14 - Strings

Data and Information Engineering

SYS 2202 | Fall 2019

14-strings.pdf

Contents

1 Strings and Characters **2**
- 1.1 Text as Data . . . 2

2 `stringr` R package **3**
- 2.1 Combining and Collapsing Strings . . . 3
- 2.2 String Subsetting . . . 4
- 2.3 String Length . . . 5
- 2.4 String Formatting . . . 6

3 Regular Expressions: Finding Patterns **8**
- 3.1 Literal String Matching . . . 8
- 3.2 Multiple Choice . . . 9
- 3.3 Shortcuts . . . 9
- 3.4 Quantifiers . . . 10
- 3.5 Grouping . . . 10
- 3.6 Greedy Matching . . . 11
- 3.7 Anchors . . . 11

4 String Manipulation **12**
- 4.1 Remove text . . . 12
- 4.2 Finding all words . . . 12

5 Pattern Detection **13**
- 5.1 Finding all words . . . 13
- 5.2 Word Count . . . 13
- 5.3 Baby Names . . . 13

6 Examples **14**
- 6.1 Baseball 3,000 Hit Club . . . 14

Required Packages and Data

```
library(rvest)       # for getting html tables
library(babynames)   # for babynames data
library(tidyverse)   # includes stringr
```

1 Strings and Characters

- A **character** is a single symbol
  - one byte per character
  - letter, number, punctuation, whitespace
  - special characters (`\n`, `\r`, `\t`)
  - encoding
- A **string** is a collection of 0 or more characters
  - `"Bacon and Eggs"` is a string of letters and whitespaces
  - `""` is an empty string
  - `"www.virginia.edu"`
  - `"userid@virginia.edu"`

1.1 Text as Data

Think of text (strings) as data, just like you do for numbers. In R, strings are considered *character* data.

```
x = "Bacon and Eggs"
class(x)
#> [1] "character"
```

In R, the string `x` is a one element vector. Here is a vector of three strings:

```
y = c("Bacon and Eggs", "French Toast", "coffee")
length(y)
#> [1] 3
y[2:3]
#> [1] "French Toast" "coffee"
```

Usually, single or double quotes will suffice.

```
single = 'The fear of the Lord is the beginning of wisdom'
double = "The fear of the Lord is the beginning of wisdom"
identical(single, double)
#> [1] TRUE
```

An exception is with quotations or apostrophes. The `\` is an *escape* symbol that forces the next character to be taken literally.

```
quotes = "She said, \"there\'s is no time like the present\""  # escape \
quotes2 = 'She said, "there\'s is no time like the present"'   # mix quotes
quotes
#> [1] "She said, \"there's is no time like the present\""
cat(quotes)                              # cat() displays in readable form
#> She said, "there's is no time like the present"
identical(quotes, quotes2)
#> [1] TRUE
```

The `\` can also be used to indicate special things like tab (`\t`)

```
cat("there\'s no \time like the present")
#> there's no   ime like the present
```

2 `stringr` R package

- Website: https://stringr.tidyverse.org/
- `stringr` cheatsheet: https://github.com/rstudio/cheatsheets/blob/master/strings.pdf
- Textbook Reading R4DS String

The `stringr` packages is part of tidyverse and loaded automatically, so you only need to load tidyverse and all `stringr` functions are available.

```
library(tidyverse)
#library(stringr)  # part of tidyverse and loaded automatically
```

From the Intro to `stringr` vignette:

> Strings are not glamorous, high-profile components of R, but they do play a big role in many data cleaning and preparations tasks. R provides a solid set of string operations, but because they have grown organically over time, they can be inconsistent and a little hard to learn. Additionally, they lag behind the string operations in other programming languages, so that some things that are easy to do in languages like Ruby or Python are rather hard to do in R. The **stringr** package aims to remedy these problems by providing a clean, modern interface to common string operations.
> More concretely, stringr:
>
> - Simplifies string operations by eliminating options that you don't need 95% of the time (the other 5% of the time you can functions from base R or stringi).
> - Uses consistent function names and arguments.
> - Produces outputs than can easily be used as inputs. This includes ensuring that missing inputs result in missing outputs, and zero length inputs result in zero length outputs. It also processes factors and character vectors in the same way.
> - Completes R's string handling functions with useful functions from other programming languages.
>
> To meet these goals, stringr provides two basic families of functions:
>
> - basic string operations, and
> - pattern matching functions which use regular expressions to detect, locate, match, replace, extract, and split strings.
>
> As of version 1.0, stringr is a thin wrapper around stringi, which implements all the functions in stringr with efficient C code based on the ICU library. Compared to stringi, stringr is considerably simpler: it provides fewer options and fewer functions. This is great when you're getting started learning string functions, and if you do need more of stringi's power, you should find the interface similar.

