Week 17 Cohort 4: R4DS Book Club

Chapter 14: Strings

Collin K. Berke

Twitter: @BerkeCollin

Last updated: 2021-03-25

5-minute ice breaker

- What's your favorite thing about your job/school?

Quick housekeeping/reminders

- Video camera is optional, but encouraged.
- If we need to slow down and discuss, let me know.
    - Most likely someone has the same question.
- Take time to learn the theory.
- Please attempt the chapter exercises.
- Please plan on teaching one of the lessons.

Tonight's discussion

- Chapter 14 - Strings
    - Finish our discussion on using regular expressions.
    - Tools provided by `stringr` package.
    - Other uses for regular expressions.

Quick review

- Let's do a quick quiz

Quick disclaimer

- I am **not** a computer programmer/scientist.

- Our discussion will be about the very basics of using regular expressions (regexps).
  - Learn more by checking out these resources:
    - `vignette("regular-expressions")`
    - *Mastering Regular Expressions Book*
    - regular expressions 101

- The `stringr` package provides functions for *common* string operations
  - I'm going to only overview a few
  - `stringi` package is more comprehensive

Why learn the basics of regular expressions?

- Not all text processing can be handled with a function.
- Some parts of unstructured text data are semi-structured.
    - Functions are available to help tidy this data for analysis.
- Allows you to convert long, monotonous tasks into simple code -- thus, increasing productivity.
- What other benefits can you think of?

String Basics

- These are strings:

```
string1 ← "Hey look, I'm a string!" # Using double quotes
string2 ← 'Hello World!'            # Using single quotes
```

- These are also strings:

```
email ← "example.email@gmail.com"
march_madness ← c("Texas Tech", "Gonzaga", "Georgetown", "Creighton")
```

- Even tweets and emojis are strings:

String Basics - Rules to follow

- Escape characters for literal characters

```
double_quote ← "\"" # or '"'
single_quote ← '\'' # or "'"
```

- Special characters (common)
  - "\n" - newline
  - "\t" - tab
  - "\u00b5" - non-English characters
  - More can be found here `?'"'`
- Multiple strings can be stored in a vector

```
string_vector ← c("string", "in", "a", "vector")
string_vector
```

```
## [1] "string" "in"     "a"      "vector"
```

String Basics - Common operations

- **Counting length**

```
str_length(c("Check", "out this cool string   ", NA, NA_character_))
```

```
## [1]  5 23 NA NA
```

- **Combining**

```
# Notice the recycling happening here
str_c("Check out ", c("Lincoln", "Omaha", "Scotts Bluff"), ", NE")
```

```
## [1] "Check out Lincoln, NE"      "Check out Omaha, NE"
## [3] "Check out Scotts Bluff, NE"
```

```
# Collapse into single string
str_c(c("x", "y", "z"), collapse = ", ")
```

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

String Basics - Common operations

- **Subsetting**

```
# State names
state.name[1:3]
```

```
## [1] "Alabama" "Alaska"  "Arizona"
```

```
# State abbreviations
str_sub(state.name[1:3], 1, 3)
```

```
## [1] "Ala" "Ala" "Ari"
```

```
# Reverse it
str_sub(state.name[1:3], -3, -1)
```

```
## [1] "ama" "ska" "ona"
```

String Basics - Common operations

- **Convert case**

```
# Case to lower
(state_lower ← str_to_lower(state.name[1:3]))
```

```
## [1] "alabama" "alaska"  "arizona"
```

```
# Case to upper
(str_to_upper(state_lower))
```

```
## [1] "ALABAMA" "ALASKA"  "ARIZONA"
```

```
# Case to title
(str_to_title(state_lower))
```

```
## [1] "Alabama" "Alaska"  "Arizona"
```

Using `REGEXPS` - Rules to follow

- Interesting perspective

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. ~ Jaime Zawinski, quoted in book

- Regular expressions are powerful, but use them wisely (example from book)

- In your work, where might you get a false sense of power using regular expressions?
- Break the problem into smaller bits whenever possible
- Utilize the `str_view()` and `str_view_all()` to see the matches
    - Use other tools

Using `REGEXPS` - The basics

- Exact matching

```
month.name %>%
  as_tibble() %>%
  mutate(
    match = str_detect(value, "ber")
  )
```

```
## # A tibble: 12 x 2
##    value     match
##    <chr>     <lgl>
##  1 January   FALSE
##  2 February  FALSE
##  3 March     FALSE
##  4 April     FALSE
##  5 May       FALSE
##  6 June      FALSE
##  7 July      FALSE
##  8 August    FALSE
##  9 September TRUE
## 10 October   TRUE
## 11 November  TRUE
## 12 December  TRUE
```

