Getting and Cleaning Data: Week 4

Video 4-1: Editing text variables

Fixing character vectors: tolower(), toupper()

```
if(!file.exists("data")) {
  dir.create("data")
}
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-
2aru/rows.csv?accessType=DOWNLOAD"
if(!file.exists("./data/cameras.csv")) {
  download.file(fileUrl, destfile="./data/cameras.csv",
method="curl")
}
```

```
cameraData <- read.csv("./data/cameras.csv")
names(cameraData)
```

```
## [1] "address"      "direction"    "street"       "crossStreet"
## [5] "intersection" "Location.1"
```

```
tolower(names(cameraData))
```

```
## [1] "address"      "direction"    "street"       "crossstreet"
## [5] "intersection" "location.1"
```

My note: Converting “crossStreet” is a bad idea. What is truly error-prone is having three “s” characters in a row, overlapping two words. I do not agree that “crossstreet” is in any way better than “crossStreet”.

Fixing character vectors: strsplit()

- Good for automatically splitting variable names
- Important parameters: x, split

Note that the split parameter is a regular expression, a topic not yet covered. We need to add a backslash in front of it so that the regex engine knows we are specifying the period character; “.” by itself in regex means “any character” (it's a wildcard). And we specify two backslashes because the expression is inside double-quotes, and in that context, a single backslash is used to denote an escaped character such as a tab (“\t”).

```
splitNames = strsplit(names(cameraData), split = "\\.")
splitNames[[5]]
```

```
## [1] "intersection"
```

```
splitNames[[6]]  # 'Location.1' changed into two values, 'Location' and '1'
```

```
## [1] "Location" "1"
```

Quick aside: lists

The following creates a list with three elements, the first two of which are named.

```
mylist <- list(letters = c("A", "b", "c"), numbers = 1:3, matrix(1:25, ncol = 5))
head(mylist)
```

```
## $letters
## [1] "A" "b" "c"
##
## $numbers
## [1] 1 2 3
##
## [[3]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
```

This shows three ways of selecting the first element of the list:

```
mylist[1]  # returns a list
```

```
## $letters
## [1] "A" "b" "c"
```

```
class(mylist[1])
```

```
## [1] "list"
```

```
mylist$letters  # returns a vector
```

```
## [1] "A" "b" "c"
```

```
class(mylist$letters)
```

```
## [1] "character"
```

```
mylist[[1]]  # returns a vector
```

```
## [1] "A" "b" "c"
```

```
class(mylist[[1]])
```

```
## [1] "character"
```

Fixing character vectors: sapply

- Applies a function to each element in a vector or list
- Important parameters: x, FUN

```
splitNames[[6]][1]  #splitNames[[6]] contains two values, 'Location' and '1'; this
returns the first
```

```
## [1] "Location"
```

```
firstElement <- function(x) {
    x[1]
}  # returns first element of x
sapply(splitNames, firstElement)  # passes each element in splitNames into
firstElement
```

```
## [1] "address"      "direction"    "street"       "crossStreet"
## [5] "intersection" "Location"
```

Peer review experiment data

Same as used in week 3.

http://www.plosone.org/article/info:doi/10.1371/journal.pone.0026895

```
if (!file.exists("./data")) {
    dir.create("./data")
}
fileUrl1 = "https://dl.dropboxusercontent.com/u/7710864/data/reviews-apr29.csv"
fileUrl2 = "https://dl.dropboxusercontent.com/u/7710864/data/solutions-apr29.csv"
if (!file.exists("./data/reviews.csv")) {
    download.file(fileUrl1, destfile = "./data/reviews.csv", method = "curl")
}
if (!file.exists("./data/solutions.csv")) {
    download.file(fileUrl2, destfile = "./data/solutions.csv", method = "curl")
}
reviews = read.csv("./data/reviews.csv")
solutions = read.csv("./data/solutions.csv")
head(reviews, 2)
```

```
##   id solution_id reviewer_id      start       stop time_left
accept
## 1  1           3          27 1304095698 1304095758      1754
1
## 2  2           4          22 1304095188 1304095206      2306
1
```

```
head(solutions, 2)
```

```
##   id problem_id subject_id      start       stop time_left
answer
## 1  1        156         29 1304095119 1304095169      2343
B
## 2  2        269         25 1304095119 1304095183      2329
C
```

