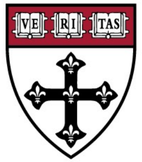
Introduction to Data Wrangling III

Summer Institute in Data Science

Rolando J. Acosta



HARVARD
SCHOOL OF PUBLIC HEALTH

June 22, 2022

@RJANunez

What to expected today

- Today we will learn about functions to process strings
- Specifically, we will learn about functions to detect, locate, extract, and replace strings
- Furthermore, we will learn the basics of regular expressions (regex)

The stringr package

- In general, string processing tasks can be divided into *detecting*, *locating*, *extracting*, or *replacing* patterns in string

The stringr package

- In general, string processing tasks can be divided into *detecting*, *locating*, *extracting*, or *replacing* patterns in string
- There are many useful functions in the ***stringr*** package and we will not have time to go over all of them

The stringr package

- In general, string processing tasks can be divided into *detecting*, *locating*, *extracting*, or *replacing* patterns in string
- There are many useful functions in the ***stringr*** package and we will not have time to go over all of them
- Hence, we first present the functions available in the ***stringr*** package

The stringr package

- In general, string processing tasks can be divided into *detecting*, *locating*, *extracting*, or *replacing* patterns in string
- There are many useful functions in the ***stringr*** package and we will not have time to go over all of them
- Hence, we first present the functions available in the ***stringr*** package
- Then, we will use some of them in the lecture and hands-on sections today

Detect functions

Function	Description
`str_detect`	Is the pattern in the string?
`str_which`	Returns the index of entries that contain the pattern
`str_subset`	Returns the subset of strings that contain the pattern

Locate functions

Function	Description
`str_locate`	Returns the position of first occurrence of the pattern in the string
`str_locate_all`	Returns the positions of all occurrences of the pattern in the string
`str_view`	Shows the first part of the string that matches the pattern
`str_view_all`	Shows all the parts of the string that matches the pattern

Extract functions

Function	Description
`str_extract`	Extract the first part of the string that matches the pattern
`str_extract_all`	Extract all the parts of the string that match the pattern
`str_match`	Extract the first part of the string that matches the groups and the patterns defined by the groups
`str_match_all`	Extract all the parts of the string that match the groups and the patterns defined by the groups
`str_sub`	Extract a substring
`str_split`	Split a string into a list with parts separated by the pattern
`str_split_fixed`	Split a string into a matrix with parts separated by the patterns

Describe functions

Function	Description
`str_count`	Count the number of times a pattern appears in a string
`str_length`	Number of characters in the string

Replace functions

Function	Description
`str_replace`	Replace first part of a string matching a pattern with another.
`str_replace_all`	Replace all parts of a string matching a pattern with another.
`str_to_upper`	Change all characters to upper case.
`str_to_lower`	Change all characters to lower case.
`str_to_title`	Change first character to upper and rest to lower.
`str_replace_na`	Replace all NAs to a new value.
`str_trim`	Remove white space from start and end of string.

Manipulate functions

Function	Description
`str_c`	Join multiple strings
`str_conv`	Change the encoding of the string
`str_sort`	Sort the vector in alphabetical order
`str_order`	Index needed to order the vector in alphabetical order
`str_trunc`	Truncate a string to a fixed size
`str_pad`	Add white space to string to make it a fixed size
`str_dup`	Repeat a string
`str_wrap`	Wrap things into formatted paragraphs
`str_interp`	String interpolation

Basics of a string

- In R, you can create strings with either double quotes or single quotes

```
string1 <- "This is a string”
[1] "This is a string”
```

Basics of a string

- In R, you can create strings with either double quotes or single quotes

```
string1 <- "This is a string”
[1] "This is a string”

string2 <- ‘This is a string’
[1] "This is a string”
```

Basics of a string

- In R, you can create strings with either double quotes or single quotes

```
string1 <- "This is a string”
[1] "This is a string”

string2 <- ‘This is a string’
[1] "This is a string”

string3 <- "Say you want to add a "quote" quote”
Error: unexpected symbol in "string2 <- "Say you want to add a “quote"
```

