ST720 Data Science

Handling Strings

Seung Jun Shin (sjshin@krea.ac.kr)

Department of Statistics, Korea University

Introduction

- Base **R** contains many functions to work with strings but not convenient at all.
- `stringr` package is more intuitive.

String basics

- String Length

```r
str_length(c("a", "R for data science", NA))
```

```
## [1]  1 18 NA
```

- Combining Strings

```r
str_c("x", "y")
```

```
## [1] "xy"
```

```r
str_c("x", "y", sep=",")
```

```
## [1] "x,y"
```

- Handling Missing

```r
x <- c('abc', NA)
str_c("|-", x, "-|")
```

```
## [1] "|-abc-|" NA
```

```r
str_c("|-", str_replace_na(x),"-|")
```

```
## [1] "|-abc-|" "|-NA-|"
```

Combining Strings

- More complicating.

```
name <- 'Hadley'
time_of_day <- 'morning'
birthday <- FALSE
str_c('Good ', time_of_day," ", name,
      if(birthday) 'and HAPPY BIRTHDAY','.')
```

```
## [1] "Good morning Hadley."
```

- Collapse a vector into a single string

```
str_c(c("x", "y", "z"), collapse= ",")
```

```
## [1] "x,y,z"
```

Subsetting Strings

```
x <- c("Apple", "Banana", "Pear")
str_sub(x,  1,  3)
```

```
## [1] "App" "Ban" "Pea"
```

```
str_sub(x, -3, -1)
```

```
## [1] "ple" "ana" "ear"
```

```
str_sub("a", 1, 5)
```

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

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x,1,1))
x
```

```
## [1] "apple"  "banana" "pear"
```

Locales

```
dog <- "The quick brown dog"
str_to_upper(dog)
str_to_lower(dog)
str_to_title(dog)
```

```
## [1] "THE QUICK BROWN DOG"

## [1] "the quick brown dog"

## [1] "The Quick Brown Dog"
```

- Turkish has tow i's : with and without a dot, and it has a different ruel for captializing them :

```
str_to_upper(c("i", "ı"))
str_to_upper(c("i", "ı"), locale = "tr")
```

```
## [1] "I" "I"

## [1] "İ" "I"
```

Locales

```r
x <- c("apple", "eggplant", "banana")
str_sort(x, locale ='en') # English
```

```
## [1] "apple"    "banana"   "eggplant"
```

```r
str_sort(x, locale = 'haw') # Hawaiian
```

```
## [1] "apple"    "eggplant" "banana"
```

- The locale is specified as an ISO 639 language code, which is a two or three-letter abbreviation.

Regulars Expressions

Regexps

Regexps are a very terse language that allow you to describe patterns in strings.

메타문자	기능	설명
.	문자	1개의 문자와 일치한다. 단일행 모드에서는 새줄 문자를 제외한다.
[ ]	문자 클래스	"["과 "]" 사이의 문자 중 하나를 선택한다. "¦"를 여러 개 쓴 것과 같은 의미이다. 예를 들면 [abc]d는 ad, bd, cd를 뜻한다. 또한, "-" 기호와 함께 쓰면 범위를 지정할 수 있다. "[a-z]"는 a부터 z까지 중 하나, "[1-9]"는 1부터 9까지 중의 하나를 의미한다.
[^]	부정	문자 클래스 안의 문자를 제외한 나머지를 선택한다. 예를 들면 [^abc]d는 ad, bd, cd는 포함하지 않고 ed, fd 등을 포함한다. [^a-z]는 알파벳 소문자로 시작하지 않는 모든 문자를 의미한다.
^	처음	문자열이나 행의 처음을 의미한다.
$	끝	문자열이나 행의 끝을 의미한다.
( )	하위식	여러 식을 하나로 묶을 수 있다. "abc¦adc"와 "a(b¦d)c"는 같은 의미를 가진다.
\n	일치하는 n번째 패턴	일치하는 패턴들 중 n번째를 선택하며, 여기에서 n은 1에서 9 중 하나가 올 수 있다.
*	0회 이상	0개 이상의 문자를 포함한다. "a*b"는 "b", "ab", "aab", "aaab"를 포함한다.
{m, n}	m회 이상 n회 이하	"a{1,3}b"는 "ab", "aab", "aaab"를 포함하지만, "b"나 "aaaab"는 포함하지 않는다.

Figure 1: regexp1

Basic Matches

- The simplest patterns match exact strings:

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

apple

banana

pear

- The next step up in complexity is ., which matches any character :

```
str_view(x, ".a.")
```

apple

banana

pear

Basic Matches

- To create the regular expression \. we need the string “\\.”