2.1 Combining and Collapsing Strings

function	description
`str_c()`	Join multiple strings into a single string
`paste()`	Concatenate vectors after converting to character
`str_dup()`	Duplicate and concatenate strings within a character vector

The `stringr` function `str_c()` is basically the base R function `paste0()`. We will use `str_c()` just to stay consistent.

2.1.1 Combining vectors into strings

```
str_c("A", "B", "C")
#> [1] "ABC"
```

```
#- change the sep= argument to a space
str_c("A", "B", "C", sep=" ")
#> [1] "A B C"

#- change the sep= argument to a :
str_c("A", "B", "C", sep= ":")
#> [1] "A:B:C"
```

Vectorized (remember vector recycling)

```
str_c("X", 1:5)
#> [1] "X1" "X2" "X3" "X4" "X5"
ABC = c("A", "B", "C")
str_c(ABC, 1:3)
#> [1] "A1" "B2" "C3"
str_c(ABC, 1:3, sep='-')
#> [1] "A-1" "B-2" "C-3"
str_c(LETTERS[1:6], 1:3, sep='-')
#> [1] "A-1" "B-2" "C-3" "D-1" "E-2" "F-3"
```

```
str_c("---", "Iteration: ", 1:10, "---" )
#>  [1] "---Iteration: 1---"  "---Iteration: 2---"  "---Iteration: 3---"
#>  [4] "---Iteration: 4---"  "---Iteration: 5---"  "---Iteration: 6---"
#>  [7] "---Iteration: 7---"  "---Iteration: 8---"  "---Iteration: 9---"
#> [10] "---Iteration: 10---"
```

There is also the `str_dup()` function (compare to `rep()`)

```
str_dup(ABC, times = 2)
#> [1] "AA" "BB" "CC"
str_dup(ABC, times = 1:3)
#> [1] "A"   "BB"  "CCC"

rep(ABC, times=1:3)
#> [1] "A" "B" "B" "C" "C" "C"
rep(ABC, each=2)
#> [1] "A" "A" "B" "B" "C" "C"
```

2.1.2 Make one long string

The `collapse=` argument of `str_c()` (and `paste()`),

```
str_c(ABC)                 # three element vector
#> [1] "A" "B" "C"
str_c(ABC, collapse="")    # one element vector
#> [1] "ABC"
str_c(ABC, collapse=", ")  # one element vector
#> [1] "A, B, C"
```

2.2 String Subsetting

function	description
`str_sub()`	Extract and replace substrings from a character vector

We have already used the function `str_sub()` to subset strings by position.

```
data(state)  # access to state.name data
str_sub(state.name, start=1, end=4)    # 1st 4 letters
#>  [1] "Alab" "Alas" "Ariz" "Arka" "Cali" "Colo" "Conn" "Dela" "Flor" "Geor"
#> [11] "Hawa" "Idah" "Illi" "Indi" "Iowa" "Kans" "Kent" "Loui" "Main" "Mary"
#> [21] "Mass" "Mich" "Minn" "Miss" "Miss" "Mont" "Nebr" "Neva" "New " "New "
#> [31] "New " "New " "Nort" "Nort" "Ohio" "Okla" "Oreg" "Penn" "Rhod" "Sout"
#> [41] "Sout" "Tenn" "Texa" "Utah" "Verm" "Virg" "Wash" "West" "Wisc" "Wyom"
str_sub(state.name, start=-4, end=-1)  # last 4 letters
#>  [1] "bama" "aska" "zona" "nsas" "rnia" "rado" "icut" "ware" "rida" "rgia"
#> [11] "waii" "daho" "nois" "iana" "Iowa" "nsas" "ucky" "iana" "aine" "land"
#> [21] "etts" "igan" "sota" "ippi" "ouri" "tana" "aska" "vada" "hire" "rsey"
#> [31] "xico" "York" "lina" "kota" "Ohio" "homa" "egon" "ania" "land" "lina"
#> [41] "kota" "ssee" "exas" "Utah" "mont" "inia" "gton" "inia" "nsin" "ming"
```