Using `REGEXPS` - The basics

- Using `.` to match any character

```
# Months that have a u in between characters
month.name %>%
  as_tibble() %>%
  mutate(
    match = str_detect(value, "(A.|.A.)")
  )
```

```
## # A tibble: 12 x 2
##    value     match
##    <chr>     <lgl>
##  1 January   FALSE
##  2 February  FALSE
##  3 March     FALSE
##  4 April     TRUE
##  5 May       FALSE
##  6 June      FALSE
##  7 July      FALSE
##  8 August    TRUE
##  9 September FALSE
## 10 October   FALSE
## 11 November  FALSE
```

Using `REGEXPS` - The basics

- Using anchors `(^, $)`

> If you begin with power `(^)`, you end up with money `($)`.

```
# Months that begin with J, end in y
month.name %>%
      as_tibble() %>%
      mutate(
        match = str_detect(value, "^J[a-z]*y$")
      )
```

```
## # A tibble: 12 x 2
##    value     match
##    <chr>     <lgl>
##  1 January   TRUE
##  2 February  FALSE
##  3 March     FALSE
##  4 April     FALSE
##  5 May       FALSE
##  6 June      FALSE
##  7 July      TRUE
##  8 August    FALSE
##  9 September FALSE
```

Using `REGEXPS` - complex patterns

- Character classes
- 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
  - `[ $.*|()]`: match special characters

```
# Months that begin with J, end in y, where any number of
# lower case letter is present
month.name %>%
      as_tibble() %>%
      mutate(
        match = str_detect(value, "^J[a-z]*y$")
      )
```

Using `REGEXPS` - complex patterns

- Alternatives
  - Use the `(|)`
- Special patterns that match more than one character
  - Pick between on or more alternative patterns

```
# No gravy
str_detect(c("grey", "gray", "gravy"), "gr(e|a)y")
```

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

```
# Gravy, please
str_detect(c("grey", "gray", "gravy"), "gr(e|a|av)y")
```

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

Using `REGEXPS` - complex patterns

- Repetition

- Use special characters with common rules
  - `?`: 0 or 1
  - `+`: 1 or more
  - `*`: 0 or more

- Use notation for precise numbers
  - `{n}` : exactly n
  - `{n, }`: n or more
  - `{,m}` : at most m
  - `{n,m}`: between n and m

Using `REGEXPS` - complex patterns

- Repetition matching using special characters

```
# Months that begin with J, end in y, where any number of
# lower case letter is present
month.name %>%
      as_tibble() %>%
      mutate(
        match = str_detect(value, "^J[a-z]*y$")
      )
```

```
## # A tibble: 12 x 2
##    value     match
##    <chr>     <lgl>
##  1 January   TRUE
##  2 February  FALSE
##  3 March     FALSE
##  4 April     FALSE
##  5 May       FALSE
##  6 June      FALSE
##  7 July      TRUE
##  8 August    FALSE
##  9 September FALSE
## 10 October   FALSE
```

Using `REGEXPS` - complex patterns

- Repetition precise matching

```
x ← "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_extract_all(x, "C{2}")
```

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

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

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

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

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

```
# These matches are greedy: it returns the longest match
# To make it not greedy, use ?
str_extract(x, "C{2,3}?")
```

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

Using `REGEXPS` - complex patterns

- Grouping and backreferences

```
str_extract(fruit, "(..)\\1") %>%
  as_tibble() %>%
  drop_na()
```

```
## # A tibble: 6 x 1
##   value
##   <chr>
## 1 anan
## 2 coco
## 3 cucu
## 4 juju
## 5 papa
## 6 alal
```

Using `REGEXPS` - complex patterns

- Grouping and backreferences, another example

```
starwars$name %>%
  as_tibble() %>%
  filter(str_detect(value, "( ... \\s)\\1"))
```

```
## # A tibble: 1 x 1
##   value
##   <chr>
## 1 Jar Jar Binks
```

🔨 The tools for string operations 🔨

- Common operations `stringr` has a function.
  - Determine which strings match a pattern
  - Find the positions of matches
  - Extract the content of matches
  - Replace matches with new values
  - Split a string based on a match

The tools for string operations - a diagram and cheatsheet

- Check out the stringr cheatsheet
- Check out the examples I sent

Other uses for `REGEXPS`

- `apropos()` to find objects in your global environment.

```
apropos("replace")
```

```
## [1] "%+replace%"       "replace"          "replace_na"       "setRepl
## [5] "str_replace"      "str_replace_all"  "str_replace_na"   "theme_r
```

- `dir()` to find files based on a pattern

```
dir(path = "data/", pattern = "\\.csv$")
```

```
## [1] "20210317151523_mtcars.csv" "20210317151543_mtcars.csv"
## [3] "20210317151546_mtcars.csv" "20210317151549_mtcars.csv"
## [5] "20210317151550_mtcars.csv"
```

What if `stringr` doesn't have what I need?

- `stringr` is built on the `stringi` package
  - Check out the stringr cheatsheet
- Some interesting functions from scanning the `stringi` package
  - `stri_enc_detect()` - detects character set and language
  - `stri_join_list()` - combine strings in a list
  - `stri_reverse()` - reverse the order of the strings
  - `stri_stats()` - general stats for a character vector
  - ... and many more

Questions/comments