```r
dot <- "\\."
```

- But the expression itself only contains one:

```r
writeLines(dot)
```

```
## \.
```

- And this tells R to look for an explicit .

```r
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

```
abc
a.c
bef
```

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.
  - ^ to match the start of the string.
  - $ to match the end of the string.

```r
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

apple

banana

pear

```r
str_view(x, "a$")
```

apple

banana

Anchors

- To force a regular expression to only match a complete string, anchor it with both ^ and $:

```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

apple pie

apple

apple cake

```
str_view(x, "^apple$")
```

apple pie

apple

apple cake

Character Classes and Alternatives

- There are a number of special patterns that match more than one character.
  - `\d`: matches any digit.
  - `\s`: matches any whitespace (e.g. space, tab, newline).
  - `[abc]`: matches a, b, or c.
  - `[^abc]`: matches anything except a, b, or c.
- Remember, to create a regular expression containing `\d` or `\s`, you'll need to escape the \ for the string, so you'll type `\\d` or `\\s`.
- A character class containing a single character is a nice alternative to backslash escapes when you want to include a single metacharacter in a regex.

Character Classes and Alternatives

```r
str_view(c("abc", "a.c", "a*c", "a c"), "a[.]c")
```

abc

a.c

a*c

a c

Character Classes and Alternatives

```r
str_view(c("abc", "a.c", "a*c", "a c"), ".[*]c")
```

```
abc
a.c
a*c
a c
```

Character Classes and Alternatives

```r
str_view(c("abc", "a.c", "a*c", "a c"), "a[ ]")
```

abc

a.c

a*c

a c

Character Classes and Alternatives

- You can use *alternation* to pick between one or more alternative patterns.
- For example, abc|d..f will match either “abc”, or “deaf”.

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

grey

gray

Repetition

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

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

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

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

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

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Repetition

- We can also specify the number of matches precisely :
  - `{n}` : exactly n
  - `{n,}` : n or more
  - `{, m}` : at most m
  - `{n,m}` : between n and m

Repetition

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

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

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

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

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

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C{2,3}?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+?')
```

Grouping and Backreferences

- Parentheses also create a numbered capturing group (number 1, 2 etc.).
- A capturing group stores the part of the string matched by the part of the regular expression inside the parentheses.
- You can refer to the same text as previously matched by a capturing group with backreferences, like \1, \2 etc.
- For example, the following regular expression finds all fruits that have a repeated pair of letters.

Grouping and Backreferences

```
str_view(fruit, "(..)\\1", match = TRUE)
```

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

Detect Matches

- To determine if a character vector matches a pattern, use `str_detect`. It returns a logical vector the same length as the input:

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

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

- How many common words strat with t?

```
sum(str_detect(words, "^t"))
```

```
## [1] 65
```

- What proportion of common words end with a vowel?

```
mean(str_detect(words, '[aeiou]$'))
```

```
## [1] 0.2765306
```

Detect Matches

- For example, here are two ways to find all words that don't contain any vowels:

```
no_vowels1 <- !str_detect(words, '[aeiou]')
no_vowels2 <- str_detect(words, '^[^aeiou]+$')
identical(no_vowels1, no_vowels2)
```

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

Detect Matches

- Typically, however, your strings will be one column of a data frame, and you'll want to use `filter` instead:

```
df <- tibble(
  word <- words,
  i = seq_along(words)
)
df %>%
  filter(str_detect(words, "x$"))
```

```
## # A tibble: 4 x 2
##   `word <- words`     i
##   <chr>           <int>
## 1 box               108
## 2 sex               747
## 3 six               772
## 4 tax               841
```

Detect Matches

- A variation on `str_detect()` is `str_count`: rather than a simple yer or no, it tells you how many matches there are in a string:

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

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

- On average, how many vowels per world?

