Chapter 14 & 15: Exercises and Executables

Github screenshot at the end of this document

14.2.4 Exercises

1. Create strings that contain the following values:
    1. He said "That's amazing!"
- string1 <- "He said \"That's amazing!\""
    2. \a\b\c\d
- string2 <- "\\a\\b\\c\\d"
    3. \\\\\\
- string3 <- "\\\\\\"

2. Create the string in your R session and print it. What happens to the special “\u00a0”? How does str_view() display it? Can you do a little googling to figure out what this special character is?
- x <- "This\u00a0is\u00a0tricky"
- print(x)
- The special \u00a0 is interpreted as a non-breaking space, which looks like a regular space when printed but is treated differently and highlighted separately by functions like str_view(). str_view() displays \u00a0 by highlighting it as a distinct character, showing that it is not a regular space but a special non-breaking space. For example, names like **"Dr. Smith"** — a non-breaking space between "Dr." and "Smith" makes sure the title and name stay together on the same line, instead of breaking it into two different lines.

14.3.4 Exercises

1. Compare and contrast the results of paste0() with str_c() for the following inputs:
- str_c("hi ", NA)

- str_c(letters[1:2], letters[1:3])
- When using str_c("hi ", NA), the function returns NA because str_c() treats any NA input as missing and does not concatenate it. In contrast, paste0("hi ", NA) returns "hi NA" because paste0() converts NA into the string "NA" and includes it in the result.
- For the input str_c(letters[1:2], letters[1:3]), both str_c() and paste0() recycle the shorter vector to match the length of the longer one, resulting in the same output: "aa", "bb", and "ac". So while both functions behave similarly when handling vector recycling, they differ in how they handle NA values.

2. What's the difference between paste() and paste0()? How can you recreate the equivalent of paste() with str_c()?

- paste() adds a space (" ") between strings by default.
- paste0() adds no space — it sticks strings together directly.
- To recreate the equivalent of paste() using str_c(), you add the argument sep = " " because paste() joins words with a space by default, while str_c() does not. By specifying sep = " ", str_c() will behave the same way as paste() and insert a space between the strings.

3. Convert the following expressions from str_c() to str_glue() or vice versa:
    1. str_c("The price of ", food, " is ", price)

- str_glue("The price of {food} is {price}")

    2. str_glue("I'm {age} years old and live in {country}")

- str_c("I'm ", age, " years old and live in ", country)

    3. str_c("\\section{", title, "}")

- str_glue("\\section{{{title}}}")

14.5.3 Exercises

1. When computing the distribution of the length of babynames, why did we use wt = n?

- In the babynames dataset:
- Each row represents a **name** for a **specific year and gender**.
- The column n tells us **how many babies** got that name that year.

If we don't use wt = n, we would treat each **name entry equally**, no matter if only babies or 5,000 babies had that name.

2. Use str_length() and str_sub() to extract the middle letter from each baby name. What will you do if the string has an even number of characters?

- Use str_length() to find how many letters are in each baby name.
- Use str_sub() to extract the middle letter from the name.
- If the name has an odd number of letters:
  - There's one clear middle letter → use (length + 1) / 2.
- If the name has an even number of letters:
  - There are two middle letters → choose the first one → use length / 2.

3. Are there any major trends in the length of babynames over time? What about the popularity of first and last letters?

- Over time, baby names have generally become shorter. In the late 1800s and early 1900s, longer names were more common and widely used. However, in recent decades, there's been a noticeable shift toward shorter names, with simple and concise names like Mia, Leo, and Ava growing in popularity. This trend reflects changing naming preferences across generations.

Trends in first and last letters of names:

First letters:

- Some starting letters like J (e.g., James, John, Jennifer) have remained popular across many decades.
- Letters like A, E, and L have become more common in modern names (e.g., Ava, Emma, Liam).

Last letters:

- For girls, names ending in “a” (like Emma, Olivia, Sophia) have become especially popular in recent years.
- For boys, names ending in “n” (like Mason, Aiden, Logan) have seen a big rise.
- Names ending in “e” or “y” were more common in earlier decades.

