Strings in R and Interesting problems using them

Kevin Lee

11/24/2017

Introduction

Data can come in many forms. In R, there are numerics that represent numbers, logic that represent boolean values, characters that represent letters and words. When people think of working with data or statistics in general, only numerical data comes to mind. In this post, we will be focusing on strings. First, what is a string? A string in R is a character variable that contains one or more characters. This idea is easy enough to grasp as you can think of a string as a 'word' and a character as a 'letter'. In R, akin to other programming languages, the user can simply create a varaible to hold the whole string. In high-level languages such as C, the string has to be thought of as an array of characters. Both illustrate what a string is with a slight difference in how strings are used in the language. The stringr package allows users to do more with less stress. In other programming languages, string manipulation is common and many companies test applicants' knowledge and mastery of the language with string operation problems.

We will be working with the stringr package and see some example string manipulation problems.

Background

The 'stringr' package is part of the tidyverse set of packages. However, it is not included in the core tidyverse (ex. `library(tidyverse)`) as not all datasets and users dealing with these datasets will need to deal with string manipulation and operation. The 'stringr' package created by Hadley Wickham who also created other open source R packages which make data science accessible and intuitive. And 'stringr' does just that for strings in R. R has existing functionality and operations for strings, but the given string operations are difficult to use for unexperienced users. The 'stringr' package allows

- uniform processing of characters and factors
- constant names and arguments for functions used in stringr
- eliminates unnecessary work in R for the same result
- ensures outputs can be used as inputs in stringr
- takes into account other programming languages' string functions

Instruction / Examples

Stringr Functions

First, we install package stringr and load the package to our current working session.

```r
# install stringr
install.packages("stringr")
```

```r
# load stringr
library(stringr)
```

Now to refresh on some of the basic functions in the stringr package. Some existing base R parallels will be noted.

str_length

The length of the string is obtained with `str_length()`

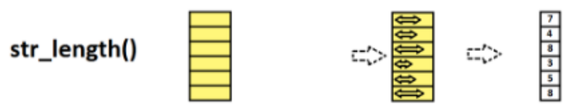
```r
# count the length of the string
str_length("hello world!")
```

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

This works similar to `nchar()`

```r
# count the characters
nchar("hello world!")
```

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



str_c

To concatenate strings, `str_c` helps! The separator is default to an empty string.

```r
# put hello world! together
str_c("hello", " ", "world!")
```

```
## [1] "hello world!"
```

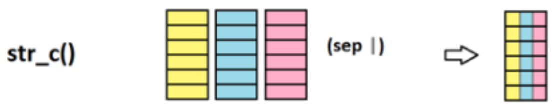
```
str_c("hello", "world!", sep = " ")
```

```
## [1] "hello world!"
```

This works similar to `paste()`

```
# put hello world! together
paste("hello", "world!")
```

```
## [1] "hello world!"
```



str_sub

To get substrings from a string, `str_sub()` is used.

```
# get the first word
str_sub("hello world!", 1, 5)
```

```
## [1] "hello"
```

```
# get the second word
str_sub("hello world!", 7)
```

```
## [1] "world!"
```

```
x <- c("hello", "world!")
# get 5 length substrings in each string
str_sub(x, 1, 5)
```

```
## [1] "hello" "world"
```

With stringr, ```str_sub``` can also be used to replace the selected substring with a string of your choice. When I found this out, I was actually pretty surprised. This is great.

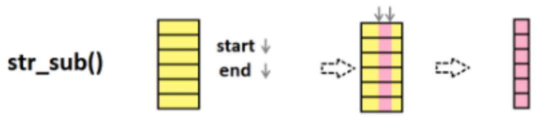
```
# rewrite string
x <- "hello world!"
str_sub(x, 7, -1) <- "you!"
x
```

```
## [1] "hello you!"
```

