Strings and Regular Expressions

```
library(tidyverse)
```

Learning Objectives

- Manipulating strings with the stringr package.
- Regular expressions
- Chapter 14 of RDS.
- Work with Strings Cheatsheet.
- Stringr Overview.

Strings

- In R, strings (also called "characters") are created and displayed within quotes:

```
x <- "I am a string!"
x
```

```
[1] "I am a string!"
```

- Anything within quotes is a string, even numbers!

```
y <- "3"
class(y)
```

```
[1] "character"
```

- You can have a vector of strings.

```
x <- c("I", "am", "a", "string", "vector")
x[2:3]
```

```
[1] "am" "a"
```

- The backslash `"\"` means that what is after the backslash is special in some way. For example, if you want to put a quotation mark in a string, you can "escape" the quotation mark with a backslash.

```
x <- "As Tolkein said, \"Not all those who wonder are lost\""
writeLines(x)
```

```
As Tolkein said, "Not all those who wonder are lost"
```

- Above, `writeLines()` will print out the string itself. `print()` will print out the printed representation of the string (with backslashes and all).

```
print(x)
```

```
[1] "As Tolkein said, \"Not all those who wonder are lost\""
```

- `"\n"` represents a new line.

```
x <- "Not all those\nwho wonder are lost."
writeLines(x)
```

```
Not all those
who wonder are lost.
```

- `"\t"` represents a tab.

```
x <- "Not all those\twho wonder are lost."
writeLines(x)
```

```
Not all those   who wonder are lost.
```

- You can add any Unicode character with a `\u` followed by the hexadecimal unicode representation of that character.

```
mu <- "\u00b5"
writeLines(mu)
```

```
µ
```

stringr Intro

- The stringr package contains a lot of convenience functions for manipulating strings (and they are a lot more user friendly than base R's string manipulation functions like `grep()` and `gsub()`).
- stingr is part of the tidyverse so you do not have to load it separately.

```
library(tidyverse)
```

- All of stringr's functions begin with "`str_`", so you can press tab after typing "`str_`" and a list of possible string manipulation functions will pop up (in RStudio).

Length

- To get the number of characters in a string, use `str_length()`.

```
str_length("Upon the hearth the fire is red,")
```

```
[1] 32
```

Detect Matches

- `str_detect(string, pattern)`: Returns a logical value or vector: TRUE for a match to the pattern and FALSE if no match is found.
- The negate = argument reverses the result.
- Useful for creating new variables or filtering rows.

```
data("presedential") ## 12 former US presidents
```

```
Warning in data("presedential"): data set 'presedential' not found
```

```r
presidential  |>
  mutate(has_a_e = str_detect(name, "e|a"))
```

```
# A tibble: 12 x 5
   name       start      end        party      has_a_e
   <chr>      <date>     <date>     <chr>      <lgl>
 1 Eisenhower 1953-01-20 1961-01-20 Republican TRUE
 2 Kennedy    1961-01-20 1963-11-22 Democratic TRUE
 3 Johnson    1963-11-22 1969-01-20 Democratic FALSE
 4 Nixon      1969-01-20 1974-08-09 Republican FALSE
 5 Ford       1974-08-09 1977-01-20 Republican FALSE
 6 Carter     1977-01-20 1981-01-20 Democratic TRUE
 7 Reagan     1981-01-20 1989-01-20 Republican TRUE
 8 Bush       1989-01-20 1993-01-20 Republican FALSE
 9 Clinton    1993-01-20 2001-01-20 Democratic FALSE
10 Bush       2001-01-20 2009-01-20 Republican FALSE
11 Obama      2009-01-20 2017-01-20 Democratic TRUE
12 Trump      2017-01-20 2021-01-20 Republican FALSE
```

```r
presidential  |>
  filter(str_detect(name, "e|a"))
```

```
# A tibble: 5 x 4
  name       start      end        party
  <chr>      <date>     <date>     <chr>
1 Eisenhower 1953-01-20 1961-01-20 Republican
2 Kennedy    1961-01-20 1963-11-22 Democratic
3 Carter     1977-01-20 1981-01-20 Democratic
4 Reagan     1981-01-20 1989-01-20 Republican
5 Obama      2009-01-20 2017-01-20 Democratic
```

```r
presidential  |>
  filter(str_detect(name, "e|a", negate = TRUE))
```

```
# A tibble: 7 x 4
  name    start      end        party
  <chr>   <date>     <date>     <chr>
1 Johnson 1963-11-22 1969-01-20 Democratic
2 Nixon   1969-01-20 1974-08-09 Republican
3 Ford    1974-08-09 1977-01-20 Republican
```

```
4 Bush    1989-01-20 1993-01-20 Republican
5 Clinton 1993-01-20 2001-01-20 Democratic
6 Bush    2001-01-20 2009-01-20 Republican
7 Trump   2017-01-20 2021-01-20 Republican
```