Fixing character vectors: sub

- Important parameters: pattern, replacement, x

Removes all underscores:

```
names(reviews)
```

```
## [1] "id"          "solution_id" "reviewer_id" "start"
"stop"
## [6] "time_left"   "accept"
```

```
sub(pattern = "_", replacement = "", x = names(reviews))
```

```
## [1] "id"         "solutionid" "reviewerid" "start"      "stop"
## [6] "timeleft"   "accept"
```

Fixing character vectors: gsub

- sub replaces only the first match
- gsub ("global" sub) replaces all matches

```
testName <- "this_is_a_test"
sub("_", "", testName)
```

```
## [1] "thisis_a_test"
```

```
gsub("_", "", testName)
```

```
## [1] "thisisatest"
```

Finding values: grep, grepl

- grep returns indices of elements in a data set that match a pattern
- grepl returns an array of TRUE/FALSE values for all elements in a data, TRUE=match

```
grep(pattern = "Alameda", x = cameraData$intersection)  # Find all intersections
that contain 'Alameda'
```

```
## [1]  4  5 36
```

```
table(grepl(pattern = "Alameda", x = cameraData$intersection))  # build a table of
intersections that do, and do not, contain 'Alameda'
```

```
##
## FALSE  TRUE
##    77     3
```

```
cameraData2 <- cameraData[!grepl(pattern = "Alameda", x =
cameraData$intersection),
    ]  # subset of intersections that do not contain 'Alameda'
```

More on grep

My note: regular expressions are an art. grep is one tool for working with regular expressions, except that grep is in fact many tools, or rather flavors of a regex tool. For example, the Perl language has its own variant of regex. Many bash shells allow emulation of Perl regex; but they're not all exactly the same. The grep and grepl functions both have perl=TRUE options, but how closely do they match the Perl language's regexes? I don't know yet. At any rate, if you're new to regex, it will take time and practice to master them (much like R).

```
grep(pattern = "Alameda", x = cameraData$intersection, value = TRUE)  # value=TRUE
means return the values, rather than indices to the original data set
```

```
## [1] "The Alameda  & 33rd St"   "E 33rd  & The Alameda"
## [3] "Harford \n & The Alameda"
```

```
grep("JeffStreet", cameraData$intersection)  # if no matches, returns intger(0)
```

```
## integer(0)
```

```
length(grep("JeffStreet", cameraData$intersection))  # returns 0
```

```
## [1] 0
```

More useful string functions

Note: nchar, substr, paste, paste0 are all in base R.

```
library(stringr)
nchar("Jeffrey Leek")
```

```
## [1] 12
```

```
substr("Jeffrey Leek", 1, 7)
```

```
## [1] "Jeffrey"
```

```
paste("Jeffrey", "Leek")  # sep defaults to ' '
```

```
## [1] "Jeffrey Leek"
```

```
paste0("Jeffrey", "Leek")  # there is no sep param, just concatenates
```

```
## [1] "JeffreyLeek"
```

```
str_trim("Jeff      ")  # trims trailing whitespace
```

```
## [1] "Jeff"
```

```
str_trim("           Jeff        ", side = "both")  # or on both sides (can be 
left, right, both)
```

```
## [1] "Jeff"
```

Important points about text in data sets

- Names of variables should be
  - All lower case when possible
  - Descriptive (Diagnosis versus Dx)
  - Not duplicated
  - Not have unde3rscores or dots or white spaces
- Variables with character values
  - Should usually be made into factor variables (depends on application)
  - SHould be descriptive (use TRUE/FALSE instead of 0/1 and Male/Female instead of 0/1 or M/F)

My note: Some of the above is absurd. It could mean that Rtc_Sep_Stages_Sign_X_12_3 becomes rctsepstagessignx123, and if you think that's an improvement, well you're just wrong! Pffffft.

Video 4-2: Regular Expressions I

Regular expressions were introduced in previous video, this section will explore regex more fully.