15.3.5 Exercises

1. What baby name has the most vowels? What name has the highest proportion of vowels? (Hint: what is the denominator?)

❖ Baby name with the most vowels:

```
babynames %>%
  distinct(name) %>%  # only need unique names
  mutate(
    vowel_count = str_count(name, "[aeiouAEIOU]"),
    name_length = str_length(name),
    vowel_prop = vowel_count / name_length
  ) %>%
  arrange(desc(vowel_count)) %>%
  slice(1)
```

❖ Highest proportion of vowels:

```
babynames %>%
  distinct(name) %>%
  mutate(
    vowel_count = str_count(name, "[aeiouAEIOU]"),
    name_length = str_length(name),
    vowel_prop = vowel_count / name_length
```

) %>%

arrange(desc(vowel_prop)) %>%

slice(1)

2. Replace all forward slashes in "a/b/c/d/e" with backslashes. What happens if you attempt to undo the transformation by replacing all backslashes with forward slashes? (We'll discuss the problem very soon.)
- To replace all forward slashes (/) in the string "a/b/c/d/e" with backslashes (\), you need to use double backslashes in the replacement string ("\\\\") because backslashes are special escape characters in R.
- If you then try to undo this transformation by replacing the backslashes with forward slashes, it works correctly only if you escape the backslash properly in the pattern. The main issue is that backslashes are used for escaping in both R strings and regular expressions, so you need extra backslashes to represent just one. If you're not careful with escaping, R might misinterpret the pattern, and the replacement won't work as expected.

3. Implement a simple version of str_to_lower() using str_replace_all().
- simple_to_lower <- function(text) {
-   uppercase <- LETTERS
-   lowercase <- letters
-   str_replace_all(text, setNames(lowercase, uppercase))
- }

4. Create a regular expression that will match telephone numbers as commonly written in your country.
- phone_pattern <- "^(\\(?\\d{3}\\)?[-]?)?\\d{3}[-]?\\d{4}$"
- str_detect(c("123-456-7890", "(123) 456-7890", "1234567890", "12345"), phone_pattern)

15.4.7 Exercises

1. How would you match the literal string "'\? How about "$^$"?

- pattern <- "\\$\\^\\$"

2. Explain why each of these patterns don’t match a \: "\", "\\", "\\\".

Pattern: "\\"

- In R, \\ becomes one backslash (\).
- Regex sees just one backslash — which starts an escape sequence but doesn’t complete it.

Pattern: "\\\\"

- In R, \\\\ becomes two backslashes → regex sees one backslash.
- This is the correct way to match one literal backslash.

Pattern: "\\\\\""

- This is five backslashes and a quote.
- R gets confused:
    - \\ → \
    - \\ → \
    - \" → ends the string (quote)

3. Given the corpus of common words in stringr::words, create regular expressions that find all words that:
    1. Start with “y”.
    2. Don’t start with “y”.
    3. End with “x”.
    4. Are exactly three letters long. (Don’t cheat by using str_length()!)
    5. Have seven letters or more.
    6. Contain a vowel-consonant pair.
    7. Contain at least two vowel-consonant pairs in a row.
    8. Only consist of repeated vowel-consonant pairs.

- # 1. Start with "y"
- str_subset(words, "^y")
- # 2. Don't start with "y"
- str_subset(words, "^[^y]")

- # 3. End with "x"
- str_subset(words, "x$")
- # 4. Exactly three letters long
- str_subset(words, "^...$")
- # 5. Seven letters or more
- str_subset(words, "^.{7,}$")
- # 6. Contain a vowel-consonant pair
- str_subset(words, "[aeiou][^aeiou]")
- # 7. Contain at least two vowel-consonant pairs in a row
- str_subset(words, "([aeiou][^aeiou]){2,}")
- # 8. Only consist of repeated vowel-consonant pairs
- str_subset(words, "^([aeiou][^aeiou])+$")

4. Create 11 regular expressions that match the British or American spellings for each of the following words: airplane/aeroplane, aluminum/aluminium, analog/analogue, ass/arse, center/centre, defense/defence, donut/doughnut, gray/grey, modeling/modelling, skeptic/sceptic, summarize/summarise. Try and make the shortest possible regex!