2.3 String Length

function	description
`str_length()`	The length of a string

The function `str_length()` returns the number of characters (or *code points*) in a string:

```
tibble(state=state.name, letters = str_length(state.name))
#> # A tibble: 50 x 2
#>    state      letters
#>    <chr>        <int>
#> 1 Alabama          7
#> 2 Alaska           6
#> 3 Arizona          7
#> 4 Arkansas         8
#> 5 California      10
#> 6 Colorado         8
#> # ... with 44 more rows
```

2.3.1 Characters in Colossians

We will further illustrate with a text version of the Book of Colossians from the bible. The text can be found here https://mdporter.github.io/SYS2202/data/colossians.txt.

```
url = "https://mdporter.github.io/SYS2202/data/colossians.txt"
lines = read_lines(url)       # vector: each element a string
single = read_file(url)       # single element: string of entire file
```

Notice that the vector `lines` has one element per line in the document, while `single` is one long string.

Let's see how many characters are in the document

```
str_length(lines) %>% head()    # notice there are empty lines
#> [1]  91   0 116   0  93   0
sum(str_length(lines)) # total number of characters
#> [1] 9527
str_length(single)      # number of characters in long string
#> [1] 9679
```

Why do the number of characters not match?

Because `single` has new line (`\n`) and carriage return (`\r`) symbols.

```
lines[1:3]
#> [1] "[Col 1:1 ESV] Paul, an apostle of Christ Jesus by the will of God, and Timothy our brother,"
#> [2] ""
#> [3] "[Col 1:2 ESV] To the saints and faithful brothers in Christ at Colossae: Grace to you and pea
str_sub(single, start=1, end=211)
#> [1] "[Col 1:1 ESV] Paul, an apostle of Christ Jesus by the will of God, and Timothy our brother,\r
```

We can remove these with the `str_remove_all()` function.

```
single2 = str_remove_all(string=single, pattern="\\n|\\r")
str_length(single2)
#> [1] 9527
sum(str_length(lines))
#> [1] 9527
str_sub(single2, 1, 211)
#> [1] "[Col 1:1 ESV] Paul, an apostle of Christ Jesus by the will of God, and Timothy our brother,[C
```

The `|` symbol in `pattern="\\n|\\r"` means we want to remove all new line (`\n`) and carriage return (`\r`).

Side note: we can collapse the line-by-line vector `lines` to one long string.

```
lines2 = str_c(lines, collapse="")
identical(lines2, single2)
#> [1] TRUE
```

2.4 String Formatting

function	description
`str_wrap()`	Wrap strings into nicely formatted paragraphs
`str_trunc()`	Truncate a character string
`str_trim()`	Trim whitespace from start and end of string
`str_pad()`	Pad a string
`str_to_lower()`	Convert string to all lowercase: see `tolower()`
`str_to_upper()`	Convert string to all uppercase: see `toupper()`
`str_to_title()`	Convert string to all title case
`str_to_sentence(0)`	Convert string to sentence case
`str_conv()`	Specify the encoding of a string

To help see long strings, the `str_wrap()` and `str_trunc()` are helpful

```
writeLines(lines[1])                  # writeLines() displays in readable form
#> [Col 1:1 ESV] Paul, an apostle of Christ Jesus by the will of God, and Timothy our brother,
writeLines(str_wrap(lines[1], width=50)) # adds newlines '\n'
#> [Col 1:1 ESV] Paul, an apostle of Christ Jesus by
#> the will of God, and Timothy our brother,
writeLines(str_wrap(lines[1], width=1))  # special treament for width<=1
#> [Col
#> 1:1
#> ESV]
#> Paul,
#> an
#> apostle
#> of
#> Christ
```

```
#> Jesus
#> by
#> the
#> will
#> of
#> God,
#> and
#> Timothy
#> our
#> brother,
str_trunc(lines[1], width = 30)      # reduces string to only `width` characters
#> [1] "[Col 1:1 ESV] Paul, an apos..."
```