Quick note of `str_dub`. This function can duplicate strings repeatedly. In base R, this is similar to `rep` or `strrep`.

```
# shout our hello 3 times
str_dup("hello world!", 3)
```

```
## [1] "hello world!hello world!hello world!"
```



str_split

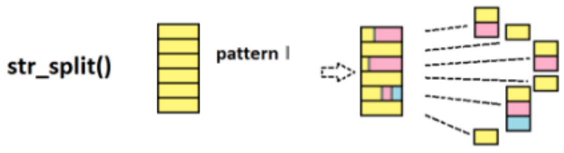
`str_sub` can select a substring of the string. However, if we needed multiple substrings of a string, we would need to use a for loop or do some sort of iteration. If the string has a separator, we can use `str_split` to obtain all the members of the string in a vector.

```
# lets split up hello-world!
str_split("hello-world!", "-")
```

```
## [[1]]
## [1] "hello"  "world!"
```

```
# what if the separator was a letter
str_split("hello-world!", "l")
```

```
## [[1]]
## [1] "he"     ""       "o-wor" "d!"
```



str_pad

In some cases, we might need to add whitespace to the string. Some examples of when this would be useful is when printing results in a specific user-friendly format. Or in datasets with fixed-width cells or that requires a certain length.

```
# pad to the left 1 white space
str_pad("hello world!", 13)
```

```
## [1] " hello world!"
```

```
x <- "hello world!"
str_pad(x, str_length(x) + 1)
```

```
## [1] " hello world!"
```

```
# pad both sides 2 spaces
str_pad(x, str_length(x) + 4, "both")
```

```
## [1] "  hello world!  "
```

Usage of `str_length` is helpful as the function will not pad strings shorter and make code more readable.

str_trim

Maybe more often, you will need to remove whitespaces instead of adding them to get the string and characters itself. `str_trim` helps with this.

```
# remove the white spaces left and right of the string.
str_trim("    hello world!    ")
```

```
## [1] "hello world!"
```

str_detect

When trying to find a certain string or substring, `str_detect` is a very powerful function. It can detect a given literal string or a pattern in regular expression. Here are some regular expressions used in R - [:alnum:] or [0-9A-Za-z]: alphanumeric characters - [:alpha:] or [A-Za-z]: alphabetic characters - [:digit:] or [0-9]: digits 0 1 2 3 4 5 6 7 8 9 - ^ : empty string at beginning of line - $: empty string at end of line - ? : preceding item is optional and will be matched at most once - * : preceding item matched zero or more times - + : preceding item will be matched one or more times - {n} : preceding item is matched exactly n times - {n,} : matched n or more times

```
x <- c("hello", "world", "hellw", "&^#%$^@#%!", "hellohello", "hello!")
# find o
str_detect(x, "o")
```

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

```
# find o at the end
str_detect(x, "o$")
```

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

```
# find w at the start
str_detect(x, "^w")
```

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

```
# find ll
str_detect(x, "ll")
```

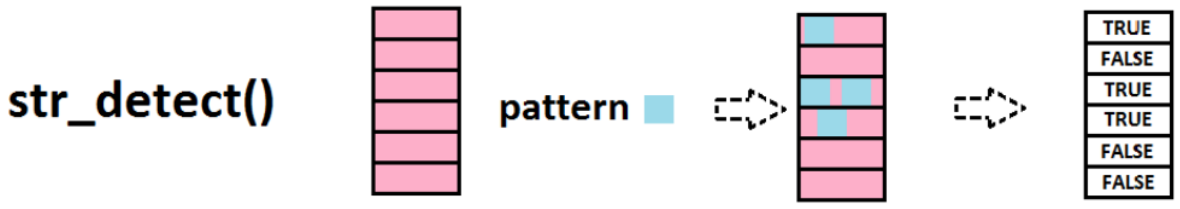
```
## [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE
```

```
# see pattern of alpha
str_detect(x, "[:alnum:]")
```

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

```
# find exactly hello
str_detect(x, "\\bhello\\b")
```