- `str_count(string, pattern)`: Counts the number of matches within a string.

```
tibble(name = presidential$name,
   count = str_count(presidential$name, "a|e"))
```

```
# A tibble: 12 x 2
   name       count
   <chr>      <int>
 1 Eisenhower     2
 2 Kennedy        2
 3 Johnson        0
 4 Nixon          0
 5 Ford           0
 6 Carter         2
 7 Reagan         3
 8 Bush           0
 9 Clinton        0
10 Bush           0
11 Obama          2
12 Trump          0
```

- `str_count()` counts non-overlapping matches.

```
str_count("abababa", "aba")
```

```
[1] 2
```

Combining/ Join and Split Strings

Joining Strings

- Combine strings with `str_c()`.

```
    x <- "Faithless is he that says"
    y <- "farewell when the road darkens."
    str_c(x, y)
```

```
[1] "Faithless is he that saysfarewell when the road darkens."
```

- The default is to separate strings by nothing, but you can use `sep` to change the separator.

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

```
[1] "Faithless is he that says farewell when the road darkens."
```

- Just like `c()`, `str_c()` can take multiple arguments.

```
    str_c("Short", "cuts", "make", "long", "delays.", sep = " ")
```

```
[1] "Short cuts make long delays."
```

- If you provide `str_c()` a vector of arguments, it will vectorize the combining unless you provide a `collapse` argument.

```
    x <- c("Short", "cuts", "make", "long", "delays.")
    str_c(x, "LOTR", sep = " ")
```

```
[1] "Short LOTR"   "cuts LOTR"    "make LOTR"    "long LOTR"    "delays. LOTR"
```

```
    x <- c("Short", "cuts", "make", "long", "delays.")
    str_c(x, "LOTR",  collapse = " ")
```

```
[1] "ShortLOTR cutsLOTR makeLOTR longLOTR delays.LOTR"
```

- Combining with `NA` results in `NA`:

```
    str_c("Faithless is he that says", NA)
```

```
[1] NA
```

- **Exercise**: In the `flights` data frame from the nycflights13 package, use string concatenation to create a new variable called `date` which is of the form `"DD-MM-YYYY"`.

```
library(nycflights13)
names(flights)
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
 [9] "arr_delay"      "carrier"        "flight"         "tailnum"
[13] "origin"         "dest"           "air_time"       "distance"
[17] "hour"           "minute"         "time_hour"
```

Joining Strings with Values from Variables or Functions

- `str_glue(..., .sep)` is a wrapper around the `glue::glue()` function which allows you to "interpolate" strings and variables, e.g., combine a string with the dynamic value of a variable.
  - Notice the argument is `.sep` not `sep`
- This can be useful in in-line code and text elements in graphics as an alternative to `paste()` and `paste0()`.

```
str_glue("The mtcars data frame has",
         nrow(mtcars), "rows.", .sep = "")
```

```
The mtcars data frame has32rows.
```

```
str_glue("The mtcars data frame has",
         nrow(mtcars), "rows.", .sep = " ")
```

```
The mtcars data frame has 32 rows.
```

Split a String into Multiple Strings

- `str_split()` will split a string based on a character we choose; useful with dates or full names at times. Can help simplify working with long strings by breaking them into shorter pieces for analysis.
- By default, the output is a list. Have to be careful when the string could split into different numbers of elements.

```
str_split(nycflights13::airports$name, "\\s") %>%
  head()
```

```
[[1]]
[1] "Lansdowne" "Airport"

[[2]]
[1] "Moton"     "Field"     "Municipal" "Airport"

[[3]]
[1] "Schaumburg" "Regional"
```

```
[[4]]
[1] "Randall" "Airport"

[[5]]
[1] "Jekyll"  "Island"  "Airport"

[[6]]
[1] "Elizabethton" "Municipal"    "Airport"
```

plit a Data Frame Column of Type Character with `separate()`

- The `separate()` function, from the {tidyr} package, is useful for splitting all the strings in a data frame column into multiple columns.
- You can use regex or a vector of positions to turn a single column into multiple columns.
- You need to specify at least three arguments:
    1. The column you want to separate that has two (or more) variables,
    2. The character vector of the names of the new variables, and
    3. The character or numeric positions by which to separate out the new variables from the current column.
- Consider {tidyr}'s table three which has cases and country population combined.

```
head(table3) ## from tidyr package
```

```
# A tibble: 6 x 3
  country      year rate
  <chr>       <dbl> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

```
table3  |>
  separate(rate, into = c("cases", "population"),
           sep = "/") %>%
  head()
```

```
# A tibble: 6 x 4
  country      year cases  population
  <chr>       <dbl> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

- See also `tidyr::extract()`.