The functions `str_trim()` and `str_pad()` help with whitespace and length

```
str_trim("  String with trailing and leading white space\t")
#> [1] "String with trailing and leading white space"
str_trim("\n\nString with trailing and leading white space\n\n")
#> [1] "String with trailing and leading white space"
```

```
x = c("A", "BB", "CCC", "DDDD")
(x.pad = str_pad(x, width=4, side="right"))  # `width` is minimum width
#> [1] "A   " "BB  " "CCC " "DDDD"
str_length(x.pad)
#> [1] 4 4 4 4
str_pad(x, width=4, side="right", pad="-")
#> [1] "A---" "BB--" "CCC-" "DDDD"
```

2.4.1 Convert the case of a string

For extracting words from text, we often want to remove the effects of case.

```
s1 = "This is not the same sentence"
s2 = "this is NOT the same sentence"

#- identical when all lowercase
str_to_lower(s1)
#> [1] "this is not the same sentence"
str_to_lower(s2)
#> [1] "this is not the same sentence"

#- identical when all uppercase
str_to_upper(s1)
#> [1] "THIS IS NOT THE SAME SENTENCE"
str_to_upper(s2)
#> [1] "THIS IS NOT THE SAME SENTENCE"

#- title case example
str_to_title(s1)
#> [1] "This Is Not The Same Sentence"

#- Sentence case example
str_to_sentence(s2)
#> [1] "This is not the same sentence"
```

3 Regular Expressions: Finding Patterns

Regular expressions, *regex* for short, are a very terse language that allow to describe patterns in strings. They take a little while to get your head around, but once you've got it you'll find them extremely useful. We have only used the most basic patterns so far:

- remove the `and` from `'Bacon and Eggs'`

```
y = 'Bacon and Eggs'
str_remove(y, pattern = " and")  # removed `whitespace + and`
#> [1] "Bacon Eggs"
```

- split the first and last names by finding a whitespace

```
student <- c("John Davis","Angela Williams","Bullwinkle Moose",
             "David Jones","Janice Markhammer",
             "Cheryl Cushing","Reuven Ytzrhak",
             "Greg Knox","Joel England","Mary Rayburn")
str_split_fixed(student, " ", n=2)
#>       [,1]         [,2]
#>  [1,] "John"       "Davis"
#>  [2,] "Angela"     "Williams"
#>  [3,] "Bullwinkle" "Moose"
#>  [4,] "David"      "Jones"
#>  [5,] "Janice"     "Markhammer"
#>  [6,] "Cheryl"     "Cushing"
#>  [7,] "Reuven"     "Ytzrhak"
#>  [8,] "Greg"       "Knox"
#>  [9,] "Joel"       "England"
#> [10,] "Mary"       "Rayburn"
```

In these examples, the pattern was an actual symbol or set of symbols. But we need to be able to find patterns in a string (e.g., an uppercase letter followed by one or more lowercase, an optional three digit area code followed by 7 numbers, an email address). Regular expressions gives us a very flexible way to specify patterns.

3.1 Literal String Matching

Consider the quote from Albert Einstein, "Not everything that can be counted counts, and not everything that counts can be counted."

```
#- literal strings must be on one line, or R will insert a new line (\n) character
x = "Not everything that can be counted counts, and not everything that counts can be counted."
```

Note: if you want to write long string on multiple lines, use another function line `strwrap()`.

Regular expressions will find all consecutive characters that match the pattern. The simplest example is when the pattern is a usual string:

```
str_view_all(x, 'counts')     # finds all `counts`
str_view_all(x, 'counted')    # finds all `counted`
str_view_all(x, 'count')      # finds the `count` in `counted` and `counts`
str_view_all(x, 'c')          # finds all `c`
str_view_all(x, 'a')          # finds all `a`
str_view_all(x, ' ')          # finds all whitespaces
```

```
str_view_all(x, 'can be')      # finds the `can be` string
str_view_all(x, 'not')         # finds all `not` with lowercase `n`
str_view_all(x, 'Not')         # finds all `Not` with uppercase `N`
str_view_all(x, 'not ever')    # does this find what you expect?
```

Notice that the pattern `'count'` finds the sequence 'c', followed by a 'o', followed by a 'u', followed by a 'n', followed by a 't'.

