Post 2 - Introduction to stringr

Michael Zhang

12/3/2017

A Brief Introduction to Stringr Package

Intro: Why stringr?

There are four main families of functions in stringr:

- **Character manipulation:** these functions allow you to manipulate the individual characters inside the strings inside character vectors.
- **Whitespace tools:** to add, remove, and manipulation whitespace.
- **Locale sensitive operation:** whose operation will vary for locale to locale
- **Pattern matching functions:** These recognise four engines of pattern description. The most common is regular expresssions, but there are a three other tools.

Before we go any further, you should install stringr package in R by running `install.packages(stringr)`

In the following sections, I'm going to introduce these families and some pattern engines in stringr package.

I. Character Manipulation Functions

This family of functions manipulates specific characters in a string.

a. str_length()

This function checks the number of characters in a string, which serves the same purpose as `nchar()`. For example:

```
library(stringr)
str_length("abcd")
```

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

b. sub_str()

This function allows us to access individual characters in a string. `sub_str()` takes 3 arguments: a character vector, a starting position and an end position. If the position is negative, then it's going to be counted from the right. Also, the position is inclusive. For example:

```
x <- c("abcdef", "ghifjk")

# the third character in each element in x
str_sub(x, 3, 3)
```

```
## [1] "c" "i"
```

```
# the third to the thrid to last charactor in each element in x
str_sub(x, 3, -3)
```

```
## [1] "cd" "if"
```

c. str_sub()

This function is similar as the previous one. Instead of showing the characters, `str_sub()` changes the selected characters to a new input. For example:

```
# change the third character in each element to character "X"
str_sub(x, 3, 3) <- "X"
x
```

```
## [1] "abXdef" "ghXfjk"
```

II. Whitespace Tools

This family of functions manipulates the whitespace in strings. *(i.e. add, delete and move whitespace)*

a. str_pad()

This function makes a string to a determined length by adding whitespace to it. User can choose to add space on the left, right or both sides of the string. For example:

```r
x <- c("abc", "defghi")

# the default is to add space on the left
str_pad(x, 10)
```

```
## [1] "       abc" "    defghi"
```

```r
# specify the position of the space to be added in the third argument
str_pad(x, 10, "right")
```

```
## [1] "abc       " "defghi    "
```

```r
str_pad(x, 10, "both")
```

```
## [1] "   abc    " "  defghi  "
```

One thing worth mentioning is that if the determined length is shorter than the original length of a string, no space will be added, and the original string will be kept.

b. str_trunc()

This function truncate long strings to determined length, for example:

```r
str_trunc("This is sentence is longer than 10 characters", 10)
```

```
## [1] "This is..."
```

c. str_trim()

While `str_pad()` adds space to a string, `str_trim()` trims off the space in a string. User is able to choose to trim the space on the left, right, or as default, both sides. For example:

```r
x <- c("  a   ", "b   ", "   c")

# the function trims off the space on the both sides of a string as default
str_trim(x)
```

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

```r
# specify the side in the second argument
str_trim(x, "left")
```

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

```r
str_trim(x, "right")
```

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

d. str_wrap()

This function wraps a paragraph and set the width of it to a determined length. For example:

```r
x <- str_c("This is the first string.",
           "This is the second string.",
           "This is the third string.",
           "This is the fourth string.",
           "Make this as a paragragh with a width of 20 characters.")

cat(str_wrap(x, 20))
```

```
## This is the first
## string.This is the
## second string.This
## is the third
## string.This is the
## fourth string.Make
## this as a paragragh
## with a width of 20
## characters.
```

III. Locale Sensitive Operations:

This family of functions will vary from locale to locale. *(i.e. depends on location in the world, etc.)*

a. Case Transformation Functions

These functions change the characters in a string to upper, lower or other sepcified cases. For example:

```
x <- "This is an example."

# to all cap
str_to_upper(x)
```

```
## [1] "THIS IS AN EXAMPLE."
```

```
# to all lower case
str_to_lower(x)
```

```
## [1] "this is an example."
```

```
# Format x as a title (i.e. Cap the first letter for each word)
str_to_title(x)
```

```
## [1] "This Is An Example."
```

```
# Sometimes user can specify the language they are using.
# For instance, there are two kinds of "i" in Turkish.
# We can specify the language in the second argument.
# "tr" stands for Turkish in this case.
str_to_lower(x, "tr")
```

```
## [1] "this is an example."
```

b. String Ordering and Sorting

These functions order and sort characters in a string. For example:

```
x <- c("y", "i", "k")

# Order the letters according to the alphabet.
# Note the output stands for the number of elements in x.
str_order(x)
```