The `separate()` function has recently been superseded by three new functions in {tidyr}.

- `separate_wider_delim()` splits by delimiter.
- `separate_wider_position()` splits at fixed widths.
- `separate_wider_regex()` splits with regular expression matches.

```
separate_wider_delim(table3, rate,
                     names = c("cases", "population"),
                     delim = "/")
```

```
# A tibble: 6 x 4
  country      year cases  population
  <chr>       <dbl> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

Extracting substrings,

`str_sub(string, start, end)`

- `str_sub()` extracts a substring between the location of two characters.

```
    x <- "The Road goes ever on and on"
    str_sub(x, start = 2, end = 6)
```

```
[1] "he Ro"
```

- Replace substrings with assignment

```
str_sub(x, start = 2, end = 6) <- " Tolkein "
x
```

```
[1] "T Tolkein ad goes ever on and on"
```

- **Exercise**: Reproduce this quote

  > But under a tall tree I will lie, And let the clouds go sailing by. with these strings

```
w <- "But under a tall tree"
x <- "FRELL I will lie"
y <- "and let clouds go"
z <- "sailing by."
```

or

```
x1 <- str_sub(x, -10,-1)
x1
```

```
[1] "I will lie"
```

- You can index from the end of the string using negative indices:

```
x <- "The Road goes ever on and on"
str_sub(x, -9, -1)
```

```
[1] "on and on"
```

`str_subset(string, pattern, negate)`

- `str_sub(string, start, end)` extracts the substring between the location of two characters (inclusive).
- Default values for `start` and `end` are `1L` and `-1L`.

```r
x <- "The Road goes ever on and on"
str_sub(x)
```

```
[1] "The Road goes ever on and on"
```

```r
str_sub(x, start = 3)
```

```
[1] "e Road goes ever on and on"
```

```r
str_sub(x, end = 6)
```

```
[1] "The Ro"
```

```r
str_sub(x, start = 3, end = 6) <- "Tolkien"
x
```

```
[1] "ThTolkienad goes ever on and on"
```

- `str_subset(string, pattern, negate)` returns the strings where there is a match.
- `negate = TRUE` returns those without a match.

```r
str_subset(presidential$name, "(e|a)")
```

```
[1] "Eisenhower" "Kennedy"    "Carter"     "Reagan"     "Obama"
```

```r
str_subset(presidential$name, "(e|a)", negate = TRUE)
```

```
[1] "Johnson" "Nixon"   "Ford"    "Bush"    "Clinton" "Bush"    "Trump"
```

Manage Lengths

- `str_length(string)` returns the count of "characters" (including spaces and punctuation).
- Special characters, starting with the escape symbol \, only count as `1` character.

```
str_length("My apple tastes sweet")
```

```
[1] 21
```

```
str_length("My apple tastes sweet.")
```

```
[1] 22
```

```
str_length("My apple tastes sweet  .")
```

```
[1] 24
```

```
str_length("My apple tastes \nsweet  .")
```

```
[1] 25
```

- `str_trim(string, side)` removes (potentially troublesome) invisible white space on the beginning and/or the end of a string.
- `str_squish(string)` removes invisible spaces at the beginning and end, and also collapses multiple spaces in the middle of a string into one space.

```
x <- " This is a string with   extra whitespaces   in the beginning, middle and end. "
x
```

```
[1] " This is a string with   extra whitespaces   in the beginning, middle and end. "
```

```
str_trim(x)
```

```
[1] "This is a string with   extra whitespaces   in the beginning, middle and end."
```

```
str_squish(x)
```

```
[1] "This is a string with extra whitespaces in the beginning, middle and end."
```

Regular Expressions

- Regular expressions (regex or regexp) is a syntax for pattern matching in strings.
- We'll use `str_replace()` and `str_replace_all()` to demonstrate using regex in stringr. These functions search for a pattern and then replace it with another string.
- But wherever there is a `pattern` argument in a stringr function, you can use regex (to extract strings, get a logical if there is a match, etc…).
- Basic usage: finds exact match of string.