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



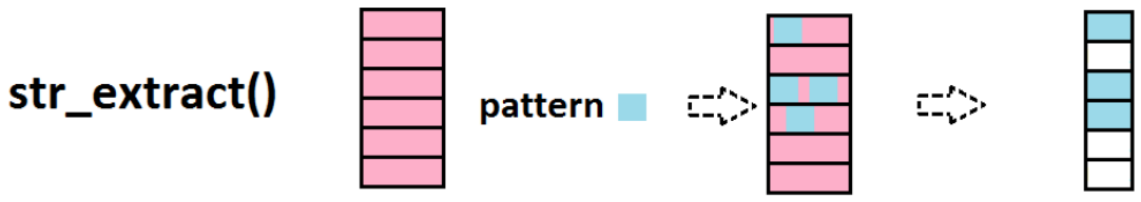
`str_extract` extracts the string from the first match. Use `str_extract_all` to extract all the matched patterns.

```
# extract only alphanumeric
str_extract(x, "[:alnum:]{1,}")
```

```
## [1] "hello"      "world"      "hellw"      NA           "hellohello"
## [6] "hello"
```

```
# extract only hello
str_extract(x, "hello")
```

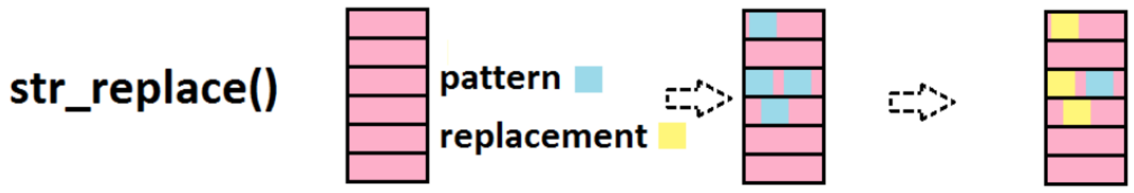
```
## [1] "hello" NA      NA      NA      "hello" "hello"
```



`str_replace` replaces the matched pattern. Use `str_replace_all` to replace all the matched patterns.

```
# replace first hello with goodbye
str_replace(x, "hello", "goodbye")
```

```
## [1] "goodbye"      "world"        "hellw"        "&^#%$^@#%!"
## [5] "goodbyehello" "goodbye!"
```



`str_count` returns the number of matches to the pattern in the string.

```
x <- c("a,b,c", "d,e,f,g", "i love a b c", "1,2,3", "1 2 3 45 ")
comma <- "([a-z][,]{1,})"
# find how many matches of the pattern a-z with a comma following
str_count(x, comma)
```

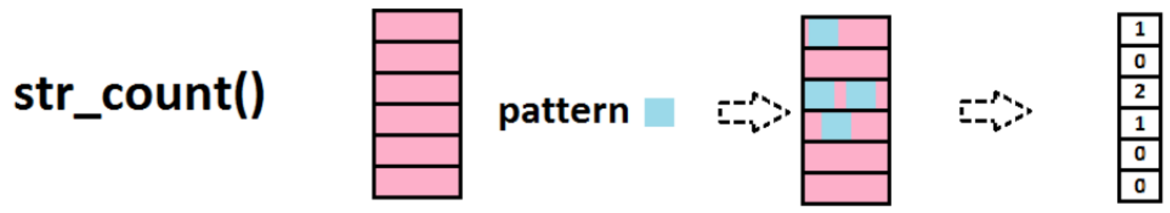
```
## [1] 2 3 0 0 0
```

```
spaces <- "([a-z]{1,}[ ])"
# find how many matches of pattern a-z with space
str_count(x, spaces)
```

```
## [1] 0 0 4 0 0
```

```
spaces <- "([0-9]{1,}[ ]{1})"
# find how many matches of pattern 0-9 with space
str_count(x, spaces)
```

```
## [1] 0 0 0 0 4
```



String Problems

Let's try some string problems to fully master and think critically about string manipulation and operations. After each one, try thinking of what I could have improved on. And then try to improve it yourself! Kind of like a stackoverflow thread.

CamelCase

What if we want to know the number of words in a string where the string is in CamelCase (the first word is lowercase and the rest have their first letter uppercase and rest lowercase).

```
# let's count the words of this camelcase
x <- "helloWorldWow"
# extract the first word
first <- str_extract(x, "^[a-z]{0,}")
# count the rest
str_count(x, "([A-Z]{1}[a-z]{0,})") + length(first)
```

```
## [1] 3
```

```
x <- "bwowWowWow"
first <- str_extract(x, "^[a-z]{0,}")
str_count(x, "([A-Z]{1}[a-z]{0,})") + length(first)
```

```
## [1] 3
```

Here is one solution for correct camelcase strings! The thought process was since the first word is the only one that's different than the rest, we should check/count that first before calling `str_count` on the rest of the string. Do you see any issues with this solution? Can you think of a better one?