Basics of a string

- In R, you can create strings with either double quotes or single quotes

```
string1 <- "This is a string”
[1] "This is a string”

string2 <- ‘This is a string’
[1] "This is a string”

string3 <- "Say you want to add a "quote" quote”
Error: unexpected symbol in "string2 <- "Say you want to add a “quote"

string4 <- 'Say you want to add a "quote" quote’
[1] "Say you want to add a \"quote\" quote"
```

Basics of a string

- In R, you can create strings with either double quotes or single quotes

```
string1 <- "This is a string”
[1] "This is a string”

string2 <- ‘This is a string’
[1] "This is a string”

string3 <- "Say you want to add a "quote" quote”
Error: unexpected symbol in "string2 <- "Say you want to add a “quote"

string4 <- 'Say you want to add a "quote" quote’
[1] "Say you want to add a \"quote\" quote"
```

- You can also save multiple strings in a vector

Basics of a string

- In R, you can create strings with either double quotes or single quotes

```
string1 <- "This is a string”
[1] "This is a string”

string2 <- ‘This is a string’
[1] "This is a string”

string3 <- "Say you want to add a "quote" quote”
Error: unexpected symbol in "string2 <- "Say you want to add a “quote"

string4 <- 'Say you want to add a "quote" quote’
[1] "Say you want to add a \"quote\" quote"
```

- You can also save multiple strings in a vector

```
c("rolando", "inter", "puerto rico”)
[1] "rolando"     "inter"       "puerto rico"
```

Basics of a string

- As one last example, suppose you want to write 5’10” as a string

Basics of a string

- As one last example, suppose you want to write 5’10” as a string

```
'5'10"'
Error: unexpected numeric constant in “'5'10"
```

Basics of a string

- As one last example, suppose you want to write 5’10” as a string

```
'5'10"'
Error: unexpected numeric constant in “'5'10"

"5'10""
+. . .
```

Basics of a string

- As one last example, suppose you want to write 5’10” as a string

```
'5'10"'
Error: unexpected numeric constant in “'5'10"

"5'10""
+. . .
```

- The former yields an error because we are closing the string after 5

Basics of a string

- As one last example, suppose you want to write 5’10” as a string

```
'5'10"'
Error: unexpected numeric constant in “'5'10"

"5'10""
+. . .
```

- The former yields an error because we are closing the string after 5
- The latter closes the string after 10, and then opens a new string!

Basics of a string

- As one last example, suppose you want to write 5’10” as a string

```
'5'10"'
Error: unexpected numeric constant in “'5'10"

"5'10""
+. . .
```

- The former yields an error because we are closing the string after 5
- The latter closes the string after 10, and then opens a new string!
- In this situation we need to escape the string with the backslash \

Basics of a string

- As one last example, suppose you want to write 5’10” as a string

```
'5'10"'
Error: unexpected numeric constant in “'5'10"

"5'10""
+. . .
```

- The former yields an error because we are closing the string after 5
- The latter closes the string after 10, and then opens a new string!
- In this situation we need to escape the string with the backslash \
- Here is how you do it

```
‘5\'10"'
[1] "5'10\""
```

Regular expressions (regex)

- A *regex* is a way to describe specific patterns of characters in a general way

Regular expressions (regex)

- A *regex* is a way to describe specific patterns of characters in a general way
- For example, we can use *regex* to find patterns of text in a corpora of text documents

Regular expressions (regex)

- A *regex* is a way to describe specific patterns of characters in a general way
- For example, we can use *regex* to find patterns of text in a corpora of text documents
- Consider the following example where we want to know the strings that contain the pattern `cm` or `inches`

```
yes <- c("180 cm", "70 inches")
no  <- c("180", "70'")
s   <- c(yes, no)
s
[1] "180 cm"    "70 inches" "180"       "70'"
```

Regular expressions (regex)