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

```
# Order the letters according to the alphabet.
# Note unlike the previous function,
# the output shows the reordered x directly.
str_sort(x)
```

```
## [1] "i" "k" "y"
```

```
# In Lithuanian, y comes between i and k.
# We can specify the language in the second argument.
str_sort(x, locale = "lt")
```

```
## [1] "i" "y" "k"
```

IV. Pattern Matching Functions

Most functions in Stringr work with patterns: they find, match or manimulate patterns. This section will introduce some pattern matching functions that **detect, locate, extract, match, replace, and split** strings, and we will learn more about common pattern matching expression in next section.

First, we introduce a vector with some strings, and a format of phone numbers:

```
x <- c(
  "apple",
  "219 733 8965",
  "329-293-8753",
  "Work: 579-499-7527; Home: 543.355.3679"
)
phone <- "([2-9][0-9]{2})[- .]([0-9]{3})[- .]([0-9]{4})"
```

Now, we want to manupulate this string with some determined patterns.

a. Detect

We can use `str_detect()` to see which string in the vector matches the pattern. Also, we can use `str_subset()` to return those strings directly. `str_count()` counts the number of matches in a string. For example:

```
# To see which string contains the pattern of "phone"
str_detect(x, phone)
```

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

```
# To return those matching strings
str_subset(x, phone)
```

```
## [1] "219 733 8965"
## [2] "329-293-8753"
## [3] "Work: 579-499-7527; Home: 543.355.3679"
```

```
# To return the number of matches in each string
str_count(x, phone)
```

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

b. Locate

`str_locate()` locates the first position of a pattern and returns a numeric matrix with columns start and end. Notice that even there might be some characters between two matches that don't match the patterns, this function will include the number of those characters.

`str_locate_all()` locates all matches, returning a list of numeric matrices. Unlike the first function, this function won't count those unmatched characters between two matches. For example:

```
# This function will include the numeber of unmatched characters
str_locate(x, phone)
```

```
##      start end
## [1,]    NA  NA
## [2,]     1  12
## [3,]     1  12
## [4,]     7  18
```

```
# This function will exclude the number of unmatched characters
str_locate_all(x, phone)
```

```
## [[1]]
##      start end
##
## [[2]]
##      start end
## [1,]     1  12
##
## [[3]]
##      start end
## [1,]     1  12
##
## [[4]]
##      start end
## [1,]     7  18
## [2,]    27  38
```

c. Extract

`str_extract()` extracts the matching text, returning a character **vector**. `str_extract_all()` extracts all matches and returns a **list** of character vectors. For example:

```
str_extract(x, phone)
```

```
## [1] NA             "219 733 8965" "329-293-8753" "579-499-7527"
```

```
# use "simplify = T" to simplify the structure of the list
str_extract_all(x, phone, simplify = TRUE)
```

```
##      [,1]           [,2]
## [1,] ""             ""
## [2,] "219 733 8965" ""
## [3,] "329-293-8753" ""
## [4,] "579-499-7527" "543.355.3679"
```

d. Match

`str_match()` extracts capture groups formed by () from the first match. It returns **a character matrix** with one column for the complete match and one column for each group.