Regular expressions

- Regular expressions can be thought of as a combination of literals and metacharacters
- To draw an analogy with natural language, think of literal text forming the words of this language, and the metacharacters defining its grammar
- Regular expressions have a rich set of metacharacters

Literals

Simplest pattern consists of only literals. The literal “nuclear” would match the following lines:

```
Ooh, I just learned that to keep myself alive after a nuclear
blast! All I have to do is milk some rats then drink the milk.
Awesome. :}

Laozi says nuclear weapons are mas macho.

Chaos in a country that has nuclear weapons--not good.

my nephew is trying to teach me nuclear physics, or possibly just
trying to show how smart he is so I'll be proud of him (which I
am)

lol if you ever say "nuclear"" people immediately think DEATH by
radiation LOL
```

Regular expressions

- Simplest pattern consists of only literals; a match occurs if the sequence of literals occurs anywhere in the text being tested
- What if we only want the word “Obama”? Or sentences that end in the word “Clinton”, “clinton” or “clinto”?

We need a way to express

- whitespace word boundaries
- sets of literals
- the beginning and end of a line
- alternatives (“war” or “peace”)
- Metacharacters to the rescue

Metacharacters

Some metacharacters represent the start of a line

(Note: The ^ metacharacter means “start of line”; it is the *only* metacharacter I've ever seen that represents the start of a line.)

```
^i think
```

which will match

```
i think we all rule for participating
i think i have been outed
i think this will be quite fun actually
```

It will not match:

```
that's not what i think
```

because “i think” is not at the start of the line

$ represents the end of a line

```
morning$
```

This matches any line that ends with “morning” (along with a line end sequence such as linefeed, or carriage return-linefeed)

Character classes with []

We can list a set of characters we will accept at a given point in the match.

(NOTE: What this means is that *any one* of the set will count as a match)

The following (which defines four consecutive character classes) will match “bush” in any combination of upper and lower case. So “Bush”, “bUsh”, “bush”, “BUSH” would all match.

```
[Bb][Uu][Ss][Hh]
```

You can combine metacharacters. The following matches either “i am” or “I am”, so long as it is at the start of a line:

```
^[Ii] am
```

So this would match “i am”, “I am”, and “i amped”, at the start of a line.

Similarly, you can specify a range of letters <u>a-z</u> or <u>A-Z</u>; notice that order doesn't matter.

NOTE: Order *most certainly does* matter. If you specify [z-a] it is very different from [a-z]! I think what is meant here is that if you combine more than one range of letters in a single class, then the order of the ranges doesn't matter, e.g., [a-zA-Z] is the same as [A-Za-z].

So this:

```
^[0-9][a-zA-Z]
```

will match any single numeric digit, followed by any single letter, uppre or lower case, at the start of a line.

When used at the beginning of a character class, the ^ is also a metacharacter and indicates characters that should not be matched.

For example this:

```
[^?.]$
```

will match any lines that do *not* end in either a question mark or a period.

Also note that ^ has this functionality *only at the the beginning* of a character class. This:

```
[a-z^]
```

matches any lower case letter or the ^ symbol.

Video 4-3: Regular Expressions II

I veer away from many of the examples in the video and provide my own, in order to simplify some things.

More metacharacters

“.” is used to refer to any character. So:

```
9.11
```

matches a “9” followed by any character, followed by “11”.

NOTE: This isn't entirely true. If 9 were followed by a linefeed, for example, this would not (by default) match. Generally speaking, regular expressions work on only one line at a time. But apart from line-ending characters, the above would match any string that had a 9, then any character, then 11. So it would match “9.11”, “9211”, “9:11”, “9a11” and so on.

The “l” character is used to create “or” conditions in order to combine subexpressions:

```
flood|fire
```

this matches either “flood” or “fire”.

We can include any number of alternatives:

```
flood|earthquake|hurricane|coldfire
```

The alternatives can also be more complex expressions:

```
^[Gg]ood|[Bb]ad
```

The above matches either:

- “Good” or “good” at the start of a line
- “Bad” or “bad” anywhere in a line

So the l takes precedence over the ˒ kind of like a * taking precedence over a + in math.

If you wanted to match either “Good” or “good” or “Bad” or “bad” at the start of a line, group both expressions in parentheses:

```
^([Gg]ood|[Bb]ad)
```

The question mark indicates that the indicated expression is optional.

The following matches “George Bush”, “george bush”, “George W. Bush”, “george w. bush”. So it will match with or without a “ W.” between “George” and “Bush”.

