Lab 6: Text Manipulation

Statistical Computing, 36-350

Week of Tuesday October 5, 2019

Name:
Andrew ID:
Collaborated with:

This lab is to be done in class (completed outside of class time if need be). You can collaborate with your classmates, but you must identify their names above, and you must submit **your own** lab as an knitted PDF file on Gradescope, by Friday 9pm, this week.

This week's agenda: basic string manipulations; practice reading in and summarizing real text data (Shakespeare); practice with iteration; just a little bit of regular expressions.

Q1. Some string basics

- **1a.** Define two strings variables, equal to "Statistical Computing" and 'Statistical Computing', and check whether they are equal. What do you conclude about the use of double versus single quotation marks for creating strings in R? Give an example that shows why might we prefer to use double quotation marks as the standard (think of apostrophes).

```
# YOUR CODE GOES HERE
```

- **1b.** The functions `tolower()` and `toupper()` do as you'd expect: they convert strings to all lower case characters, and all upper case characters, respectively. Apply them to the strings below, as directed by the comments, to observe their behavior.

```
"I'M NOT ANGRY I SWEAR"          # Convert to lower case
```

```
## [1] "I'M NOT ANGRY I SWEAR"
```

```
"Mom, I don't want my veggies"  # Convert to upper case
```

```
## [1] "Mom, I don't want my veggies"
```

```
"Hulk, sMasH"                    # Convert to upper case
```

```
## [1] "Hulk, sMasH"
```

```
"R2-D2 is in prime condition, a real bargain!" # Convert to lower case
```

```
## [1] "R2-D2 is in prime condition, a real bargain!"
```

```
# YOUR CODE GOES HERE
```

- **1c.** Consider the string vector `presidents` of length 5 below, containing the last names of past US presidents. Define a string vector `first.letters` to contain the first letters of each of these 5 last names. Hint: use `substr()`, and take advantage of vectorization; this should only require one line of code. Define `first.letters.scrambled` to be the output of `sample(first.letters)` (the `sample()` function can be used to perform random permutations, we'll learn more about it later in the course). Lastly, reset the first letter of each last name stored in `presidents` according to the scrambled letters

in `first.letters.scrambled`. Hint: use `substr()` again, and take advantage of vectorization; this should only take one line of code. Display these new last names.

```
presidents = c("Clinton", "Bush", "Reagan", "Carter", "Ford")
```

```
# YOUR CODE GOES HERE
```

- **1d.** Now consider the string `phrase` defined below. Using `substr()`, replace the first four characters in `phrase` by "Provide". Print `phrase` to the console, and describe the behavior you are observing. Using `substr()` again, replace the last five characters in `phrase` by "kit" (don't use the length of `phrase` as magic constant in the call to `substr()`, instead, compute the length using `nchar()`). Print `phrase` to the console, and describe the behavior you are observing.

```
phrase = "Give me a break"
```

```
# YOUR CODE GOES HERE
```

- **1e.** Consider the string `ingredients` defined below. Using `strsplit()`, split this string up into a string vector of length 5, with elements "chickpeas", "tahini", "olive oil", "garlic", and "salt." Using `paste()`, combine this string vector into a single string "chickpeas + tahini + olive oil + garlic + salt". Then produce a final string of the same format, but where the ingredients are sorted in alphabetical (increasing) order.

```
ingredients = "chickpeas, tahini, olive oil, garlic, salt"
```

```
# YOUR CODE GOES HERE
```

Shakespeare's complete works

Project Gutenberg offers over 50,000 free online books, especially old books (classic literature), for which copyright has expired. We're going to look at the complete works of William Shakespeare, taken from the Project Gutenberg website.

To avoid hitting the Project Gutenberg server over and over again, we've grabbed a text file from them that contains the complete works of William Shakespeare and put it on our course website. Visit http://www.stat.cmu.edu/~ryantibs/statcomp/data/shakespeare.txt in your web browser and just skim through this text file a little bit to get a sense of what it contains (a whole lot!).