```
mean(str_count(words, "[aeiou]"))
```

```
## [1] 1.991837
```

Extract Matches

- To extract the actual text of a match, use `str_extract()`.
- To show that off, we're going to need a more complicated example. I'm going to use the **Harvard sentences**, which were designed to test VOIP systems, but are also useful for practicing regexes.

```
length(sentences)
```

```
## [1] 720
```

```
head(sentences)
```

```
## [1] "The birch canoe slid on the smooth planks."
## [2] "Glue the sheet to the dark blue background."
## [3] "It's easy to tell the depth of a well."
## [4] "These days a chicken leg is a rare dish."
## [5] "Rice is often served in round bowls."
## [6] "The juice of lemons makes fine punch."
```

Extract Matches

```r
colors <- c("red", "orange","yellow","green","blue","purple")
color_match <- str_c(colors, collapse = "|")
color_match
```

```
## [1] "red|orange|yellow|green|blue|purple"
```

```r
has_color <- str_subset(sentences, color_match)
matches <- str_extract(has_color, color_match)
head(matches)
```

```
## [1] "blue" "blue" "red"  "red"  "red"  "blue"
```

- Note that `str_extract()` only extracts the **first** match.

Extract Matches

- To get all matches, use `str_extract_all()`.

```
more <- sentences[str_count(sentences, color_match) > 1]
str_extract_all(more, color_match)
```

```
## [[1]]
## [1] "blue" "red"
##
## [[2]]
## [1] "green" "red"
##
## [[3]]
## [1] "orange" "red"
```

Grouped Matches

- Imagine we want to extract nouns from the sentences. As a heuristic, we'll look for any word that comes after "a" or "the".
- Defining a "word" in a regular expression is a little tricky, so here I use a simple approximation a sequence of at least one character that isn't a space:

```
noun <- "(a|the) ([^ ]+)"
has_noun <- sentences %>%
  str_subset(noun)%>%
  head(10)
has_noun %>%
  str_extract(noun)
```

```
##  [1] "the smooth" "the sheet"  "the depth"  "a chicken"  "the
##  [6] "the sun"    "the huge"   "the ball"   "the woman"  "a h
```

Grouped Matches

- `str_extract()` gives us the complete match; `str_match()` gives each individual component.

```
has_noun %>%
  str_match(noun)
```

```
##       [,1]          [,2]  [,3]
##  [1,] "the smooth" "the" "smooth"
##  [2,] "the sheet"  "the" "sheet"
##  [3,] "the depth"  "the" "depth"
##  [4,] "a chicken"  "a"   "chicken"
##  [5,] "the parked" "the" "parked"
##  [6,] "the sun"    "the" "sun"
##  [7,] "the huge"   "the" "huge"
##  [8,] "the ball"   "the" "ball"
##  [9,] "the woman"  "the" "woman"
## [10,] "a helps"    "a"   "helps"
```

Grouped Matches

- If your data is in tibble, it's often easier to use `tidyr::extract()`. It works like `str_match()` but requires you to name the matches, which are then plaed in new columns.

```
tibble(sentence = sentences) %>%
  tidyr::extract(
    sentence, c("article", "noun"), "(a|the) ([^ ]+)",
    remove = FALSE
  )
```