```
x <- "Ho! Ho! Ho! to the bottle I go to heal my heart and drown my woe."
str_replace_all(x, "hea", "XX")
```

```
[1] "Ho! Ho! Ho! to the bottle I go to XXl my XXrt and drown my woe."
```

- A period "." matches any character.

```
str_replace_all(x, "hea.", "XX")
```

```
[1] "Ho! Ho! Ho! to the bottle I go to XX my XXt and drown my woe."
```

- You can "escape" a period with two backslashes "\\" to match periods.

```
str_replace_all(x, ".", "X") ## Matches everything
```

```
[1] "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
```

```
str_replace_all(x, "\\.", "X") ## Matches the only period
```

```
[1] "Ho! Ho! Ho! to the bottle I go to heal my heart and drown my woeX"
```

- To match a backslash, you need four backslashes (to escape the escape).

```
y <- "Rain\\may\\fall\\and\\wind\\may\\blow"
print(y)
```

```
[1] "Rain\\may\\fall\\and\\wind\\may\\blow"
```

```
    writeLines(y)
```

```
Rain\may\fall\and\wind\may\blow
```

```
    y <- "Rain\\may\\fall\\and\\wind\\may\\blow"
    writeLines(y)
```

```
Rain\may\fall\and\wind\may\blow
```

```
    str_replace_all(y, "\\\\", "XX")
```

```
[1] "RainXXmayXXfallXXandXXwindXXmayXXblow"
```

- *Important note*: The actual regular expressions above are strings themselves, and so you view them with `writeLines()`. So using "`\\.`" as the pattern argument in R results in the regular expression "`\.`".
- **Exercise**: Construct a regular expression to match this string: `\.\.\.`

```
\.\.\.
```

- **Exercise**: Use one function call to replace `"back"` and `"lack"` with `"foo"`.

```
    x <- "but better is Beer if drink we lack,
and Water Hot poured down the back."
x
```

```
[1] "but better is Beer if drink we lack,\nand Water Hot poured down the back."
```

Anchoring

- You can anchor the pattern to only match the start or end of a string.
  - `^` matches only the start of a string.
  - `$` matches only the end of a string.

```
    x <- c("But", "under", "a", "tall", "tree", "I", "will", "lie")
    str_replace(x, "^t", "XX")
```

```
[1] "But"   "under" "a"     "XXall" "XXree" "I"     "will"  "lie"
```

```
    str_replace(x, "t$", "XX")
```

```
[1] "BuXX"  "under" "a"     "tall"  "tree"  "I"     "will"  "lie"
```

- Use both to match only a complete string.

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

```
[1] "XX pie"  "XX"      "XX cake"
```

```
    str_replace_all(x, "^apple$", "XX")
```

```
[1] "apple pie"  "XX"         "apple cake"
```

- **Exercise**: Use `str_replace()` to replace all four letter words beginning with an `"a"` with `"foo"` in the following list

```
    x <- c("apple", "barn", "ape", "cart", "alas", "pain", "ally")
```

Special Characters

- We'll use this character vector for practice:

```
    x <- c("Abba: 555-1234", "Anna: 555-0987", "Andy: 555-7654")
```

- `\\d`: matches any digit.

```
    str_replace(x, "\\d\\d\\d-\\d\\d\\d\\d", "XXX-XXXX")
```

```
[1] "Abba: XXX-XXXX" "Anna: XXX-XXXX" "Andy: XXX-XXXX"
```

- `\\s`: matches any white space (e.g. space, tab, newline).

```
    str_replace(x, "\\s", "X")
```

```
[1] "Abba:X555-1234" "Anna:X555-0987" "Andy:X555-7654"
```

- `[abc]`: matches `a`, `b`, or `c`.

```
    str_replace(x, "A[bn][bn]a", "XXXX")
```

```
[1] "XXXX: 555-1234" "XXXX: 555-0987" "Andy: 555-7654"
```

- `[^abc]`: matches anything except `a`, `b`, or `c`.

```
    str_replace(x, "A[^b]", "XXXX")
```

```
[1] "Abba: 555-1234"   "XXXXna: 555-0987" "XXXXdy: 555-7654"
```

- `abc|xyz`: matches either `abc` or `xyz`. This is called *alternation*
- You can use parentheses to control where the alternation occurs.
  - `a(bc|xy)z` matches either `abcz` or `axyz`.