Q2. Reading in text, basic exploratory tasks

- **2a.** Read in the Shakespeare data linked above into your R session with `readLines()`. Make sure you are reading the data file directly from the web (rather than locally, from a downloaded file on your computer). Call the result `shakespeare.lines`. This should be a vector of strings, each element representing a "line" of text. Print the first 5 lines. How many lines are there? How many characters in the longest line? What is the average number of characters per line? How many lines are there with zero characters (empty lines)? Hint: each of these queries should only require one line of code; for the last one, use an on-the-fly Boolean comparison and `sum()`.

```
# YOUR CODE GOES HERE
```

- **2b.** Remove all empty lines from `shakespeare.lines` (i.e., lines with zero characters). Check that that the new length of `shakespeare.lines` makes sense to you.

```
# YOUR CODE GOES HERE
```

- **2c.** Collapse the lines in `shakespeare.lines` into one big string, separating each line by a space in doing so, using `paste()`. Call the resulting string `shakespeare.all`. How many characters does this string have? How does this compare to the sum of characters in `shakespeare.lines`, and does this make sense to you?

```
# YOUR CODE GOES HERE
```

- **2d.** Split up `shakespeare.all` into words, using `strsplit()` with `split=" "`. Call the resulting string vector (note: here we are asking you for a vector, not a list) `shakespeare.words`. How long is this vector, i.e., how many words are there? Using the `unique()` function, compute and store the unique words as `shakespeare.words.unique`. How many unique words are there?

```
# YOUR CODE GOES HERE
```

- **2e.** Plot a histogram of the number of characters of the words in `shakespeare.words.unique`. You will have to set a large value of the `breaks` argument (say, `breaks=50`) in order to see in more detail what is going on. What does the bulk of this distribution look like to you? Why is the x-axis on the histogram extended so far to the right (what does this tell you about the right tail of the distribution)?

```
# YOUR CODE GOES HERE
```

- **2f.** Reminder: the `sort()` function sorts a given vector into increasing order; its close friend, the `order()` function, returns the indices that put the vector into increasing order. Both functions can take `decreasing=TRUE` as an argument, to sort/find indices according to decreasing order. See the code below for an example.

```
set.seed(0)
(x = round(runif(5, -1, 1), 2))
```

```
## [1]  0.79 -0.47 -0.26  0.15  0.82
```

```
sort(x, decreasing=TRUE)
```

```
## [1]  0.82  0.79  0.15 -0.26 -0.47
```

```
order(x, decreasing=TRUE)
```

```
## [1] 5 1 4 3 2
```

Using the `order()` function, find the indices that correspond to the top 5 longest words in `shakespeare.words.unique`. Then, print the top 5 longest words themselves. Do you recognize any of these as actual words? **Challenge**: try to pronounce the fourth longest word! What does it mean?

```
# YOUR CODE GOES HERE
```

Q3. Computing word counts

- **3a.** Using `table()`, compute counts for the words in `shakespeare.words`, and save the result as `shakespeare.wordtab`. How long is `shakespeare.wordtab`, and is this equal to the number of unique words (as computed above)? Using named indexing, answer: how many times does the word "thou" appear? The word "rumour"? The word "gloomy"? The word "assassination"?

```
# YOUR CODE GOES HERE
```

- **3b.** How many words did Shakespeare use just once? Twice? At least 10 times? More than 100 times?

```
# YOUR CODE GOES HERE
```

- **3c.** Sort `shakespeare.wordtab` so that its entries (counts) are in decreasing order, and save the result as `shakespeare.wordtab.sorted`. Print the 25 most commonly used words, along with their counts. What is the most common word? Second and third most common words?

```
# YOUR CODE GOES HERE
```

- **3d.** What you should have seen in the last question is that the most common word is the empty string `""`. This is just an artifact of splitting `shakespeare.all` by spaces, using `strsplit()`. Redefine `shakespeare.words` so that all empty strings are deleted from this vector. Then recompute

`shakespeare.wordtab` and `shakespeare.wordtab.sorted`. Check that you have done this right by printing out the new 25 most commonly used words, and verifying (just visually) that is overlaps with your solution to the last question.

```
# YOUR CODE GOES HERE
```

- **3e.** As done at the end of the lecture notes, produce a plot of the word counts (y-axis) versus the ranks (x-axis) in `shakespeare.wordtab.sorted`. Set `xlim=c(1,1000)` as an argument to `plot()`; this restricts the plotting window to just the first 1000 ranks, which is helpful here to see the trend more clearly. Do you see **Zipf's law** in action, i.e., does it appear that Frequency $\approx C(1/\text{Rank})^a$ (for some C, a)? **Challenge**: either programmatically, or manually, determine reasonably-well-fitting values of C, a for the Shakespeare data set; then draw the curve $y = C(1/x)^a$ on top of your plot as a red line to show how well it fits.

```
# YOUR CODE GOES HERE
```

Q4. A tiny bit of regular expressions

- **4a.** There are a couple of issues with the way we've built our words in `shakespeare.words`. The first is that capitalization matters; from Q3c, you should have seen that "and" and "And" are counted as separate words. The second is that many words contain punctuation marks (and so, aren't really words in the first place); to see this, retrieve the count corresponding to "and," in your word table `shakespeare.wordtab`.

