Stat 260, Lecture 8, Working with Strings

Brad McNeney

Load packages and datasets

```
library(tidyverse)
library(stringr)
```

Reading

- Strings with `stringr`: Chapter 11 of printed text, Chapter 14 of online text.
  - The text emphasizes regular expressions more than we will in class.
- Working with strings (stringr) cheatsheet at https://github.com/rstudio/cheatsheets/raw/master/strings.pdf]

Working with . . .

- Fixed, or literal strings, like `fish`:
  - count the number of characters in a string
  - detect (yes/no) or find (starting position) substrings
  - extract and substitute substrings
  - split and combine strings
- String patterns, like `f[aeiou]sh` (more on patterns, or regular expressions in a minute):
  - detect, find, extract and substitute
- Use tools from the `stringr` package

The ‘stringr package

- Character string manipulation in base R has evolved over time as a bit of a patch-work of tools.
  - The names and functionality of these tools has been taken from string manipulation tools in Unix and scripting languages like Perl.
  - Steep learning curve for many users.
- The `stringr` package aims for a cleaner interface for tasks that relate to detecting, extracting, replacing and splitting on substrings.

Counting the number of characters

```
mystrings <- c("one fish", "two fish", "red fish", "blue fish")
str_length(mystrings)
```

```
## [1] 8 8 8 9
```

Combining Strings with str_c()

```
str_c(mystrings[1],mystrings[2])
```

```
## [1] "one fishtwo fish"
```

```
str_c(mystrings[1],mystrings[2],sep=", ")
```

```
## [1] "one fish, two fish"
```

```
str_c(mystrings[1],NA,sep=", ")
```

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

```
str_c(mystrings[1],str_replace_na(NA), sep=", ")
```

```
## [1] "one fish, NA"
```

```
str_c(mystrings,collapse=", ")
```

```
## [1] "one fish, two fish, red fish, blue fish"
```

Subsetting Strings with `str_sub()`

- Specify start and stop.
- If stop greater than number of characters, stop at the end of the string.
- If `start` greater than number of characters, return `""`

```
str_sub(mystrings,1,3)
```

```
## [1] "one" "two" "red" "blu"
```

```
str_sub(mystrings,-4,-1) # negative means back from end
```

```
## [1] "fish" "fish" "fish" "fish"
```

```
str_sub(mystrings,1,10000)
```

```
## [1] "one fish"  "two fish"  "red fish"  "blue fish"
```

```
str_sub(mystrings,9,10000)
```

```
## [1] ""  ""  ""  "h"
```

Exercise

- For `demog` as defined in the following code chunk,
  1. extract the substring that represents the gender and age category (`u` stands for unknown) from each of the three components,
  2. extract the last four characters of each of the three components,
  3. Combine the three components into one string, separated by a plus-sign.

```
demog <- c("new_sp_f014",
           "new_sp_m1524",
           "new_sp_mu")
```

Fixed Strings vs Regular Expressions

- Fixed strings are interpreted literally, while regular expressions are a language for specifying patterns.
  - For example, "fish" is fixed and matches only "fish", while "f[aeiou]sh" matches to "fash", "fesh", ..., "fush".
- Functions from `stringr` that detect/find/extract/substitute strings can do so with ether fixed strings or regular expressions.
- We will illustrate these functions with fixed strings first, then discuss regular expressions.
- The text discusses regular expressions first.

Detecting substrings with `str_detect()`

```
pattern <- "red"
str_detect(mystrings,pattern)
```

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

```
mystrings[str_detect(mystrings,pattern)]
```

```
## [1] "red fish"
```

```
pattern <- "fish"
str_detect(mystrings,pattern)
```

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

- (We will later see that we can specify a more general pattern than a fixed string.)

Finding substring starting position

- `str_locate()` returns the start and stop positions of the *first* occurance of a string.
- `str_locate_all()` returns the start and stop of *all* occurances.

```
Seuss <- str_c(mystrings,collapse=", ")
str_locate(Seuss,pattern)
```

```
##      start end
## [1,]     5   8
```

```
str_locate_all(Seuss,pattern)
```

```
## [[1]]
##      start end
## [1,]     5   8
## [2,]    15  18
## [3,]    25  28
## [4,]    36  39
```

```
#str_locate_all(mystrings,pattern)
```

Replacing (substituting) substrings

- Use `str_replace` and `str_replace_all`.

```
str_replace(Seuss,"fish","bird") # replace first occurance
```

```
## [1] "one bird, two fish, red fish, blue fish"
```

```
str_replace_all(Seuss,"fish","bird") # replace all
```

```
## [1] "one bird, two bird, red bird, blue bird"
```

```
str_replace_all(Seuss,c("one" = "1","two"="2")) # multiple replacements
```

```
## [1] "1 fish, 2 fish, red fish, blue fish"
```

Splitting Strings

- Some characters in strings, such as `.`, have a special meaning (more in a minute). One option is to wrap such patterns in `fixed()` for a fixed string

```
mystrings <- c("20.50", "33.33")
str_split(mystrings,pattern=".")
```

```
## [[1]]
## [1] "" "" "" "" "" ""
##
## [[2]]
## [1] "" "" "" "" "" ""
```

```
str_split(mystrings,pattern=fixed("."))
```

```
## [[1]]
## [1] "20" "50"
##
## [[2]]
## [1] "33" "33"
```

Working with string patterns: regular expressions

- Regular expressions (abbreviated regexps) are recipes used to specify search patterns.
- We use character strings to specify regexps in R.
- Regular expressions is a complex topic. We'll only cover the basics.