3.2 Multiple Choice

You can specify multiple options (logical OR) with a vertical bar `|`. For example, to match 'Not' *or* 'not' we use the pattern `'Not|not'`

```
str_view_all(x, pattern='Not|not')
```

3.2.1 Character Classes: Pick one

For *single characters*, you can indicate multiple options by enclosing in brackets (`[ ]`). For example, the previous example can also be obtained

```
str_view_all(x, pattern='[Nn]ot')   # matches 'N' or 'n', followed by 'ot'
str_view_all(x, pattern='[aeiou]')  # find all lowercase vowels
```

If the first character is the caret (`^`), then it finds any character **not** in the list. For example, to find all (lowercase) consonants

```
str_view_all(x, pattern='[^aeiou]')  # find all lowercase consonants (not vowels)
```

3.3 Shortcuts

Some fruit and phone numbers:

```
fruit = c("apple", "banana", "pear", "pineapple")

strings = c(
  "apple",
  "219 733 8965",
  "329-293-8753",
  "Work: 579-499-7527; Home: 543.355.3679")
```

To help with common patterns, most regular expression engines recognize shortcuts. For example, ranges of characters are also recognized in character classes.

- `[a-d]` matches the letters 'a' or 'b' or 'c' or 'd'
- `[A-Z]` matches all uppercase letters
- `[0-5]` matches the numbers '0' or '1' or '2' ... or '5'

```
str_view_all(strings, pattern='[7-9]')
```

Other special shortcuts include:

- The period (`.`) matches any single character (including whitespace, but not newline)

```
str_view_all(fruit, pattern='a.')  # finds 'a' followed by anything
str_view_all(x, pattern='t.')      # finds 't' followed by anything
```

- Type `?regex` to see the following shortcuts (there are more than these)
  - `[:digit:]` (or `\d`) is same as `[0-9]`

  - `[:lower:]` and `[:upper:]`, the upper and lower case letters. This is generally the same as `[a-z]` and `[A-Z]` expect the former are locale independent.
  - `[:alpha:]` all letters of the alphabet. Union of `[:lower:]` and `[:upper:]`
  - `[:alnum:]` the alphanumeric characters. Union of `[:alpha:]` and `[:digit:]` and same as `[0-9A-Za-z]`
  - `[:punct:]` all the punctuation characters : `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ { | } ~.‘`
  - `[:space:]` (also `\s`) all the whitespace characters: tab, newline, space, carriage return

Some other common shortcuts are:

- `\w` stands for the word characters `[A-Za-z0-9_]` or `[[:alnum:]_]` (alphanumeric *union* underscore)
- `\W` is anything but a word character
- `\d` any digit (same as `[:digit:]` and `[0-9]`)
- `\D` any non-digit
- `\s` any whitespace
- `\S` any non-whitespace
- `\w` any word character
- `\b` boundary
- We have already seen the new line (`\n`) and carriage return (`\r`)

But notice to use these in R, we need to use a double backslash because the first one is treated as an escape

```
str_view_all(strings, pattern='\\d')  # finds digits
str_view_all(strings, pattern='\\s')  # finds all whitespaces
str_view_all(strings, pattern='[\\s[:upper:]]') # whitespaces OR uppercase
```

3.4 Quantifiers

Quantifiers allow control over how many times a pattern is matched.

- `?` (zero or one time)
- `*` (zero or more times)
- `+` (one or more times)
- `{n}` (exactly n times)
- `{n,}` (at least n times)
- `{,m}` (at most m times)
- `{n,m}` (at least n times but no more than m times)

```
#- find the phone numbers
str_view_all(strings, pattern='\\d{3}-\\d{3}-\\d{4}')
str_view_all(strings, pattern='\\d{3}[- \\.]\\d{3}[- \\.]\\d{4}')
```

The first one finds the pattern `XXX-XXX-XXXX` where `X` is any digit. The second allows several separators.

3.5 Grouping

A quantifier modifies the character (or shortcut) to its immediate left. So the pattern `0abc+0` matches ‘0abc0’, ‘0abcc0’, ‘0abccc0’, etc. but not ‘0abcabc0’, ‘0abcabcabc0’, etc. To get this behavior just wrap in parentheses. E.g., `0(abc)+0` will find `abc` one or more times.

```
y = c("0abc0", "0abcc0", "0abccc0", "0abcabc0", "0abcabcabc0")
```

```
str_view_all(y, pattern="0abc+0")
str_view_all(y, pattern="0(abc)+0")
```

3.6 Greedy Matching

Regular expressions will match as much as it can. For example, say we want to find the `ggplot()` function in an R script.

```
R = "ggplot(cars, aes(reorder(type, enginesize)))+geom_bar()+facet_wrap(~origin) +
labs(x='car type')"
```

The pattern is `ggplot` followed by a `(`, the varying set of arguments, then ends in a `)`. We may want to try

```
str_view_all(R, pattern='ggplot\\(.*\\)')     # entire string matched
```

Notice that it matches to the last `)` in the entire string. This is technically correct, but not what we want.