- A *regex* is a way to describe specific patterns of characters in a general way
- For example, we can use *regex* to find patterns of text in a corpora of text documents
- Consider the following example where we want to know the strings that contain the pattern `cm` or `inches`

```
yes <- c("180 cm", "70 inches")
no  <- c("180", "70'")
s   <- c(yes, no)
s
[1] "180 cm"    "70 inches" "180"       "70'"

str_detect(s, "cm") | str_detect(s, “inches")
[1]  TRUE  TRUE FALSE FALSE
```

Regular expressions (regex)

- The beauty of the *regex* language is that we can use special characters inside the quotes

Regular expressions (regex)

- The beauty of the *regex* language is that we can use special characters inside the quotes
- Special characters are imbued with meaning

Regular expressions (regex)

- The beauty of the *regex* language is that we can use special characters inside the quotes
- Special characters are imbued with meaning
- For example, the special character | means “or”:

```
s
[1] "180 cm"    "70 inches" "180"       "70'''"

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

Regular expressions (regex)

- The beauty of the *regex* language is that we can use special characters inside the quotes
- Special characters are imbued with meaning
- For example, the special character `|` means “or”:

```
s
[1] "180 cm"    "70 inches" "180"       "70'''"

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

- Note that order does not matter. We can write “`cm|inches`” or “`inches|cm`”

```
s
[1] "180 cm"    "70 inches" "180"       "70'''"

str_detect(s, “inches|cm")
[1]  TRUE  TRUE FALSE FALSE
```

Regular expressions (regex)

- Another useful special character is \d which represents a single digit: 0, 1, 2, 3, 4, 5, 6, 7 , 8, 9

Regular expressions (regex)

- Another useful special character is \d which represents a single digit: 0, 1, 2, 3, 4, 5, 6, 7 , 8, 9
- The backslash is used to distinguish the special character from the character d

Regular expressions (regex)

- Another useful special character is `\d` which represents a single digit: 0, 1, 2, 3, 4, 5, 6, 7 , 8, 9
- The backslash is used to distinguish the special character from the character d
- Recall that we have the escape backslash \, hence we actually use `\\d` to represent digits

Regular expressions (regex)

- Another useful special character is \d which represents a single digit: 0, 1, 2, 3, 4, 5, 6, 7 , 8, 9
- The backslash is used to distinguish the special character from the character d
- Recall that we have the escape backslash \, hence we actually use \\d to represent digits
- Here is an example:

```
yes <- c("5", "6", "5'10", "5 feet", "4'11")
no  <- c("", ".", "Five", "six")
s   <- c(yes, no)
s
[1] "5" "6" "5'10" "5 feet" "4'11" "" "." "Five" "six"
```

Regular expressions (regex)

- Another useful special character is \d which represents a single digit: 0, 1, 2, 3, 4, 5, 6, 7 , 8, 9
- The backslash is used to distinguish the special character from the character d
- Recall that we have the escape backslash \, hence we actually use \\d to represent digits
- Here is an example:

```
yes <- c("5", "6", "5'10", "5 feet", "4'11")
no  <- c("", ".", "Five", "six")
s   <- c(yes, no)
s
[1] "5" "6" "5'10" "5 feet" "4'11" "" "." "Five" "six"

pattern <- "\\d"
str_detect(s, pattern)
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

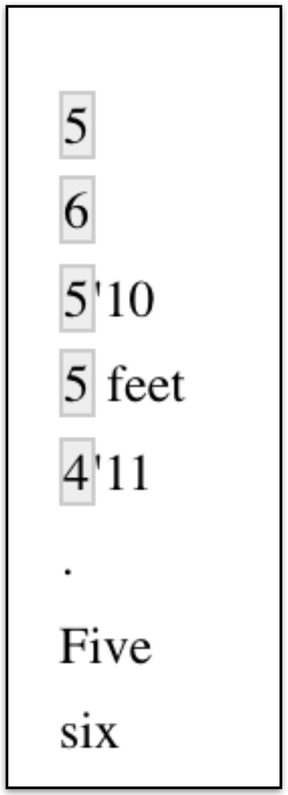
Regular expressions (regex)

- The function `str_view` and `str_view_all` allow us to see the matches

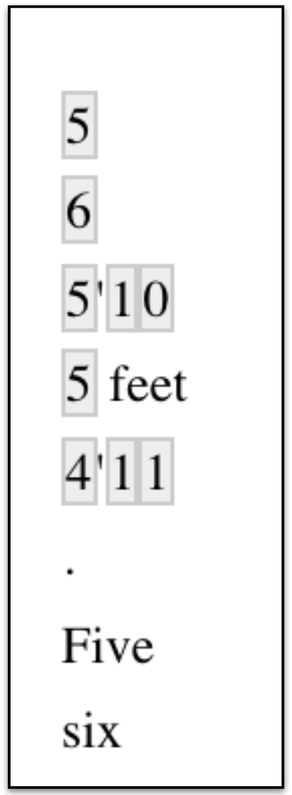
Regular expressions (regex)

- The function `str_view` and `str_view_all` allow us to see the matches
- Here is an example:

```
str_view(s, pattern)
```




```
str_view_all(s, pattern)
```



Character classes

- Character classes are used to define a series of characters that can be matched

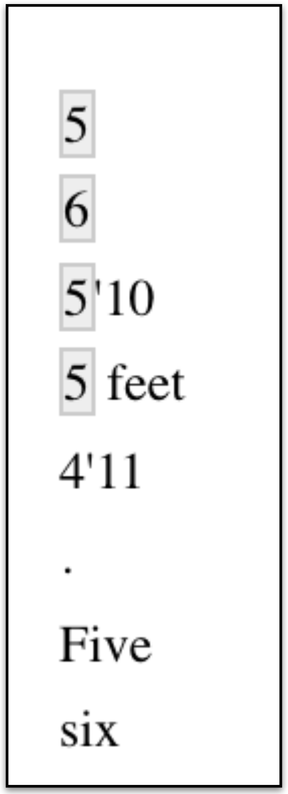
Character classes

- Character classes are used to define a series of characters that can be matched
- We define character classes with square brackets

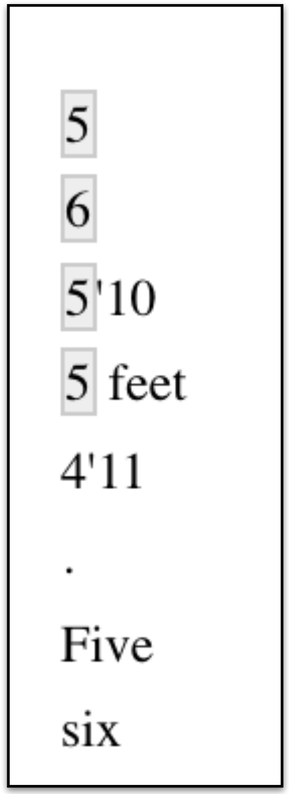
Character classes

- Character classes are used to define a series of characters that can be matched
- We define character classes with square brackets
- Here is an example where the pattern matches only if we have a 5 or a 6:

```
str_view(s, “[56]”)
```




```
str_view_all(s, “[56]”)
```



Character classes

- Suppose we want to match values between 4 and 7

Character classes

- Suppose we want to match values between `4` and `7`
- For this, we can use ranges. For example, `[0-9]` is equivalent to `\\d`

Character classes

- Suppose we want to match values between 4 and 7
- For this, we can use ranges. 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)
s
[1] "4" "5" "6" "7" "1" "2" "3"

str_detect(s, “[4-7]")
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Character classes

- Suppose we want to match values between 4 and 7
- For this, we can use ranges. 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)
s
[1] "4" "5" "6" "7" "1" "2" "3"

str_detect(s, “[4-7]")
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

- However, recall that in *regex* everything is a character
- Hence, 4 is the character “4”, not the number 4

Character classes

- Suppose we want to match values between 4 and 7
- For this, we can use ranges. 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)
s
[1] "4" "5" "6" "7" "1" "2" "3"

str_detect(s, “[4-7]")
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

- However, recall that in *regex* everything is a character
- Hence, 4 is the character “4”, not the number 4
- This is a common mistake and you have be wary of it

Anchors

- Now suppose you want to find patterns with exactly one digit

Anchors

- Now suppose you want to find patterns with exactly one digit
- For this, we can use *anchors*, which let us define patterns that start and end at a specific place

Anchors

- Now suppose you want to find patterns with exactly one digit
- For this, we can use *anchors*, which let us define patterns that start and end at a specific place
- The most commonly used anchors are ^ and $ which represent the beginning and end of a string, respectively

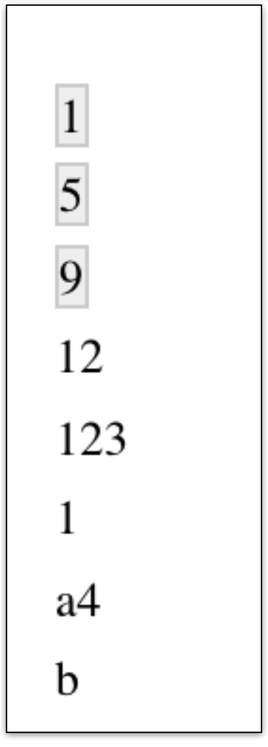
Anchors

- Now suppose you want to find patterns with exactly one digit
- For this, we can use *anchors*, which let us define patterns that start and end at a specific place
- The most commonly used anchors are ^ and $ which represent the beginning and end of a string, respectively
- Therefore, the pattern ^\\d$ reads as “start of the string followed by one digit followed by the end of the string”

Anchors

```
yes <- c("1", "5", "9")
no  <- c("12", "123", " 1", "a4", "b")
s   <- c(yes, no)
s
[1] "1"   "5"   "9"   "12"  "123" " 1"  "a4"  "b"

pattern <- "^\\d$"
str_view_all(s, pattern)
```



Quantifiers

- *Quantifiers* allow us to, you guessed it, quantify the number of characters we want to use for matching

Quantifiers

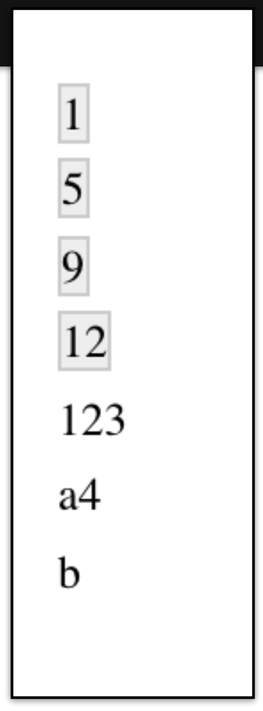
- *Quantifiers* allow us to, you guessed it, quantify the number of characters we want to use for matching
- To do this, we follow the pattern with curly brackets, `{}`, containing the number of times the previous entry can be repeated

Quantifiers

- Here is an example:

```
yes <- c("1", "5", "9", "12")
no  <- c("123", " 1", "a4", "b")
s   <- c(yes, no)
s
[1] "1"   "5"   "9"   "12"  "123" "a4"  "b"

pattern <- "^\\d{1,2}$"
str_view_all(s, pattern)
```



Other special characters

- `\\s`: white space

Other special characters

- `\\s`: white space
- `*`: zero or more instances of the previous character

Other special characters

- `\\s`: white space
- `*`: zero or more instances of the previous character
- `?`: zero or one instance of the previous character

Other special characters

- `\\s`: white space
- `*`: zero or more instances of the previous character
- `?`: zero or one instance of the previous character
- `+`: one more instances of the previous character

Other special characters

- `\\s`: white space
- `*`: zero or more instances of the previous character
- `?`: zero or one instance of the previous character
- `+`: one more instances of the previous character
- `[^]`: this specifies patterns we do **NOT** want to detect

Groups

- *Groups* are yet another powerful aspect of *regex* that allow us to extract values

Groups

- *Groups* are yet another powerful aspect of *regex* that allow us to extract values
- These groups are defined by the addition of parenthesis, `()`, to the *regex*

Groups

- *Groups* are yet another powerful aspect of *regex* that allow us to extract values
- These groups are defined by the addition of parenthesis, (), to the *regex*
- Suppose that you have height data that was self reported by a group of people, and that some of them incorrectly typed, for example, 5,6 instead of 5'6

Groups

- *Groups* are yet another powerful aspect of *regex* that allow us to extract values
- These groups are defined by the addition of parenthesis, (), to the *regex*
- Suppose that you have height data that was self reported by a group of people, and that some of them incorrectly typed, for example, 5,6 instead of 5'6
- You want to fix that

Groups

- *Groups* are yet another powerful aspect of *regex* that allow us to extract values
- These groups are defined by the addition of parenthesis, (), to the *regex*
- Suppose that you have height data that was self reported by a group of people, and that some of them incorrectly typed, for example, 5,6 instead of 5’6
- You want to fix that
- Here is an example