A simple pattern with .

- To illustrate pattern matching, use a simple pattern `p.n`, meaning `p` followed by any any character, followed by `n`.

```
pattern <- "p.n"
mystrings <- c("pineapple","apple","pen")
str_detect(mystrings,pattern)
```

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

Matching Special Characters

- Suppose we want to match a pattern involving .
- We need to precede, or "escape" the special by a \.
- Unfortunately, \ is a special for character strings, so we need to escape it too; that is, we need to type the character string "\\." to represent the regexp \.

```
pattern2 <- "3.40"
mystrings2 <- c("33.40","3340")
str_detect(mystrings2,pattern2)
```

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

```
pattern2 <- "3\\.40"
str_detect(mystrings2,pattern2)
```

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

Splitting, Locating and Extracting with Patterns

```
pattern
```

```
## [1] "p.n"
```

```
str_split(mystrings,pattern)
```

```
## [[1]]
## [1] ""       "eapple"
##
## [[2]]
## [1] "apple"
##
## [[3]]
## [1] "" ""
```

```
str_locate(mystrings,pattern)
```

```
##      start end
## [1,]     1   3
## [2,]    NA  NA
## [3,]     1   3
```

```
str_extract(mystrings,pattern)
```

```
## [1] "pin" NA    "pen"
```

```
str_match(mystrings,pattern)
```

```
##      [,1]
## [1,] "pin"
## [2,] NA
## [3,] "pen"
```

Replacing patterns

- `str_replace` and `str_replace_all` accept regular expressions; e.g.,

```
str_replace(mystrings,pattern,"PPAP")
```

```
## [1] "PPAPeapple" "apple"      "PPAP"
```

- The replacement string is literal; e.g.,

```
str_replace(mystrings,pattern,"p.n")
```

```
## [1] "p.neapple" "apple"     "p.n"
```

Exercise

- Replace the decimals with commas in the following strings.

```
exstring <- c("$55.30","$22.43")
```

Adding * and + quantifiers to .

- The combinations `.*` and `.+` match multiple characters.
  - E.G., `f.*n` matches `f` followed by 0 or more characters, followed by `n`.
  - `f.+n` matches `f` followed by **1** or more characters, followed by `n`.

```
mystrings <- c("fun","for fun","fn")
pattern1 <- "f.*n"; pattern2 <- "f.+n"
str_extract(mystrings,pattern1)
```

```
## [1] "fun"     "for fun" "fn"
```

```
str_extract(mystrings,pattern2)
```

```
## [1] "fun"     "for fun" NA
```

"Greedy" matching with *

- The * quantifier matches the longest possible string.

```
mystrings <- c("fun","fun, fun, fun","fn")
pattern1 <- "f.*n"
str_extract(mystrings,pattern1)
```

```
## [1] "fun"             "fun, fun, fun" "fn"
```

Numerical quantifiers

- Use {n} to require exactly n matches, {n,} to require n or more, {,m} at most m, and {n,m} between n and m

```
str_extract(mystrings,"f.{6}n")
```

```
## [1] NA          "fun, fun" NA
```

```
str_extract(mystrings,"f.{1,13}n")
```

```
## [1] "fun"           "fun, fun, fun" NA
```

Anchors

- Regular expressions match any part of a string.
- Use the "anchor" ^ to restrict a match to the start and the anchor $ to restrict a match to the end of a string.
-

```
str_extract(mystrings,"^p")
```

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

```
str_extract(mystrings,"e$")
```

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

Exercise

- Create a regular expression that matches words that are exactly three letters long.

Other characters to match

- We have illustrated matching on the pattern `.`, which is any character.
- Instead we can specify a class of characters to match.

```
pattern4 <- "f[aeiou]*n"
mystrings <- c("fan","fin","fun","fan, fin, fun",
               "friend","faint")
str_extract(mystrings,pattern4)
```

```
## [1] "fan"  "fin"  "fun"  "fan"  NA     "fain"
```

```
str_extract_all(mystrings,pattern4)
```

```
## [[1]]
## [1] "fan"
##
## [[2]]
## [1] "fin"
##
## [[3]]
## [1] "fun"
##
## [[4]]
## [1] "fan" "fin" "fun"
##
## [[5]]
## character(0)
##
## [[6]]
## [1] "fain"
```

Shorthands for Common Character Classes

- `\d` matches any digit (create with `"\\d"`)
- `\s` matches any whitespace (create with `"\\s"`)
- Use a dash to specify a range of characters; e.g.,
  - `[A-Z]` matches capital letters
  - `[a-z]` matches lower-case letters
  - `[1-9]` matches any digit (and so is the same as `\d`)
- Use the caret to negate: `[^abc]` matches anything except a, b or c.

Exercise

- Create a regular expression that matches words that end in `ed` but not `eed`.

Alternatives

- The | in a regular expression is like the logical OR.

```
str_replace_all(Seuss,"red|blue","color")
```

```
## [1] "one fish, two fish, color fish, color fish"
```

```
str_replace_all("Is it grey or gray?","gr(e|a)y","white")
```

```
## [1] "Is it white or white?"
```

Converting Case

- Use `str_to_upper()` to change lower- to upper-case and `str_to_lower()` to change upper- to lower-case.

```
str_to_upper(Seuss)
```

```
## [1] "ONE FISH, TWO FISH, RED FISH, BLUE FISH"
```