To match the first occurrence, we can add an extra `?` to the quantifier. So for example,

```
str_view_all(R, pattern='ggplot\\(.*?\\)')     # finds first closing parenthesis
str_view_all(R, pattern='ggplot\\(.*?\\)+')    # finds first set of closing parentheses
```

The first example stops too early. We really want to match to the last of a set of closing parentheses.

3.7 Anchors

Regular expressions match any part of a string. So the pattern `apple` matches the apple in the fruit 'apple' as well as the last part of 'pineapple'

```
str_view_all(fruit, pattern='apple')  # finds all 'apple' substrings
```

We can limit to finding all strings that start with 'apple' by using the `^`

```
str_view_all(fruit, pattern='^apple')  # finds strings that *start* with `apple`
```

The special pattern `^` matches only at the beginning of your input and `$` to match only at the end. This allows *bookends* where you can specify what is at the beginning and end.

```
str_view_all(fruit, pattern='apple$')  # finds strings that *end* with `apple`
```

4 String Manipulation

function	description
`str_remove()`	Remove matched patterns in a string
`str_replace()`	Replace matched patterns in a string
`str_replace_na()`	Turn NA into "NA"
`str_sub()`	Extract and replace substrings from a character vector
`str_split()`	Split up a string into pieces
`str_split_fixed()`	Split up a string into exactly n pieces

- Many of these functions have a version to a find *all* matching patterns using `str_<function>_all`.

Recall the bible verses from the Book of Colossians

```
url = "https://mdporter.github.io/SYS2202/data/colossians.txt"
lines = read_lines(url)       # vector: each element a string
single = read_file(url)       # single element: string of entire file
```

Notice that the vector `lines` has one element per line in the document, while `single` is one long string.

4.1 Remove text

We have already seen most of these functions in action. Let's use `str_remove_all()` to remove the verse reference.

```
col_text = str_remove_all(single, pattern='\\[.+? ESV\\]')
str_trunc(col_text, 50)
#> [1] " Paul, an apostle of Christ Jesus by the will o..."
```

4.2 Finding all words

One useful task is to extract all words from a string. We can use the `str_split()` function to break a string up into components. The helper function `boundary(type = "word")` will split a string into word components, dropping the punctuation.

```
str_split(col_text, " ")[[1]] %>% head(30)
#>  [1] ""             "Paul,"        "an"           "apostle"
#>  [5] "of"           "Christ"       "Jesus"        "by"
#>  [9] "the"          "will"         "of"           "God,"
#> [13] "and"          "Timothy"      "our"          "brother,\n\n"
#> [17] "To"           "the"          "saints"       "and"
#> [21] "faithful"     "brothers"     "in"           "Christ"
#> [25] "at"           "Colossae:"    "Grace"        "to"
#> [29] "you"          "and"
str_split(col_text, boundary("word"))[[1]] %>% head(30)
#>  [1] "Paul"     "an"       "apostle"  "of"       "Christ"   "Jesus"
#>  [7] "by"       "the"      "will"     "of"       "God"      "and"
#> [13] "Timothy"  "our"      "brother"  "To"       "the"      "saints"
#> [19] "and"      "faithful" "brothers" "in"       "Christ"   "at"
#> [25] "Colossae" "Grace"    "to"       "you"      "and"      "peace"
```

Notice that `str_split()` will return a length one *list* - and we need to extract the first component using `[[1]]` (which will return a vector of words).

5 Pattern Detection

function	description
`str_detect()`	Detect the presence or absence of a pattern in a string
`str_count()`	Count the number of matches in a string
`str_extract()`	Extract matching patterns from a string
`str_match()`	Extract matched groups from a string
`str_subset()`	Keep strings matching a pattern
`str_locate()`	Locate the position of patterns in a string

- Many of these functions have a version to a find *all* matching patterns using `str_<function>_all`.