\# 1. airplane / aeroplane

"a[ei]roplane"

\# 2. aluminum / aluminium

"alumin(i)?um"

\# 3. analog / analogue

"analogu?e"

\# 4. ass / arse

"a(rs)?e"

5. center / centre

"cent(re|er)"

6. defense / defence

"defen[sc]e"

7. donut / doughnut

"dough?nuts?"

8. gray / grey

"gr[ae]y"

9. modeling / modelling

"modell?ing"

10. skeptic / sceptic

"sc?eptic"

11. summarize / summarise

"summariz?e"

5. Switch the first and last letters in words. Which of those strings are still words?

- # Swap first and last letters
- swapped_words <- str_replace(words,
-                                  "^(.)(.*)(.)$",
-                                  "\\3\\2\\1")
-
- # Find which swapped words are still valid words

- valid_swaps <- swapped_words[swapped_words %in% words]

6. Describe in words what these regular expressions match: (read carefully to see if each entry is a regular expression or a string that defines a regular expression.)

1. ^.*$ - Matches any line of text, including an empty one.
   - `^` = start of line
   - `.*` = any number of any characters
   - `$` = end of line
2. "\\{.+\\}" - Matches a string that starts with `{`, ends with `}`, and has at least one character in between.
   - `\\{` = literal `{`
   - `.+` = one or more of any character
   - `\\}` = literal `}`
3. \d{4}-\d{2}-\d{2} - Matches a date in the format YYYY-MM-DD.
   - `\d{4}` = four digits
   - `-` = hyphen
   - `\d{2}` = two digits (for month and day)
4. "\\\\{4}" - Matches exactly four backslashes (`\\\\`).
   - In R, `\\\\{4}` becomes `\\{4}` in regex → match 4 literal `\`
5. \..\..\.. - Matches three periods separated by any character.
   - `\.` = a literal dot
   - `.` = any character

   So this matches things like: `.a.b.c`, `.1.2.3`, etc.
6. (.)\1\1 - Matches three repeated copies of the same character in a row.
   - `(.)` = captures any character
   - `\1\1` = that character repeated two more times
7. "(..)\\1" - Matches a quoted string where two characters repeat right after each other.
   - `(..)` = captures two characters
   - `\\1` = those same two characters again

7. Solve the beginner regex crosswords at https://regexcrossword.com/challenges/beginner.

- Completed

15.6.4 Exercises

1. For each of the following challenges, try solving it by using both a single regular expression, and a combination of multiple str_detect() calls.
    1. Find all words that start or end with x.
    2. Find all words that start with a vowel and end with a consonant.
    3. Are there any words that contain at least one of each different vowel?

```
str_view(words, "^x|x$")
words[str_detect(words, "^x") | str_detect(words, "x$")]

str_view(words, "^[aeiou].*[^aeiou]$")
words[str_detect(words, "^[aeiou]") & str_detect(words, "[^aeiou]$")]

words[
 str_detect(words, "a") &
 str_detect(words, "e") &
 str_detect(words, "i") &
 str_detect(words, "o") &
 str_detect(words, "u")
]
```

2. Construct patterns to find evidence for and against the rule "i before e except after c"?

```
str_view(words, "cie")  # Breaks rule
str_view(words, "cei")  # Follows rule
```

3. colors() contains a number of modifiers like "lightgray" and "darkblue". How could you automatically identify these modifiers? (Think about how you might detect and then remove the colors that are modified).

```
modifiers <- str_subset(colors(), "^(light|dark|medium)")
str_remove(modifiers, "^(light|dark|medium)")
```

4. Create a regular expression that finds any base R dataset. You can get a list of these datasets via a special use of the data() function: data(package = "datasets")$results[, "Item"]. Note that a number of old datasets are individual vectors; these contain the name of the grouping "data frame" in parentheses, so you'll need to strip those off.

```
ds_names <- data(package = "datasets")$results[, "Item"]
cleaned <- str_replace(ds_names, "\\s*\\(.*\\)", "")
str_subset(cleaned, "^[a-zA-Z]+$")
```