  The fix for the first issue is to convert `shakespeare.all` to all lower case characters. Hint: recall `tolower()` from Q1b. The fix for the second issue is to use the argument `split="[[:space:]]|[[:punct:]]"` in the call to `strsplit()`, when defining the words. In words, this means: *split on spaces or on punctuation marks* (more precisely, it uses what we call a **regular expression** for the `split` argument). Carry out both of these fixes to define new words `shakespeare.words.new`. Then, delete all empty strings from this vector, and compute word table from it, called `shakespeare.wordtab.new`.

```
# YOUR CODE GOES HERE
```

- **4b.** Compare the length of `shakespeare.words.new` to that of `shakespeare.words`; also compare the length of `shakespeare.wordtab.new` to that of `shakespeare.wordtab`. Explain what you are observing.

```
# YOUR CODE GOES HERE
```

- **4c.** Compute the unique words in `shakespeare.words.new`, calling the result `shakespeare.words.new.unique`. Then repeat the queries in Q2e and Q2f on `shakespeare.words.new.unique`. Comment on the histogram—is it different in any way than before? How about the top 5 longest words?

```
# YOUR CODE GOES HERE
```

- **4d.** Sort `shakespeare.wordtab.new` so that its entries (counts) are in decreasing order, and save the result as `shakespeare.wordtab.sorted.new`. Print out the 25 most common words and their counts, and compare them (informally) to what you saw in Q3d. Also, produce a plot of the new word counts, as you did in Q3e. Does Zipf's law look like it still holds?

```
# YOUR CODE GOES HERE
```

Q5. Where are Shakespeare's plays, in this massive text?

- **5a.** Let's go back to `shakespeare.lines`. Take a look at lines 19 through 23 of this vector: you should see a bunch of spaces preceding the text in lines 21, 22, and 23. Redefine `shakespeare.lines` by setting it equal to the output of calling the function `trimws()` on `shakespeare.lines`. Print out lines 19 through 23 again, and describe what's happened.

```
# YOUR CODE GOES HERE
```

- **5b.** Visit http://www.stat.cmu.edu/~ryantibs/statcomp/data/shakespeare.txt in your web browser and just skim through this text file. Near the top you'll see a table of contents. Note that "THE SONNETS" is the first play, and "VENUS AND ADONIS" is the last. Using `which()`, find the indices of the lines in `shakespeare.lines` that equal "THE SONNETS", report the index of the *first* such occurence, and store it as `toc.start`. Similarly, find the indices of the lines in `shakespeare.lines` that equal "VENUS AND ADONIS", report the index of the *first* such occurence, and store it as `toc.end`.

```
# YOUR CODE GOES HERE
```

- **5c.** Define `n = toc.end - toc.start + 1`, and create an empty string vector of length `n` called `titles`. Using a `for()` loop, populate `titles` with the titles of Shakespeare's plays as ordered in the table of contents list, with the first being "THE SONNETS", and the last being "VENUS AND ADONIS". Print out the resulting `titles` vector to the console. Hint: if you define the counter variable `i` in your `for()` loop to run between 1 and `n`, then you will have to index `shakespeare.lines` carefully to extract the correct titles. Think about the following. When `i=1`, you want to extract the title of the first play in `shakespeare.lines`, which is located at index `toc.start`. When `i=2`, you want to extract the title of the second play, which is located at index `toc.start + 1`. And so on.

```
# YOUR CODE GOES HERE
```

- **5d.** Use a `for()` loop to find out, for each play, the index of the line in `shakespeare.lines` at which this play begins. It turns out that the *second* occurence of "THE SONNETS" in `shakespeare.lines` is where this play actually begins (this first ocurrence is in the table of contents), and so on, for each play title. Use your `for()` loop to fill out an integer vector called `titles.start`, containing the indices at which each of Shakespeare's plays begins in `shakespeare.lines`. Print the resulting vector `titles.start` to the console.

```
# YOUR CODE GOES HERE
```