5.1 Finding all words

We can also use `str_extract()` to get all the words from a string.

```
str_extract_all(col_text, boundary("word"))[[1]] %>% head()
#> [1] "Paul"    "an"      "apostle" "of"      "Christ"  "Jesus"
```

5.2 Word Count

The function `str_count()` counts the number of times the pattern is matched. Here we will extract the number of times the pattern `God` is found

```
str_count(single, 'God')
#> [1] 20
```

Notice that the *pattern* `God` would match `God's`, `Godspeed`, etc.

We can get a count of all words using the combination of `str_split()`, `tibble()` and `count()`

```
#- get the words from Book of Colossians
col_words = str_split(col_text, boundary("word"))[[1]] # vector of words

#- Get frequency of words
word_freq = tibble(word=col_words) %>% count(word, sort=TRUE)
```

But here we see that `God` is matched only 18 times (not 20). This is because our regular expression pattern matched multiple *words*.

```
filter(word_freq, str_detect(word, 'God'))
#> # A tibble: 2 x 2
#>   word      n
#>   <chr> <int>
#> 1 God      18
#> 2 God's     2
```

5.3 Baby Names

The function `str_subset()`, which is equivalent to `x[str_detect(x, pattern)]`, returns the strings with matching pattern.

Use the `babynames` data from the `babynames` package to find variations of your name.

```
library(babynames)
```

```r
#- get names from 2014 (only most common names included)
baby2017 = filter(babynames, year==2017)
```

```r
library(stringr)

#- return matching strings
str_subset(baby2017$name, "Michael")
#> [1] "Michaela"      "Michael"       "Michaella"     "Michael"
#> [5] "Michaelangelo" "Michaeljames"
str_subset(baby2017$name, "michael")
#>  [1] "Johnmichael"   "Jamichael"     "Kingmichael"   "Lamichael"
#>  [5] "Carmichael"    "Demichael"     "Jonmichael"    "Princemichael"
#>  [9] "Seanmichael"   "Sirmichael"    "Liammichael"
str_subset(baby2017$name, "(M|m)ichael")   # contains Michael or michael
#>  [1] "Michaela"      "Michael"       "Michaella"     "Michael"
#>  [5] "Johnmichael"   "Michaelangelo" "Jamichael"     "Kingmichael"
#>  [9] "Lamichael"     "Carmichael"    "Demichael"     "Jonmichael"
#> [13] "Princemichael" "Seanmichael"   "Sirmichael"    "Liammichael"
#> [17] "Michaeljames"
str_subset(baby2017$name, "(M|m)ichael$")  # ends in Michael or michael
#>  [1] "Michael"       "Michael"       "Johnmichael"   "Jamichael"
#>  [5] "Kingmichael"   "Lamichael"     "Carmichael"    "Demichael"
#>  [9] "Jonmichael"    "Princemichael" "Seanmichael"   "Sirmichael"
#> [13] "Liammichael"
str_subset(baby2017$name, "^(M|m)ichael")  # begins with Michael or michael
#> [1] "Michaela"      "Michael"       "Michaella"     "Michael"
#> [5] "Michaelangelo" "Michaeljames"
str_subset(baby2017$name, "^(M|m)ichael$") # begins and ends in Michael or michael
#> [1] "Michael" "Michael"

#- find matching records in database
baby2017 %>% filter(str_detect(name, pattern="^(M|m)ichael$"))
#> # A tibble: 2 x 5
#>    year sex   name        n      prop
#>   <dbl> <chr> <chr>   <int>     <dbl>
#> 1  2017 F     Michael    33 0.0000176
#> 2  2017 M     Michael 12579 0.00641
```

6 Examples

6.1 Baseball 3,000 Hit Club

```r
library(rvest)
url = 'http://en.wikipedia.org/wiki/3,000_hit_club'
hits = read_html(url) %>% html_node("table.wikitable.sortable") %>% html_table()
# head(hits)
```