Executables

```
# Chapter 14 and 15................

#Chapter 14  Strings..............

#14.1.1 Prerequisites..............

library(tidyverse)
library(babynames)

#14.2 Creating a string.............

string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside a string, I use single quotes'

#14.2.1 Escapes.........

double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"

backslash <- "\\"

x <- c(single_quote, double_quote, backslash)
x


str_view(x)


#14.2.2 Raw strings................

tricky <- "double_quote <- \"\\\"\" # or '\"'
single_quote <- '\\'' # or \"'\""
str_view(tricky)

tricky <- r"(double_quote <- "\"" # or '"'
```

```
single_quote <- '\'' # or "'")"
str_view(tricky)

#To eliminate the escaping, you can instead use a raw string

tricky <- r"(double_quote <- "\"" # or '"'

single_quote <- '\'' # or "'")"

#r"(  your string goes here )"
#r"[ your string if it includes ()]"
#r"{ your string if it includes []}"


#14.2.3 Other special characters........

x <- c("one\ntwo", "one\ttwo", "\u00b5", "\U0001f604")
x

#14.3 Creating many strings from data........

#14.3.1 str_c()............

str_c("x", "y")

str_c("x", "y", "z")

str_c("Hello ", c("John", "Susan"))

df <- tibble(name = c("Flora", "David", "Terra", NA))
df |> mutate(greeting = str_c("Hi ", name, "!"))

df |>
  mutate(
    greeting1 = str_c("Hi ", coalesce(name, "you"), "!"),
    greeting2 = coalesce(str_c("Hi ", name, "!"), "Hi!")
  )

#14.3.2 str_glue()..........
```

```
df |> mutate(greeting = str_glue("Hi {name}!"))

df |> mutate(greeting = str_glue("{{Hi {name}!}}"))

#14.3.3 str_flatten()..........

str_flatten(c("x", "y", "z"))

str_flatten(c("x", "y", "z"), ", ")

str_flatten(c("x", "y", "z"), ", ", last = ", and ")

df <- tribble(
  ~ name, ~ fruit,
  "Carmen", "banana",
  "Carmen", "apple",
  "Marvin", "nectarine",
  "Terence", "cantaloupe",
  "Terence", "papaya",
  "Terence", "mandarin"
)
df |>
  group_by(name) |>
  summarize(fruits = str_flatten(fruit, ", "))

#14.4 Extracting data from strings...........

#14.4.1 Separating into rows.........

df1 <- tibble(x = c("a,b,c", "d,e", "f"))
df1 |>
  separate_longer_delim(x, delim = ",")

df2 <- tibble(x = c("1211", "131", "21"))
df2 |>
  separate_longer_position(x, width = 1)


#14.4.2 Separating into columns.........
```

```
df3 <- tibble(x = c("a10.1.2022", "b10.2.2011", "e15.1.2015"))
df3 |>
  separate_wider_delim(
    x,
    delim = ".",
    names = c("code", "edition", "year")
  )

df3 |>
  separate_wider_delim(
    x,
    delim = ".",
    names = c("code", NA, "year")
  )

df4 <- tibble(x = c("202215TX", "202122LA", "202325CA"))
df4 |>
  separate_wider_position(
    x,
    widths = c(year = 4, age = 2, state = 2)
  )

#14.4.3 Diagnosing widening problems...........

df <- tibble(x = c("1-1-1", "1-1-2", "1-3", "1-3-2", "1"))

df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z")
  )

debug <- df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_few = "debug"
  )
```

```
debug

debug |> filter(!x_ok)

df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_few = "align_start"
  )

df <- tibble(x = c("1-1-1", "1-1-2", "1-3-5-6", "1-3-2", "1-3-5-7-9"))

df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z")
  )

debug <- df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_many = "debug"
  )

debug |> filter(!x_ok)

df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_many = "drop"
  )
```

```