```
    str_replace(x, "An(na|dy)", "XXXX")
```

```
[1] "Abba: 555-1234" "XXXX: 555-0987" "XXXX: 555-7654"
```

- To ignore case, place a `(?i)` before the regex.

```
    str_replace("AB", "ab", "X")
```

```
[1] "AB"
```

```
    str_replace("AB", "(?i)ab", "X")
```

```
[1] "X"
```

- **Exercise**: Create separate regular expressions to find all words that:
  1. Start with a vowel. Test on

```
    x1 <- c("abba", "cat", "eal", "ion", "oops",
            "Uganda", "Anna", "dog")
```

2. That end in consonants. (Hint: thinking about matching "not"-vowels.)
   test on

```
        x2 <- c("bob", "Anna", "dog")
```

3. End with `ed`, but not with `eed`. Test on

```
        x3 <- c("tired", "need", "bad", "rod")
```

4. End with `ing` or `ise`. Test on

```
        x4 <- c("paradise", "firing", "jaded", "kin")
```

Repetition

- Can match a pattern multiple times in a row:
  - ?: 0 or 1
  - +: 1 or more
  - *: 0 or more

```
    x <- c("A", "AA", "AAA", "AAAA", "B", "BB")
    str_replace_all(x, "^A?", "X")
```

```
[1] "X"    "XA"   "XAA"  "XAAA" "XB"   "XBB"
```

```
    str_replace_all(x, "^A+", "X")
```

```
[1] "X"  "X"  "X"  "X"  "B"  "BB"
```

```
    str_replace_all(x, "^A*", "X")
```

```
[1] "X"   "X"   "X"   "X"   "XB"  "XBB"
```

- A more realistic example:

```
    str_replace_all("color and colour", "colou?r", "X")
```

```
[1] "X and X"
```

- Control exactly how many repetitions allowed in a match:
  - `{n}`: exactly `n`.
  - `{n,}`: `n` or more.
  - `{0,m}`: at most `m`.
  - `{n,m}`: between `n` and `m`.

```
    str_replace_all(x, "A{2}", "X")
```

```
[1] "A"  "X"  "XA" "XX" "B"  "BB"
```

```
    str_replace_all(x, "A{2,}", "X")
```

```
[1] "A"  "X"  "X"  "X"  "B"  "BB"
```

```
    str_replace_all(x, "A{0,2}", "X")
```

```
[1] "XX"    "XX"    "XXX"   "XXX"   "XBX"   "XBXBX"
```

```
    str_replace_all(x, "A{3,4}", "X")
```

```
[1] "A"  "AA" "X"  "X"  "B"  "BB"
```

- Regex will automatically match the longest string possible.

```
    str_replace("AAAA", "A*", "X")
```

```
[1] "X"
```

- **Exercise**: Create regular expressions to find all words that:
  1. Start with three consonants. Test on

```
    x1 <- c("string", "priority", "value", "distinction")
```

2. Have three or more vowels in a row. Test on

```
        x2 <- c("honorific", "delicious", "priority", "queueing")
```

```
3. Have two or more vowel-consonant pairs in a row. Test on
```

```
        x3 <- c("honorific", "sam", "prior")
```

Grouping and Backreferences

- Parentheses create a numbered group that you can then back reference with `\\1` for the match in the first parentheses, `\\2` in the second parentheses, etc…

```
    str_replace("cococola", "(..)\\1", "pepsi")
```

```
[1] "pepsicola"
```

```
    str_replace("banana", "([aeiou][^aeiou])\\1", "XX")
```

```
[1] "bXXa"
```

stringr tools

- There are a lot of tools, so we'll go over them briefly and do an exercise where you can use them in more detail.
- `str_to_lower()`, `str_to_upper()`, and `str_to_sentence(x)` convert all letters to lower or capital case.

```
x2 <- "Deeds will Not be LESS vaLiant in the USA because THey are unpraised."
x2
```

```
[1] "Deeds will Not be LESS vaLiant in the USA because THey are unpraised."
```

```
    str_to_lower(x2)
```

```
[1] "deeds will not be less valiant in the usa because they are unpraised."
```

```
    str_to_upper(x2)
```

```
[1] "DEEDS WILL NOT BE LESS VALIANT IN THE USA BECAUSE THEY ARE UNPRAISED."
```

```
    str_to_sentence(x2)
```

```
[1] "Deeds will not be less valiant in the usa because they are unpraised."
```

```
str_to_sentence(x2) %>%
  str_replace("usa", "USA")
```

```
[1] "Deeds will not be less valiant in the USA because they are unpraised."
```

- `str_detect()`: Returns `TRUE` if a regex pattern matches a string and `FALSE` if it does not. Very useful for filters.