6.1.1 Remove non-numeric from `Hits`

```r
#- We used the readr function parse_number()
parse_number(hits$Hits)
#>  [1] 4256 4191 3771 3630 3514 3465 3430 3419 3319 3315 3283 3255 3252 3202
#> [15] 3184 3166 3154 3152 3142 3141 3115 3110 3089 3060 3055 3053 3023 3020
#> [29] 3011 3010 3007 3000

#- but we could use a combination of str_extract and str_remove
```

```r
str_extract(hits$Hits, pattern='\\d+,*\\d+') %>%
  str_remove_all(pattern=',') %>%
  as.numeric()
#>  [1] 4256 4191 3771 3630 3514 3465 3430 3419 3319 3315 3283 3255 3252 3202
#> [15] 3184 3166 3154 3152 3142 3141 3115 3110 3089 3060 3055 3053 3023 3020
#> [29] 3011 3010 3007 3000
```

6.1.2 Extract the `Date` of the 3,000th hit

```r
#- Extract with regex
str_extract(hits$Date, pattern='[A-Z][a-z]+ \\d{1,2}, \\d{4}')
#>  [1] "May 5, 1978"       "August 19, 1921"   "May 17, 1970"
#>  [4] "May 13, 1958"      "May 17, 1925"      "July 9, 2011"
#>  [7] "June 9, 1914"      "September 12, 1979" "September 16, 1996"
#> [10] "June 3, 1925"      "July 18, 1970"     "June 30, 1995"
#> [13] "September 27, 1914" "May 4, 2018"      "April 15, 2000"
#> [16] "July 30, 2017"     "September 30, 1992" "June 19, 1942"
#> [19] "September 9, 1992" "August 6, 1999"    "June 19, 2015"
#> [22] "September 16, 1993" "August 7, 2016"    "June 28, 2007"
#> [25] "October 7, 2001"   "August 4, 1985"    "August 13, 1979"
#> [28] "July 15, 2005"     "July 18, 1897"     "August 7, 1999"
#> [31] "September 24, 1974" "September 30, 1972"
```

6.1.3 Get range of seasons played (`Seasons`)

```r
#- One way is to use str_extract_all(). From this it
#  not be too difficult to get the
#  *number of seasons* played
years = '(18|19|20)\\d{2}'
str_extract_all(hits$Seasons, pattern=years, simplify=TRUE)
#>       [,1]   [,2]   [,3]   [,4]
#>  [1,] "1963" "1986" ""     ""
#>  [2,] "1905" "1928" ""     ""
#>  [3,] "1954" "1976" ""     ""
#>  [4,] "1941" "1944" "1946" "1963"
#>  [5,] "1907" "1928" ""     ""
#>  [6,] "1995" "2014" ""     ""
#>  [7,] "1897" "1917" ""     ""
#>  [8,] "1961" "1983" ""     ""
#>  [9,] "1978" "1998" ""     ""
#> [10,] "1906" "1930" ""     ""
#> [11,] "1951" "1952" "1954" "1973"
#> [12,] "1977" "1997" ""     ""
#> [13,] "1896" "1916" ""     ""
#> [14,] "2001" ""     ""     ""
#> [15,] "1981" "2001" ""     ""
#> [16,] "1998" "2018" ""     ""
#> [17,] "1973" "1993" ""     ""
#> [18,] "1926" "1945" ""     ""
#> [19,] "1974" "1993" ""     ""
#> [20,] "1982" "2001" ""     ""
#> [21,] "1994" "2013" "2015" "2016"
#> [22,] "1973" "1995" ""     ""
#> [23,] "2001" "2019" ""     ""
#> [24,] "1988" "2007" ""     ""
#> [25,] "1979" "2003" ""     ""
```

```
#> [26,] "1967" "1985" ""     ""
#> [27,] "1961" "1979" ""     ""
#> [28,] "1986" "2005" ""     ""
#> [29,] "1871" "1897" ""     ""
#> [30,] "1982" "1999" ""     ""
#> [31,] "1953" "1974" ""     ""
#> [32,] "1955" "1972" ""     ""
```

6.1.4 What type of hit was the 3,000th

```
str_extract(hits$`3,000th hit`, "Single|Double|Triple|Home run")
#>  [1] "Single"   "Single"   "Single"   "Double"   "Single"   "Home run"
#>  [7] "Double"   "Single"   "Triple"   "Single"   "Single"   "Single"
#> [13] "Double"   "Double"   "Single"   "Double"   "Single"   "Single"
#> [19] "Single"   "Single"   "Home run" "Single"   "Triple"   "Single"
#> [25] "Double"   "Single"   "Single"   "Double"   "Single"   "Home run"
#> [31] "Double"   "Double"
```