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

Note above the “.” which indicates that we want to match the period character. I.e., it does not serve its normal purpose as a wildcard matching any character. This can be required for other characters; for instance, if you want to match “\(", then specify \ \). This is generally true for any metacharacter, but note what I said above: regular expressions are an art, and they take time to master. Confusions about escaping can crop up, especially since different tools can require different kinds of escaping. Even within R, the functions that use regular expressions allow you to specify that you are using Perl-style expressions, which can require different kinds of escaping. In a nutshell, be prepared for some confusion if you are new to regular expressions.

More metacharacters: * and +

The * metacharacter means "any number of the preceding character/expression", and + means "one or more of the preceding character/expression".

So this:

```
(.*)
```

will match any line that has both a left and right parens, whether there is any text between them or not.

NOTE: The video is wrong here. You must escape the parentheses, because they themselves are metacharacters. So the expression would be:

```
\(.*\)
```

This:

```
[0-9]+
```

will match any string that has at least one numeric digit, and this:

```
[0-9]+ (.*)[0-9]+
```

will match any line that has one or more digits, followed by a space, followed by zero or more of any character, followed by one or more digits.

Note that the parentheses used here are correct; the examples of matched lines in the video do not have parentheses in them. So in the above, the parentheses are metacharacters, not literals. If you are new to regular expressions, this would probably have you scratching your head.

More metacharacters: { and }

These metacharacters are used to define the minimum and maximum number of matches of an expression. For example this:

```
[Bb]ush( +[^ ]+ +){1,5} debate
```

looks for the following, in order:

- Either "Bush" or "bush"
- The following, from 1 to 5 times:
  - one or more spaces
  - one or more characters that are anything but a space
  - one or more spaces

- " debate"

So in a sense, it looks for "Bush", then one to five words, then "debate" (note that this is not an entirely accurate desription, one of those "words" might be an ellipse, "…", and this hints at the difficulties you can face with regular experssions. Did I mention is't an art?)

If you use {3} in the above, it means match exactly three times. If you use {2,7} it means match between 2 and 7 times. If you use {3,} it means match at least three times.

More metacharacters: (and)

- In most implementations of regular expressions, the parentheses not only limit the scope of alternatives divided by a "l", but also can be used to "remember" text matched by the subexpression enclosed (this is called a backreference)
- We refer to the matched text with \1, \2, etc. (where \1 refers to the first expression in the pattern enclosed by the parentheses, \2 to the second, and so on)

NOTE: Again, it's an art. Working with backreferences starts out simple, but grows more complex as your needs become more sophisticated. Two quick things I'll mention without elaborating on:

- You can have one expression in parentheses enclosed in another set of parentheses. Which backreference is which? (The outer expression is the lower number, actully.)
- What if you want to enclose alternatives with parentheses, but you don't want it to count as a backreference? (Possible, but too complex to go into here.)

So this:

```
([a-zA-Z]+) \1
```

will match any sequence of one or more letters that is repeated and separated by a space. In effect this says, "match any sequence of letters and store it as backreference 1, then a space, then whatever is stored as backreference 1".

"Greedy" matches

The * metacharacter is "greedy" by default, so this:

```
^s(.*)s
```

will match as many characters as possible after the "s" at the beginning of a line. For example, if the string searched is "soon as pass" then it will not match "soon as", but rather, "soon as pass". It will match as many characters as it possibly can.

What if you want to match starting with an s, then any characters, but up to and including only the second s in the line? Do this:

```
^s(.*?)s
```

The question mark directly after the asterisk is another special metacharacter that signals that non-greedy matching should be performed. So the command now is, "look for s at the start of a line, then zero or more characters followed by an s, but stop at the second s in the line".

Summary

- Regular expressions are used in many different languages, not just R

- Regular expressions are composed of literals and metacharacters that represent sets or classes of characters/words
- Text processing via regular expressions is a very powerful way to extract data from “unfriendly” sources (not all data comes as a CSV file)
- Used with the expressions grep, grep, sub, gsub, and others that involve searching for text strings

One final note: Remember that whenever you use a backslash in a string expression surrouned by double quotes, you have to double the backslash, so the regular expression . would become “.”