df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_many = "merge"
  )

#14.5 Letters........

#14.5.1 Length.............

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

babynames |>
  count(length = str_length(name), wt = n)

babynames |>
  filter(str_length(name) == 15) |>
  count(name, wt = n, sort = TRUE)

#14.5.2 Subsetting........

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

str_sub(x, -3, -1)

str_sub("a", 1, 5)

babynames |>
  mutate(
    first = str_sub(name, 1, 1),
    last = str_sub(name, -1, -1)
  )

#14.6 Non-English text...........

#14.6.1 Encoding..............
```

```r
charToRaw("Hadley")

x1 <- "text\nEl Ni\xf1o was particularly bad this year"
read_csv(x1)$text

x2 <- "text\n\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
read_csv(x2)$text

read_csv(x1, locale = locale(encoding = "Latin1"))$text

read_csv(x2, locale = locale(encoding = "Shift-JIS"))$text


#14.6.2 Letter variations.........


u <- c("\u00fc", "u\u0308")
str_view(u)

str_length(u)

str_sub(u, 1, 1)

u[[1]] == u[[2]]

str_equal(u[[1]], u[[2]])


#14.6.3 Locale-dependent functions.............

str_to_upper(c("i", "ı"))

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

str_sort(c("a", "c", "ch", "h", "z"))

str_sort(c("a", "c", "ch", "h", "z"), locale = "cs")


# Chapter 15  Regular expressions...............
```

```r
#15.1.1 Prerequisites

library(tidyverse)
library(babynames)

#15.2 Pattern basics...........

str_view(fruit, "berry")

str_view(c("a", "ab", "ae", "bd", "ea", "eab"), "a.")

str_view(fruit, "a...e")

str_view(c("a", "ab", "abb"), "ab?")

str_view(c("a", "ab", "abb"), "ab+")

str_view(c("a", "ab", "abb"), "ab*")

str_view(words, "[aeiou]x[aeiou]")

str_view(words, "[^aeiou]y[^aeiou]")

str_view(fruit, "apple|melon|nut")

str_view(fruit, "aa|ee|ii|oo|uu")


# 15.3 Key functions................


#15.3.1 Detect matches........

str_detect(c("a", "b", "c"), "[aeiou]")

babynames |>
  filter(str_detect(name, "x")) |>
  count(name, wt = n, sort = TRUE)
```

```
babynames |>
  group_by(year) |>
  summarize(prop_x = mean(str_detect(name, "x"))) |>
  ggplot(aes(x = year, y = prop_x)) +
  geom_line()

# 15.3.2 Count matches..........

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

str_count("abababa", "aba")

str_view("abababa", "aba")

babynames |>
  count(name) |>
  mutate(
    vowels = str_count(name, "[aeiou]"),
    consonants = str_count(name, "[^aeiou]")
  )

babynames |>
  count(name) |>
  mutate(
    name = str_to_lower(name),
    vowels = str_count(name, "[aeiou]"),
    consonants = str_count(name, "[^aeiou]")
  )

# 15.3.3 Replace values..................

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

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


# 15.3.4 Extract variables...........
```

```
df <- tribble(
  ~str,
  "<Sheryl>-F_34",
  "<Kisha>-F_45",
  "<Brandon>-N_33",
  "<Sharon>-F_38",
  "<Penny>-F_58",
  "<Justin>-M_41",
  "<Patricia>-F_84",
)

df |>
  separate_wider_regex(
    str,
    patterns = c(
      "<",
      name = "[A-Za-z]+",
      ">-",
      gender = ".",
      "_",
      age = "[0-9]+"
    )
  )

# 15.4 Pattern details.............

# 15.4.1 Escaping...........

# To create the regular expression \., we need to use \\.
dot <- "\\."

# But the expression itself only contains one \
str_view(dot)
#> [1] │ \.

# And this tells R to look for an explicit .
str_view(c("abc", "a.c", "bef"), "a\\.c")
#> [2] │ <a.c>
```

