

**Hacettepe University  
Department of Industrial Engineering  
Undergraduate Program  
2023-2024 Fall**

**EMU 430 – Data Analytics  
Week 11  
December 15, 2023**

**Instructor: Erdi Dasdemir**

[edasdemir@hacettepe.edu.tr](mailto:edasdemir@hacettepe.edu.tr)  
[www.erdidasdemir.com](http://www.erdidasdemir.com)


**String Processing**



**stringr Package**



**Case study - Heights  
and  
Regular expressions**



# Acknowledgment

I drew inspiration primarily from [Dr. Rafael Irizarry's "Introduction to Data Science" Book](#) and ["Data Science" course by HarvardX on edX](#) for the slides this week.

# String Processing

Common challenges in data wrangling are

- extracting numeric data contained in character strings,
- processing unorganized text into meaningful variable names or categorical variables.

Remember web scraping murders' data set. If you check the classes of population and total, you will see that they are character.

```
library(rvest)
library(tidyverse)
url <-
"https://en.wikipedia.org/w/index.php?title=Gun_violence_in_the_United_States_by_sta
te&direction=prev&oldid=810166167"

data_html <- read_html(url)
class(data_html)
[1] "xml_document" "xml_node"
tab <- data_html |> html_nodes("table")
tab <- tab[[1]]
tab <- tab |> html_table()
murders_raw <- tab |> setNames(c("state", "population", "total", "murder_rate"))

class(murders_raw$population)
[1] "character"
class(murders_raw$total)
[1] "character"
```

- This is very common web scraping, since web pages and other formal documents use commas in numbers to improve readability.
- String processing challenges a data scientist faces are unique and often unexpected.
- It is not possible to cover everything but we will try to learn how to approach some of the common tasks:
  - removing unwanted characters from text
  - extracting numerical values from texts
  - finding and replacing characters
  - extracting specific parts of strings
  - converting free-form text to more uniform formats
  - splitting strings into multiple values.

To define strings in R, we can use either double quotes or single quotes:

```
my_favorite_course <- "emu430"
```

```
my_favorite_course <- 'emu430'
```

Now, what happens if the string we want to define includes double quotes? For example, if we want to write

```
emu430's students or 10"?
```

We can use escaping with the backslash / .

```
emu430's students
```

```
10"
```

Escaping characters is something we often have to use when processing strings.



# stringr Package

- In general, string processing involves a string and a pattern.

```
murders_raw$population[1:3]
```

```
[1] "4,853,875" "737,709"   "6,817,565"
```

```
as.numeric(murders_raw$population[1:3])
```

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

- This is because of the commas. The string processing we want to do here is to remove the pattern comma from.
- We need to locate the comma and replace them with an empty character.

- Base R includes function to perform all these tasks.
- They don't follow a unifying convention, which makes it a bit hard to memorize and use.
- The stringr package basically repackages this functionality, but using a more consistent approach of naming functions

stringr	Task	Description	Base R
str_detect	Detect	Is the pattern in the string?	grepl
str_which	Detect	Returns the index of entries that contain the pattern.	grep
str_subset	Detect	Returns the subset of strings that contain the pattern.	grep with value = TRUE
str_locate	Locate	Returns positions of first occurrence of the pattern in a string.	regexpr
str_locate_all	Locate	Returns position of all occurrences of the pattern in a string.	gregexpr

- In general, string processing involves a string and a pattern.

```
murders_raw$population[1:3]
```

```
[1] "4,853,875" "737,709"   "6,817,565"
```

```
as.numeric(murders_raw$population[1:3])
```

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

- This is because of the commas. The string processing we want to do here is to remove the pattern comma from.
- We need to locate the comma and replace them with an empty character.

In stringr, Functions start with str\_, which means that type it and then hit Tab on keyboard,

```
murders_raw$population[1:3]
```

```
as.numeric(murders_raw$population[1:3])
```

```
murders_raw$population |> str_detect(",")
```

```
murders_raw$population |> str_replace_all(",", "", "") |> as.numeric()
```

```
as.numeric(str_replace_all(murders_raw$population, ",", "", ""))
```

#as this operation is so common, there is a function in readr package:

```
parse_number(murders_raw$population)
```

# Case study - Heights and Regular expressions

The **dslabs** package includes the raw data from which the heights dataset was obtained. These heights were obtained using a web form in which students were asked to enter their heights.

```
library(dslabs)
head(reported_heights)
class(reported_heights$height)

# if we try to parse it into numbers, we get a warning:
x <- as.numeric(reported_heights$height)

# we also do end up with many NAs:
sum(is.na(x))

# Here are some of the entries that are not successfully converted:
reported_heights |>
  mutate(new_height = as.numeric(height)) |>
  filter(is.na(new_height)) |>
  head(n = 10)
```