- **5e.** Define `titles.end` to be an integer vector of the same length as `titles.start`, whose first element is the second element in `titles.start` minus 1, whose second element is the third element in `titles.start` minus 1, and so on. What this means: we are considering the line before the second play begins to be the last line of the first play, and so on. Define the last element in `titles.end` to be the length of `shakespeare.lines`. You can solve this question either with a `for()` loop, or with proper indexing and vectorization. **Challenge**: it's not really correct to set the last element in `titles.end` to be length of `shakespeare.lines`, because there is a footer at the end of the Shakespeare data file. By looking at the data file visually in your web browser, come up with a way to programmatically determine the index of the last line of the last play, and implement it.

```
# YOUR CODE GOES HERE
```

- **5f.** In Q5d, you should have seen that the starting index of Shakespeare's 38th play "THE TWO NOBLE KINSMEN" was computed to be `NA`, in the vector `titles.start`. Why? If you run `which(shakespeare.lines == "THE TWO NOBLE KINSMEN")` in your console, you will see that there is only one occurence of "THE TWO NOBLE KINSMEN" in `shakespeare.lines`, and this occurs in the table of contents. So there was no second occurence, hence the resulting `NA` value.

  But now take a look at line 118,463 in `shakespeare.lines`: you will see that it is "THE TWO NOBLE KINSMEN:", so this is really where the second play starts, but because of colon ":" at the end of the string, this doesn't exactly match the title "THE TWO NOBLE KINSMEN", as we were looking for. The advantage of using the `grep()` function, versus checking for exact equality of strings, is that `grep()` allows us to match substrings. Specifically, `grep()` returns the indices of the strings in a vector for which a substring match occurs, e.g.,

```
grep(pattern="cat",
     x=c("cat", "canned goods", "batman", "catastrophe", "tomcat"))
```

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

so we can see that in this example, `grep()` was able to find substring matches to "cat" in the first, fourth, and fifth strings in the argument `x`. Redefine `titles.start` by repeating the logic in your solution to Q5d, but replacing the `which()` command in the body of your `for()` loop with an appropriate call to `grep()`. Also, redefine `titles.end` by repeating the logic in your solution to Q5e. Print out the new vectors `titles.start` and `titles.end` to the console—they should be free of `NA` values.

```
# YOUR CODE GOES HERE
```

Q6. Extracting and analysing a couple of plays

- **6a.** Let's look at two of Shakespeare's most famous tragedies. Programmatically find the index at which "THE TRAGEDY OF HAMLET, PRINCE OF DENMARK" occurs in the `titles` vector. Use this to find the indices at which this play starts and ends, in the `titles.start` and `titles.end` vectors, respectively. Call the lines of text corresponding to this play `shakespeare.lines.hamlet`. How many such lines are there? Do the same, but now for the play "THE TRAGEDY OF ROMEO AND JULIET", and call the lines of text corresponding to this play `shakespeare.lines.romeo`. How many such lines are there?

```
# YOUR CODE GOES HERE
```

- **6b.** Repeat the analysis, outlined in Q4, on `shakespeare.lines.hamlet`. (This should mostly just involve copying and pasting code as needed.) That is, to be clear: * collapse `shakespeare.lines.hamlet` into one big string, separated by spaces; * convert this string into all lower case characters; * divide this string into words, by splitting on spaces or on punctuation marks, using `split="[[:space:]]|[[:punct:]]"` in the call to `strsplit()`; * remove all empty words (equal to the empty string `""`), and report how many words remain; * compute the unique words, report the number of unique words, and plot a histogram of their numbers of characters; * report the 5 longest words; * compute a word table, and report the 25 most common words and their counts; * finally, produce a plot of the word counts verus rank.

```
# YOUR CODE GOES HERE
```

- **6c.** Repeat the same task as in the last part, but on `shakespeare.lines.romeo`. (Again, this should just involve copying and pasting code as needed. P.S. Isn't this getting tiresome? You'll be happy when we learn functions, next week!) Comment on any similarities/differences you see in the answers.

```
# YOUR CODE GOES HERE
```

- **Challenge.** Using a `for()` loop and the `titles.start`, `titles.end` vectors constructed above, answer the following questions. What is Shakespeare's longest play (in terms of the number of words)? What is Shakespeare's shortest play? In which play did Shakespeare use his longest word (in terms of the number of characters)? Are there any plays in which "the" is not the most common word?

```
# YOUR CODE GOES HERE
```