`str_match_all()` extracts capture groups from all matches and returns **a list of character matrices**. The difference between these two might sound confusing, but everything should be clear after seeing the following example:

```
# the output is a matrix
str_match(x, phone)
```

```
##      [,1]           [,2]  [,3]  [,4]
## [1,] NA             NA    NA    NA
## [2,] "219 733 8965" "219" "733" "8965"
## [3,] "329-293-8753" "329" "293" "8753"
## [4,] "579-499-7527" "579" "499" "7527"
```

```
# the output is list of matrices
str_match_all(x, phone)
```

```
## [[1]]
##      [,1] [,2] [,3] [,4]
##
## [[2]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "219 733 8965" "219" "733" "8965"
##
## [[3]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "329-293-8753" "329" "293" "8753"
##
## [[4]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "579-499-7527" "579" "499" "7527"
## [2,] "543.355.3679" "543" "355" "3679"
```

e. Replace

`str_replace()` replaces the first matched pattern and returns a character vector. `str_replace_all()` replaces all matches. For example:

```
# notice that the second phone number in the last string is not replaced
str_replace(x, phone, "XXX-XXX-XXXX")
```

```
## [1] "apple"
## [2] "XXX-XXX-XXXX"
## [3] "XXX-XXX-XXXX"
## [4] "Work: XXX-XXX-XXXX; Home: 543.355.3679"
```

```
# notice that every phone number is replaced
str_replace_all(x, phone, "XXX-XXX-XXXX")
```

```
## [1] "apple"
## [2] "XXX-XXX-XXXX"
## [3] "XXX-XXX-XXXX"
## [4] "Work: XXX-XXX-XXXX; Home: XXX-XXX-XXXX"
```

f. Split

`str_split()` splits a string into a variable number of pieces and returns a list of character vectors.

`str_split_fixed()` splits the string into a fixed number of pieces based on a pattern and returns a character matrix.

```
# everything separated by "," is going to be splitted apart
str_split("a,b,c,d,e", ",")
```

```
## [[1]]
## [1] "a" "b" "c" "d" "e"
```

```
# the string will be splitted into determined number of parts
# the last part won't be splitted even it contains the pattern
str_split_fixed("a,b,c,d,e", ",", n = 3)
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c,d,e"
```

V. Engines

There are four main engines that stringr can use to describe patterns:

- **Regular expressions** (default)
- **Fixed bytewise matching** (`fixed()`)
- **Locale-sensitve character matching** (`coll()`)
- **Text boundary analysis** (`boundary()`)

a. Regular Expressions

There are many regular expressions to manipulate a string based on a determined pattern. I'm going to give an example for each catagory:

1. Basic matches

```
x <- c("banana", "Banana", "BANANA")
str_detect(x, "banana")
```

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

```
# we can use "." to represent any character
str_extract(x, ".a.")
```

```
## [1] "ban" "Ban" NA
```

2. Escaping

As shown in the previous example, `.` can be used to represent any characters. However, when we need to represent a dot itself, we have to use `\`, which is called escaping. One problem is that, `\` itself is also a character, so we have to use `\\.` in order to represent a dot.

```
# here "\\." means a literal dot
str_extract(c("abc", "a.c", "bef"), "a\\.c")
```

```
## [1] NA    "a.c" NA
```

In order to represent a literal `\`, we also need to use another `\` to escape. To make that expression into a string, we need another `\`.

```
str_extract("a\\b", "\\\\")
```

```
## [1] "\\"
```

3. Special characters

Some characters might be hard to type. You can specify individual unicode characters in these ways:

- : 2 hex digits.
- : 1-6 hex digits.
- : 4 hex digits.
- : 8 hex digits.
-

You can also specify many common control characters:

- : bell.
- : match a control-X character.
- : escape (01B).
- : form feed (00C).
- : line feed (00A).
- : carriage return (00D).
- : horizontal tabulation (009).
- ooo match an octal character.

4. Matching multiple characters

```
# let x be an "a" with accent on top
x <- "a\u0301"

# this will return a character "a"
str_extract(x, ".")
```

```
## [1] "a"
```

```
# this will return a literal "a" with accent on top
str_extract(x, "\\x")
```

```
## [1] "á"
```

5. Alternation

`|` is the alternation operator, which will pick between one or more possible matches. For example:

```
str_detect(c("abc", "def", "ghi"), "abc|def")
```

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

6. Grouping

You can use parentheses to override the default precedence rules:

```
str_extract(c("grey", "gray"), "gr(e|a)y")
```

```
## [1] "grey" "gray"
```

You can use (?:…), the non-grouping parentheses, to control precedence but not capture the match in a group. This is slightly more efficient than capturing parentheses:

```
str_match(c("grey", "gray"), "gr(?:e|a)y")
```

```
##      [,1]
## [1,] "grey"
## [2,] "gray"
```

7. Anchors

By default, regular expressions will match any part of a string. It's often useful to anchor the regular expression so that it matches from the start or end of the string:

- ^ matches the start of string.
- $ matches the end of the string.