Cipher

A Caesar Cipher is an encryption where every letter is rotated by a fixed number. Create it in R!

```
# letter dictionary
letters <- c('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', '
u', 'v', 'w', 'x', 'y', 'z')
# the message
message <- "coded message"
# how many rotations for the cipher
rotation <- 27
# replace each letter with the cipher letter
for (i in 1:str_length(message)) {
  letter <- str_sub(message, i, i)
  if (letter != " ") {
    index <- which(!is.na(str_locate(letters, letter)))[1]
    str_sub(message, i, i) <- letters[(index+rotation) %% 26]
  }
}
# output the message
message
```

```
## [1] "dpefe nfttbhf"
```

This is known as one of the first cryptography. Caesar cipher takes in a regular message and encrypts it by rotating each letter left or right by some number of positions. Above I created the algorithm for left rotation Caeser Cipher.

SOS!

You obtain a string from a vessel that is sending out SOS signals repeatedly. However, the message got corrupted in transit so some of the letters in the "SOS" are switched. How many of the SOS's are corrupted?

```
# message obtained
message <- "SOESQSSOESOSSOSTOT"
# total - good SOS
str_length(message)/3 - str_count(message, "SOS")
```

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

`str_count` makes this problem easy as we can just count how many instances "SOS" appears and subtract it from the total for the corrupted ones.

Limit 10

You obtain a string of numbers. Separate the string so that each string does not exceed 10 when added and each string must be as high as possible up to 10.

```
# string of numbers
message <- "1578215709"
# holds our separated strings
separated <- c()
# temp variable
entry <- ""
# sum for comparison
sum <- 0
for (i in 1:str_length(message)) {
  # obtain the number
  number <- as.numeric(str_sub(message, i, i))
  # if less than 10, append to temp var
  if (sum + number < 10) {
    sum <- sum + number
    entry <- str_c(entry, number)
  } else {
    # if above 10, commit to separated
    separated <- c(separated,entry)
    sum <- number
    entry <- ""
    entry <- str_c(entry, number)
    if (i == str_length(message)) {
      separated <- c(separated,entry)
    }
  }
}
separated
```

```
## [1] "15"  "7"   "8"   "215" "70"  "9"
```

I employ a similar tactic to the cipher algorithm. This is a simple for loop iterative solution where the string must be converted to numeric for the summation and then converted back.

Conclusion and Reflection

In conclusion, the string is a pretty simple concept. However, if you truly wanted to explore, there are tons of mind-tickling string manipulation problems that test your mastery of programming in a lot of aspects. The stringr package makes it extremely easy to work with strings just like ggplot2 does for data visualization. If you had fun with the problems in this post, try some more! There are a ton of problems to go through online with a simple google search away.

References

- Wickham, Hadley. Stringr: Modern, consistent String Processing. 2017, https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf. Accessed 24 Nov. 2017.
- Cran R Project. 2017, https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html. Accessed 24 Nov. 2017.
- Vaudor, Lise. Stringr Visual Cheat Sheet. 2017, https://maraaverick.rbind.io/2017/08/stringr-visual-cheat-sheet/. Accessed 24 Nov. 2017.
- Character strings in R. R-bloggers. 2017, https://www.r-bloggers.com/character-strings-in-r/. Accessed 24 Nov. 2017.
- Wickham, Hadley. Cran R Project. 2017, https://cran.r-project.org/web/packages/stringr/stringr.pdf. Accessed 24 Nov. 2017.
- HackerRank. 2017, https://www.hackerrank.com/domains/algorithms/strings. Accessed 24 Nov. 2017.
- R for Data Science. 2017, http://r4ds.had.co.nz/strings.html. Accessed 24 Nov. 2017.