```
yes <- c(“5,9”, “5,11”, “6,", “6,1")
no  <- c(“5'9", " ,”, “2,8", “6.1.1”)
s   <- c(yes, no)
s
[1] "5,9"   "5,11"  "6,"    "6,1"   "5'9"   ","     "2,8"   “6.1.1"

pattern_without_groups <- “^[4-7],\\d*$”
pattern_with_groups    <- “^([4-7]),(\\d*)$”
```

Groups

- *Groups* are yet another powerful aspect of *regex* that allow us to extract values
- These groups are defined by the addition of parenthesis, (), to the *regex*
- Suppose that you have height data that was self reported by a group of people, and that some of them incorrectly typed, for example, 5,6 instead of 5’6
- You want to fix that
- Here is an example

```
yes <- c(“5,9”, “5,11”, “6,", “6,1")
no  <- c(“5'9", " ,”, “2,8", “6.1.1”)
s   <- c(yes, no)
s
[1] "5,9"   "5,11"  "6,"    "6,1"   "5'9"   ","     "2,8"   “6.1.1"

pattern_without_groups <- “^[4-7],\\d*$”
pattern_with_groups    <- “^([4-7]),(\\d*)$”
```

- Who wants to guess what these two patterns do?

Groups

```
s
[1] "5,9"   "5,11"  "6,"    "6,1"   "5'9"   ","     "2,8"   “6.1.1"

str_dectect(s, pattern_without_groups)
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE

str_dectect(s, pattern_with_groups)
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Groups

```
s
[1] "5,9"   "5,11"  "6,"    "6,1"   "5'9"   ","     "2,8"   “6.1.1"

str_dectect(s, pattern_without_groups)
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE

str_dectect(s, pattern_with_groups)
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

- Both patterns detect the same thing!

Groups

```
s
[1] "5,9"   "5,11"  "6,"    "6,1"   "5'9"   ","     "2,8"   “6.1.1"

str_dectect(s, pattern_without_groups)
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE

str_dectect(s, pattern_with_groups)
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

- Both patterns detect the same thing!
- However, when we use groups, we can use the str_match to extract the values defined by these groups

Groups

```
s
[1] "5,9"   "5,11"  "6,"    "6,1"   "5'9"   ","     "2,8"   “6.1.1"

str_match(s, pattern_with_groups)
```

	V1	V2	V3
1	5,9	5	9
2	5,11	5	11
3	6,	6	
4	6,1	6	1
5	*NA*	*NA*	*NA*
6	*NA*	*NA*	*NA*
7	*NA*	*NA*	*NA*
8	*NA*	*NA*	*NA*

References

1. Introduction to Data Science: Data analysis and prediction algorithms with R by Rafael A. Irizarry, Chapter 24. https://rafalab.github.io/dsbook/
2. R for Data Science by Grolemund & Wickham, Chapter 14. https://r4ds.had.co.nz/index.html

Referencias en español:

1. Introducción a la Ciencia de Datos: Análisis de datos y algoritmos de predicción con R por Rafael A. Irizarry, Capítulo 24. https://rafalab.github.io/dslibro/
2. R para Ciencia de Datos por Grolemund & Wickham, Capítulo 14. https://es.r4ds.hadley.nz

Your turn!

Click here for the class website