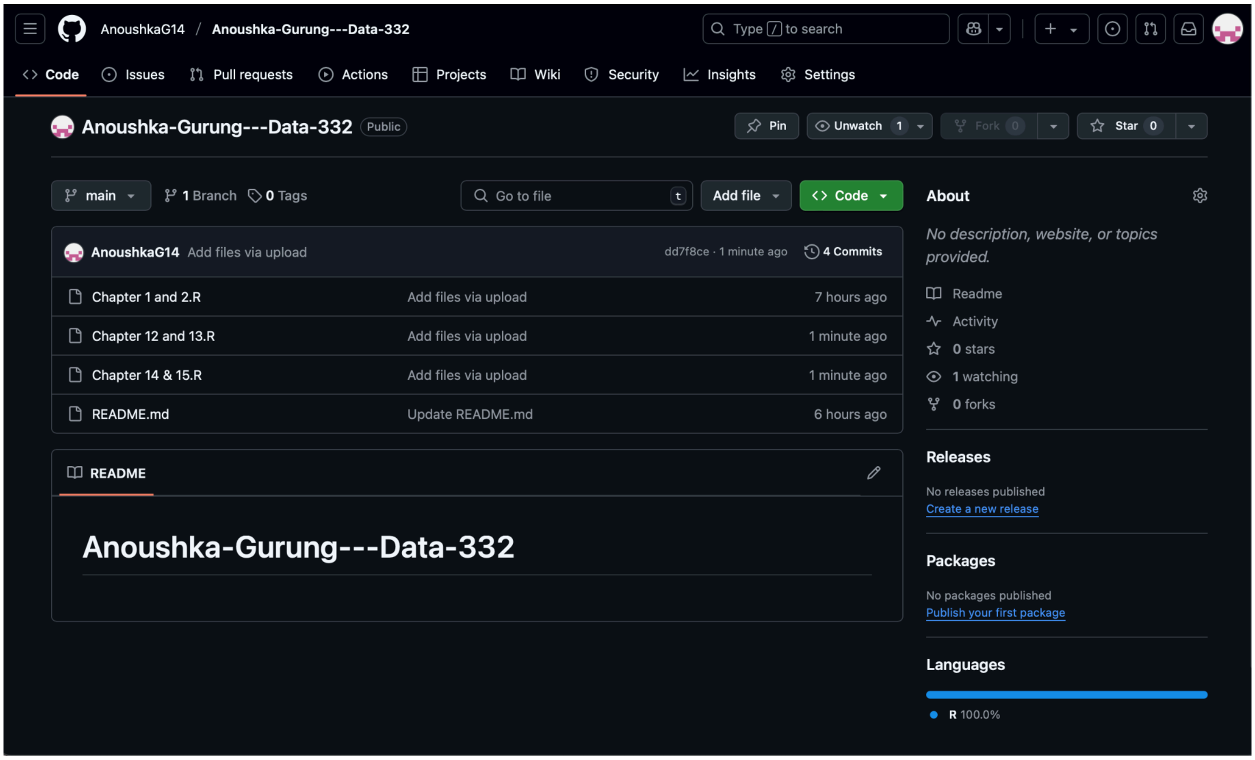
```
x <- "a\\b"
str_view(x)
#> [1] │ a\b
str_view(x, "\\\\")
#> [1] │ a<\>b

str_view(x, r"{\\}")

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

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

# 15.4.2 Anchors..................

str_view(fruit, "^a")

str_view(fruit, "a$")

str_view(fruit, "apple")

str_view(fruit, "^apple$")

x <- c("summary(x)", "summarize(df)", "rowsum(x)", "sum(x)")
str_view(x, "sum")

str_view(x, "\\bsum\\b")

str_view("abc", c("$", "^", "\\b"))

str_replace_all("abc", c("$", "^", "\\b"), "--")


# 15.4.3 Character classes...............

x <- "abcd ABCD 12345 -!@#%."
str_view(x, "[abc]+")

str_view(x, "[a-z]+")

str_view(x, "[^a-z0-9]+")
```

```
str_view("a-b-c", "[a-c]")

str_view("a-b-c", "[a\\-c]")

x <- "abcd ABCD 12345 -!@#%."
str_view(x, "\\d+")

str_view(x, "\\D+")

str_view(x, "\\s+")

str_view(x, "\\S+")

str_view(x, "\\w+")

str_view(x, "\\W+")


# 15.4.6 Grouping and capturing............

str_view(fruit, "(..)\\1")

str_view(words, "^(..).*\\1$")

sentences |>
  str_replace("(\\w+) (\\w+) (\\w+)", "\\1 \\3 \\2") |>
  str_view()

sentences |>
  str_match("the (\\w+) (\\w+)") |>
  head()

sentences |>
  str_match("the (\\w+) (\\w+)") |>
  as_tibble(.name_repair = "minimal") |>
  set_names("match", "word1", "word2")

x <- c("a gray cat", "a grey dog")
str_match(x, "gr(e|a)y")
```

```
str_match(x, "gr(?:e|a)y")


# 15.5 Pattern control...................

# 15.5.1 Regex flags................

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

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

x <- "Line 1\nLine 2\nLine 3"
str_view(x, ".Line")
str_view(x, regex(".Line", dotall = TRUE))

x <- "Line 1\nLine 2\nLine 3"
str_view(x, "^Line")

str_view(x, regex("^Line", multiline = TRUE))

phone <- regex(
  r"(
    \(?     # optional opening parens
    (\d{3}) # area code
    [)\-]?  # optional closing parens or dash
    \ ?     # optional space
    (\d{3}) # another three numbers
    [\ -]?  # optional space or dash
    (\d{4}) # four more numbers
  )",
  comments = TRUE
)

str_extract(c("514-791-8141", "(123) 456 7890", "123456"), phone)


# 15.5.2 Fixed matches...................
```

```r