- For example, in the output above, we see various cases that use the format x ' y" or x ' y' ' with x and y representing feet and inches, respectively.
- We can find the number of problematic entries:

```
problems <- reported_heights |>
  mutate(inches = suppressWarnings(as.numeric(height))) |>
  filter(is.na(inches) | inches < 50 | inches > 84) |>
  pull(height)

length(problems)

# 50 inches is 127 centimeters
# 84 inches to 213.36 centimeters
# this is the range that that covers about 99.9999% of the adult population
```



Problematic patterns:

1. A pattern of the form x' y or x' y' ' or x' y" with x and y representing feet and inches, respectively. Here are ten examples:

#> 5' 4" 5'7 5'7" 5'3" 5'11 5'9'' 5'10'' 5' 10 5'5" 5'2"

2. A pattern of the form x.y or x,y with x feet and y inches. Here are ten examples:

#> 5.3 5.5 6.5 5.8 5.6 5,3 5.9 6,8 5.5 6.2

3. Entries that were reported in centimeters rather than inches. Here are ten examples:

#> 150 175 177 178 163 175 178 165 165 180

- A regular expression (regex) is a way to describe specific patterns of characters of text.
- They can be used to determine if a given string matches the pattern.
- Some tutorials:
  - <https://www.regular-expressions.info/tutorial.html>
  - <https://r4ds.had.co.nz/strings.html#matching-patterns-with-regular-expressions>
  - **Cheat sheet:** <https://posit.co/wp-content/uploads/2022/10/strings-1.pdf>

### Strings are a regex

Technically any string is a regex, perhaps the simplest example is a single character. So the comma , used in the next code example is a simple example of searching with regex.

```
pattern <- ","
str_detect(c("1", "10", "100", "1,000", "10,000"), pattern)

[1] FALSE FALSE FALSE TRUE TRUE
```

Above, we noted that an entry included a cm. This is also a simple example of a regex. We can show all the entries that used cm like this:

```
str_subset(reported_heights$height, "cm")

[1] "165cm" "170 cm"
```

## Special characters

- The main feature that distinguishes the *regex language* from plain strings is that we can use special characters.
- Now let's consider a slightly more complicated example. Which of the following strings contain the pattern cm or inches?
- We start by introducing | which means *or*

```
yes <- c("180 cm", "70 inches")
```

```
no <- c("180", "70' ")
```

```
s <- c(yes, no)
```

```
str_detect(s, "cm|inches")
```

```
s[str_detect(s, "cm|inches")]
```

## Special characters

- Another special character that will be useful for identifying feet and inches values is `\d` which means any digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- The backslash is used to distinguish it from the character `d`. In R, we have to *escape* the backslash `\` so we actually have to use `\\d` to represent digits.

```
yes <- c("5", "6", "5'10", "5 feet", "4'11")
no <- c("", ".", "Five", "six")
s <- c(yes, no)
pattern <- "\\d"
str_detect(s, pattern)
s[str_detect(s, pattern)]
str_view(s, pattern)
str_view(s, pattern, match = NA)
```

## Character Classes

Character classes are used to define a series of characters that can be matched. We define character classes with square brackets `[]`. So, for example, if we want the pattern to match only if we have a 5 or a 6, we use the regex `[56]`:

```
yes <- c("5", "6", "5'10", "5 feet", "4'11")
```

```
no <- c("", ".", "Five", "six")
```

```
s <- c(yes, no)
```

```
str_view(s, "[56]", match = NA)
```

```
str_view(s, "5|6", match = NA)
```

## Character Classes

Suppose we want to match values between 4 and 7. A common way to define character classes is with ranges.

So, for example, `[0-9]` is equivalent to `\\d`. The pattern we want is therefore `[4-7]`.

```
yes <- as.character(4:7)
```

```
no <- as.character(1:3)
```

```
s <- c(yes, no)
```

```
str_detect(s, "[4-7]")
```

```
str_view(s, "[4-7]")
```

## Character Classes

- Keep in mind that characters do have an order and the digits do follow the numeric order. So `0` comes before `1` which comes before `2` and so on. For the same reason, we can define lower case letters as `[a-z]`, upper case letters as `[A-Z]`, and `[a-zA-Z]` as both.
- Notice that `\w` is equivalent to `[a-zA-Z0-9_]`.  
(`\w` stands for *word character* and it matches any letter, number, or underscore)



## Bounded quantifiers

For the inches part, we can have one or two digits.