For example:

```
x <- c("apple", "banana", "pear")

# check the start of each string in x
str_extract(x, "^a")
```

```
## [1] "a" NA  NA
```

```
# check the end of each string in x
str_extract(x, "a$")
```

```
## [1] NA  "a" NA
```

8. Repetition

You can control how many times a pattern matches with the repetition operators:

- ?: 0 or 1.
- +: 1 or more.
- *: 0 or more.

You can also specify the number of matches precisely:

- {n}: exactly n
- {n,}: n or more
- {n,m}: between n and m

For example:

```
x <- "MDCCCLXXXVIII"

str_extract(x, "CC?")
```

```
## [1] "CC"
```

```
str_extract(x, "CC+")
```

```
## [1] "CCC"
```

```
str_extract(x, 'C[LX]+')
```

```
## [1] "CLXXX"
```

```
str_extract(x, "C{2}")
```

```
## [1] "CC"
```

```
str_extract(x, "C{2,}")
```

```
## [1] "CCC"
```

```
str_extract(x, "C{2,3}")
```

```
## [1] "CCC"
```

b. Fixed Bytewise Matching

This method check string bytewise. Sometimes characters might look the same but have different definitions. For instance, “á” can be defined either as a single character or as an “a” plus an accent. When we check them bytewise, they are different:

```
a1 <- "\u00e1"
a2 <- "a\u0301"

a1 == a2
```

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

c. Locale-sensitve Character Matching

`coll(x)` looks for a match to x using human-language collation rules, which are differ around the world, so you’ll also need to supply a locale parameter. For example:

```
str_subset(c("I", "İ", "i", "ı"), coll("i", ignore_case = TRUE, locale = "tr"))
```

```
## [1] "İ" "i"
```

d. Text Boundary Analysis

`boundary()` matches boundaries between characters, lines, sentences or words. By convention, `""` is treated as `boundary("character")`. For example:

```
x <- "This is a sentence."

# Split x by word
str_split(x, boundary("word"))
```

```
## [[1]]
## [1] "This"     "is"       "a"        "sentence"
```

```
str_count(x, boundary("word"))
```

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

```
# Split x by characters
str_split(x, "")
```

```
## [[1]]
##  [1] "T" "h" "i" "s" " " "i" "s" " " "a" " " "s" "e" "n" "t" "e" "n" "c"
## [18] "e" "."
```

```
str_count(x, "")
```

```
## [1] 19
```

V. Reference

1. https://cran.r-project.org/web/packages/stringr/stringr.pdf
2. https://github.com/tidyverse/stringr
3. http://stringr.tidyverse.org/articles/stringr.html
4. http://stringr.tidyverse.org/articles/regular-expressions.html
5. http://stringr.tidyverse.org/reference/index.html
6. http://r4ds.had.co.nz/strings.html
7. http://www.mjdenny.com/Text_Processing_In_R.html

VI. Take Home Message

- Stringr package is mainly used to deal with pattern manipulation problems
- One should pay attention to **syntax** issues, which are addressed in **"Escaping"** part in this post
- It's easier to memorize functions by their "functions". As shown in this post, I separated the package into different sections, so it's easier to remember.
- There are many ways to express a single idea using this package, as you can see in **"Special Characters"** and **"Repetition"** sections. It always is a good idea to come back and look up those expressions.

A Side Note to Grader

Please take these points into consideration:

- This post is about working with strings, so there is no graph.
- However, this post provides numerous simple and clear examples, and the layout is way better than the lecture slides.
- Part of this post is beyond the content taught in the class. (e.g. the fourth section **Engines**)