FALSE referring to whether a string matches the given regex, grep() returns a number indicating the first position at which a match happens.

```
grepl("a", "language")
```

```
## [1] TRUE
```

```
grep("a", "language")
```

```
## [1] 1
```

sub() and gsub() functions

Two other useful functions with regex are: sub() and gsub().

The sub() function takes as arguments a regex, a "replacement," and a vector of strings. This function will replace the first instance of that regex found in each string. See this example where all first occurrences of "a" are replaced with "11" in each string of the vector countries.

```
sub("a", "11",countries)
```

```
## [1] "Indi11"     "Chin11"     "Indonesi11" "Sri L11nka" "J11pan"
```

The gsub() function is nearly the same as sub() except it will replace every instance of the regex that is matched in each string. Let us enter the exact line of code you used in the previous question, except use gsub() instead of sub().

```
gsub("a", "11",countries)
```

```
## [1] "Indi11"      "Chin11"      "Indonesi11"  "Sri L11nk11" "J11p11n"
```

strsplit() function

strsplit() is used to split a string wherever a pattern is matched. For example, see these:

```
strsplit("Mississippi", "ss")
```

```
## [[1]]
## [1] "Mi"   "i"    "ippi"
```

```
strsplit(c("Mississippi", "Missouri", "Mississauga", "Mister"), "ss")
```

```
## [[1]]
## [1] "Mi"   "i"    "ippi"
##
## [[2]]
## [1] "Mi"   "ouri"
##
## [[3]]
## [1] "Mi"   "i"    "auga"
##
## [[4]]
## [1] "Mister"
```

Introduction to stringr package:

The stringr package is wonderful for working with strings in R. Most of the functions in stringr take the same two arguments, a string and then a regex. This package takes a "data first" approach to functions involving regex, so usually the string is the first argument and the regex is the second argument. The majority of the function names in stringr begin with str_. First, add the library into your current session:

```
library(stringr)
```

Let us look at some of the functions in stringr:

The str_extract() function returns the sub-string of a string that matches the providied regular expression.

```
str_extract("Apollo13", "[0-9]+")
```

```
## [1] "13"
```

The str_order() function returns a numeric vector that corresponds to the alphabetical order of the strings in the provided vector.

```
str_order(c("p", "e", "n", "g"))
```

```
## [1] 2 4 3 1
```

The str_pad() function pads strings with other characters which is often useful when the string is going to be eventually printed for a person to read.

```
str_pad("Thai", width = 8, side = "left", pad = "-")
```

```
## [1] "----Thai"
```

The str_to_title() function acts just like tolower() and toupper() except it puts strings into Title Case.

```
str_to_title(c("CAPS", "low", "Title"))
```

```
## [1] "Caps"  "Low"   "Title"
```

The str_trim() function deletes whitespace from both sides of a string.

```
str_trim(" trim me ")
```

```
## [1] "trim me"
```

Lastly the word() function allows you to index each word in a string as if it were a vector.

```
word("See Spot run.", 2)
```

```
## [1] "Spot"
```

Now, start doing your Assignment 1! Now that you endured through this, A1 will be a breeze! This document will be useful for doing other assignments as well.