```
    library(Lahman)
```

```
Warning: package 'Lahman' was built under R version 4.5.2
```

```
    data(Master)
```

```
Warning in data(Master): data set 'Master' not found
```

```
    ## Get all John's and Joe's from the Lahman dataset
    Master %>%
      filter(str_detect(nameFirst, "^Jo(e|hn)$")) %>%
      select(nameFirst) %>%
      head()
```

```
Error: object 'Master' not found
```

- `str_subset()`: Returns the words where there is a match. Not often as useful as `str_detect()` because you don't use it in data frames that often.

```
    str_subset(Master$nameFirst, "^Jo(e|hn)$") %>%
      head()
```

```
Error: object 'Master' not found
```

- `str_count()`: Counts the occurrence of a match within a string.

```
    str_count(c("banana", "coco"), "[^aeiou][aeiou]")
```

```
[1] 3 2
```

They count *non-overlapping* matches

```
    str_count("abababa", "aba")
```

```
[1] 2
```

- `str_extract()`: Returns the pattern that it finds. `str_extract()` will only return the first match but `str_extract_all()` will return all matches.

```
    colorstr <- str_c("red", "blue", "yellow", "orange", "brown", sep = "|")
    colorstr
```

```
[1] "red|blue|yellow|orange|brown"
```

```
    str_extract("I like blue and brown and that's it", colorstr)
```

```
[1] "blue"
```

```
    str_extract_all("I like blue and brown and that's it", colorstr)
```

```
[[1]]
[1] "blue"  "brown"
```

- `str_match()`: returns a matrix where each column is a grouped component.

```
    x <- "I like blue and brown and that's it, or black"
    str_extract_all(x, "(and|or)\\s([^\\s]+)")
```

```
[[1]]
[1] "and brown"  "and that's" "or black"
```

```
    str_match_all(x, "(and|or)\\s([^\\s]+)")
```

```
[[1]]
     [,1]         [,2]  [,3]
[1,] "and brown"  "and" "brown"
[2,] "and that's" "and" "that's"
[3,] "or black"   "or"  "black"
```

- Let's look at the poem "Farewell We Call to Hearth and Hall!"

```
farewell <- c("Farewell we call to hearth and hall!
              Though wind may blow and rain may fall,
              We must away ere break of day
              Far over wood and mountain tall.")
writeLines(farewell)
```

```
Farewell we call to hearth and hall!
                Though wind may blow and rain may fall,
                We must away ere break of day
                Far over wood and mountain tall.
```

- `str_split()` will split up a string based on a character we choose.

```
## Split based on spaces
str_split(farewell, pattern = "\\s+", simplify = TRUE) ## use one or more space to split
```

```
     [,1]       [,2] [,3]   [,4] [,5]     [,6]  [,7]    [,8]     [,9]   [,10]
[1,] "Farewell" "we" "call" "to" "hearth" "and" "hall!" "Though" "wind" "may"
     [,11]  [,12] [,13]  [,14] [,15]   [,16] [,17]  [,18]  [,19] [,20]   [,21]
[1,] "blow" "and" "rain" "may" "fall," "We"  "must" "away" "ere" "break" "of"
     [,22] [,23] [,24]  [,25]  [,26] [,27]      [,28]
[1,] "day" "Far" "over" "wood" "and" "mountain" "tall."
```

- `str_replace()` and `str_replace_all()` will replace patterns with provided strings. So say we want to get rid of all punctuation.

```
str_split(farewell, pattern = "\\s+", simplify = TRUE) %>%
  str_replace_all("\\.|\\!|,", "")
```

```
 [1] "Farewell" "we"       "call"     "to"       "hearth"   "and"
 [7] "hall"     "Though"   "wind"     "may"      "blow"     "and"
[13] "rain"     "may"      "fall"     "We"       "must"     "away"
[19] "ere"      "break"    "of"       "day"      "Far"      "over"
[25] "wood"     "and"      "mountain" "tall"
```

- You can use back references to populate the replacement.

```
    str_replace_all("It is 1am", "(\\d+)(am|pm)", "\\2")
```

```
[1] "It is am"
```

- More stringr options can be found in RDS.