This can be specified in regex with *quantifiers*.

This is done by following the pattern with curly brackets containing the number of times the previous entry can be repeated.

```
pattern <- "^\\d{1,2}$"  
yes <- c("1", "5", "9", "12")  
no <- c("123", "a4", "b")  
str_view(c(yes, no), pattern, match = NA)
```

## Case Study: Heights

With what we have learned, we can now construct an example for the pattern  $x' y''$  with  $x$  feet and  $y$  inches.

```
pattern <- "^[4-7]'\d{1,2}\"$"
yes <- c("5'7\"", "6'2\"", "5'12\"")
no <- c("6,2\"", "6.2\"", "I am 5'11\"", "3'2\"", "64")
str_detect(yes, pattern)
str_detect(no, pattern)
```

The pattern is now getting complex, but you can look at it carefully and break it down:

- $\wedge$  = start of the string
- $[4-7]$  = one digit, either 4,5,6 or 7
- $'$  = feet symbol
- $\d{1,2}$  = one or two digits
- $''$  = inches symbol
- $\$$  = end of the string

## White Space

Another problem we have is spaces. For example, our pattern does not match `5 ' 4'` because there is a space between `'` and `4` which our pattern does not permit.

```
identical("Hi", "Hi ")
```

In regex, `\s` represents white space. To find patterns like `5 ' 4`, we can change our pattern to:

```
pattern_2 <- "^[4-7]'\s\d{1,2}'$"
```

```
str_subset(problems, pattern_2)
```

```
[
```

However, this will not match the patterns with no space. So do we need more than one regex pattern? It turns out we can use a quantifier for this as well.

## Unbounded quantifiers: \*, ?, +\*\*

- We want the pattern to permit spaces but not require them. Even if there are several spaces, like in this example 5 ' 4, we still want it to match. There is a quantifier for exactly this purpose.
- In regex, the character **\*** means zero or more instances of the previous character. Here is an example:

```
yes <- c("AB", "A1B", "A11B", "A111B", "A1111B")
```

```
no <- c("A2B", "A21B")
```

```
str_detect(yes, "A1*B")
```

```
str_detect(no, "A1*B")
```

Unbounded quantifiers: `*`, `?`, `+`

There are two other similar quantifiers.

- For none or once, we can use `?`,
- for one or more, we can use `+`.

You can see how they differ with this example:

```
yes <- c("AB", "A1B", "A11B", "A111B", "A1111B")
no  <- c("A2B", "A21B")
s   <- c(yes, no)

none_or_more <- str_detect(s, "A1*B")
none_or_once <- str_detect(s, "A1?B")
once_or_more <- str_detect(s, "A1+B")
```

## Not Include

To specify patterns that we do **not** want to detect, we can use the `^` symbol but only **inside** square brackets. Remember that outside the square bracket `^` means the start of the string. So, for example, if we want to detect digits that are preceded by anything except a letter we can do the following:

```
pattern <- "[^a-zA-Z]\\d"  
yes <- c(".3", "+2", "-0", "*4")  
no <- c("A3", "B2", "C0", "E4")  
str_detect(yes, pattern)  
str_detect(no, pattern)
```

## Case Study: Heights → Search and Replace

Earlier we defined the object `problems` containing the strings that do not appear to be in inches. We can see that not too many of our problematic strings match the pattern:

```
pattern <- "^[4-7]'\d{1,2}\"$"
sum(str_detect(problems, pattern))
problems[c(2, 10, 11, 12, 15)] |> str_view(pattern, match = NA)
```

**Case Study: Heights → Search and Replace**

```
# one problem is
str_subset(problems, "inches")
str_subset(problems, "'")
# we will try to obtain format : x'y -> x feet, y inches
pattern <- "^[4-7]'\d{1,2}$"
problems |>
  str_replace("feet|ft|foot", "'") |> # replace feet, ft, foot with '
  str_replace("inches|in|'|\"", "") |> # remove all inches symbols
  str_detect(pattern) |>
  sum()
```



**Case Study: Heights → Search and Replace**

```
# another problem is the spaces: x' y"
```

```
pattern <- "[4-7]\\s*'\s*\d{1,2}$"
```

```
problems |>
```

```
  str_replace("feet|ft|foot", "'") |> # replace feet, ft, foot with '
```

```
  str_replace("inches|in|'|\\\"", "") |> # remove all inches symbols
```

```
  str_detect(pattern) |>
```

```
  sum()
```