```
## # A tibble: 720 x 3
##    sentence                                    article noun
##    <chr>                                       <chr>   <chr>
##  1 The birch canoe slid on the smooth planks.  the     smooth
##  2 Glue the sheet to the dark blue background. the     sheet
##  3 It's easy to tell the depth of a well.      the     depth
##  4 These days a chicken leg is a rare dish.    a       chicke
##  5 Rice is often served in round bowls.        <NA>    <NA>
##  6 The juice of lemons makes fine punch.       <NA>    <NA>
##  7 The box was thrown beside the parked truck. the     parked
##  8 The hogs were fed chopped corn and garbage. <NA>    <NA>
##  9 Four hours of steady work faced us.         <NA>    <NA>
```

Replacing Matches

- `str_replace()` andstr_replace_all()‘ allow you to replace matches with new strings. The simplest use is to replace a pattern with a fixed string:

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
```

```
## [1] "-pple"  "p-ar"   "b-nana"
```

```
str_replace_all(x, "[aeiou]", "-")
```

```
## [1] "-ppl-"  "p--r"   "b-n-n-"
```

Splitting

Use `str_split()` to split a string up into pieces. For example, we could split sentences into words:

```
sentences %>%
  head(5) %>%
  str_split(" ")
```

```
## [[1]]
## [1] "The"     "birch"   "canoe"   "slid"    "on"      "the"     "smooth"
## [8] "planks."
##
## [[2]]
## [1] "Glue"        "the"         "sheet"       "to"          "the"
## [6] "dark"        "blue"        "background."
##
## [[3]]
## [1] "It's"  "easy"  "to"    "tell"  "the"   "depth" "of"    "a"     "well."
##
## [[4]]
## [1] "These"   "days"    "a"       "chicken" "leg"     "is"      "a"
## [8] "rare"    "dish."
##
## [[5]]
## [1] "Rice"   "is"     "often"  "served" "in"     "round"  "bowls."
```

Splitting

- We can also request a maximum number of pieces:

```
fields <- c("Name: Hadley", "Country: NZ", "Age: 35")
fields %>% str_split(": ", n=2, simplify = TRUE)
```

```
##      [,1]      [,2]
## [1,] "Name"    "Hadley"
## [2,] "Country" "NZ"
## [3,] "Age"     "35"
```

- Instead of splitting up strings by patterns, you can also split up by character, line, sentence, and word `boundary()`s:

```
x <- "This is a sentence.  This is another sentence."
str_view_all(x, boundary("word"))
```

This is a sentence. This is another sentence.

```
str_split(x, " ")[[1]]
```

```
## [1] "This"       "is"         "a"          "sentence."  ""
## [7] "is"         "another"    "sentence."
```

Other Types of Pattern

- When you use a pattern that's string, it's automatically wrapped into a call to `regex()`:
- `ignore_case = TRUE` allows characters to match either their uppercase or lowercase forms. This always uses the current locale:

```
bananas <- c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
```

banana

Banana

BANANA

```
str_view(bananas, regex("banana", ignore_case = TRUE))
```

banana

Banana

BANANA

Other Types of Pattern

- `multiline = TRUE` allows ^ and $ to match the start and end of each line rather than the start and end of the complete string.

```
x <- "Line 1\nLine 2\nLine 3"
str_extract_all(x, "^Line")[[1]]
```

```
## [1] "Line"
```

```
str_extract_all(x, regex("^Line", multiline = TRUE))[[1]]
```

```
## [1] "Line" "Line" "Line"
```

Other Types of Pattern

- `comments = TRUE` allows you to use comments and white space to make complex regular expressions more understandable. Spaces are ignored, as is everything after #. To match a literal space, you'll need to excape it : //

```
phone <- regex("
               \\(?     # optional opening parens
               (\\d{3}) # area code
               [)- ]    # optional closing parens, dash, or space
               (\\d{3}) # another three numbers
               [ -]     # optional space or dash
               (\\d{3}) # three more numbers
               ", comments = TRUE)
str_match('123-456-7890', phone)
```

```
##      [,1]          [,2]  [,3]  [,4] 
## [1,] "123-456-789" "123" "456" "789"
```

stringi

- **stringr** is built on top of the stringi package. **stringr** is useful when you're learning because it exposes a minimal set of functions, which have been carefully picked to handle the most common string manipulation functions.
- **stringi**, on the other hand, is designed to be comprehensive. It contains almost every function you might ever need: stringi has 234 functions to stringr's 42. The main difference is the prefix: str_ versus stri_.