str_view(c("", "a", "."), fixed("."))

str_view("x X", "X")

str_view("x X", fixed("X", ignore_case = TRUE))

str_view("i İ ı I", fixed("İ", ignore_case = TRUE))

str_view("i İ ı I", coll("İ", ignore_case = TRUE, locale = "tr"))


#15.6 Practice..................

str_view(sentences, "^The")

str_view(sentences, "^The\\b")

str_view(sentences, "^She|He|It|They\\b")

str_view(sentences, "^(She|He|It|They)\\b")

pos <- c("He is a boy", "She had a good time")
neg <- c("Shells come from the sea", "Hadley said 'It's a great day'")

pattern <- "^(She|He|It|They)\\b"
str_detect(pos, pattern)

str_detect(neg, pattern)

# 15.6.2 Boolean operations.............

str_view(words, "^[^aeiou]+$")

str_view(words[!str_detect(words, "[aeiou]")])

str_view(words, "a.*b|b.*a")

words[str_detect(words, "a") & str_detect(words, "b")]

words[str_detect(words, "a.*e.*i.*o.*u")]
```

```
# ...
words[str_detect(words, "u.*o.*i.*e.*a")]

words[
  str_detect(words, "a") &
    str_detect(words, "e") &
    str_detect(words, "i") &
    str_detect(words, "o") &
    str_detect(words, "u")
]

# 15.6.3 Creating a pattern with code..........

str_view(sentences, "\\b(red|green|blue)\\b")

rgb <- c("red", "green", "blue")

str_c("\\b(", str_flatten(rgb, "|"), ")\\b")

str_view(colors())

cols <- colors()
cols <- cols[!str_detect(cols, "\\d")]
str_view(cols)

pattern <- str_c("\\b(", str_flatten(cols, "|"), ")\\b")
str_view(sentences, pattern)

# 15.7 Regular expressions in other places...........

# 15.7.2 Base R............

apropos("replace")

head(list.files(pattern = "\\.Rmd$"))
```


AnoushkaG14 / Anoushka-Gurung---Data-332
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Anoushka-Gurung---Data-332
Public
Pin
Unwatch 1
Fork 0
Star 0
main
1 Branch
0 Tags
Go to file
t
Add file
Code
AnoushkaG14 Add files via upload
dd7f8ce · 1 minute ago
4 Commits
Chapter 1 and 2.R
Add files via upload
7 hours ago
Chapter 12 and 13.R
Add files via upload
1 minute ago
Chapter 14 & 15.R
Add files via upload
1 minute ago
README.md
Update README.md
6 hours ago
README
Anoushka-Gurung---Data-332
About
No description, website, or topics provided.
Readme
Activity
0 stars
1 watching
0 forks
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Languages
R 100.0%