Strings and Factors

Strings

- “character” data type
- Values within either single or double quotes
- Useful when you want to subset, extract, or parse character strings
- Based on matching a pattern defined using “regular expression” syntax

stringr helps standardize string manipulations

What is a string?

- Characters within single or double quotes (double preferred)
- Special characters require a leading backslash
    - Single or double quotes
    - Backslash
    - New Line
    - Tab
    - Unicode
- R will list all special characters with “?Quote”

What is a string?

- Characters within single or double quotes (double preferred)
- Special characters require a leading backslash
  - Single or double quotes
  - Backslash
  - New Line
  - Tab
  - Unicode
- R will list all special characters with “?Quote”

Not a string?

A) “I am a string” B) ‘23by51’ C) I_am_a_string

Strings in Statistics and Modeling of ID

When might you encounter strings?

Strings in Statistics and Modeling of ID

When might you encounter strings?

- Line level data
  - name, care notes, compound columns (27M, 35F)
- Compartment/class names output as part of a simulation
- Loading many data files with similar names
- …

Regular Expressions, regex, regexp

- Developed in the 1950s, and for most computer languages
- Syntax used to describe a pattern that will be matched in a string
- Each character in an expression is either:
    - regular - literal meaning of the character
    - Metacharacter - special meaning dictated by language specific regex syntax
        - Metacharacters: `.^$\|*+?{}[]()`
        - Using a metacharacter as a regular character requires “escaping” the character

Cheatsheet: https://github.com/rstudio/cheatsheets/blob/main/strings.pdf

Regular Expressions, regex, regexp

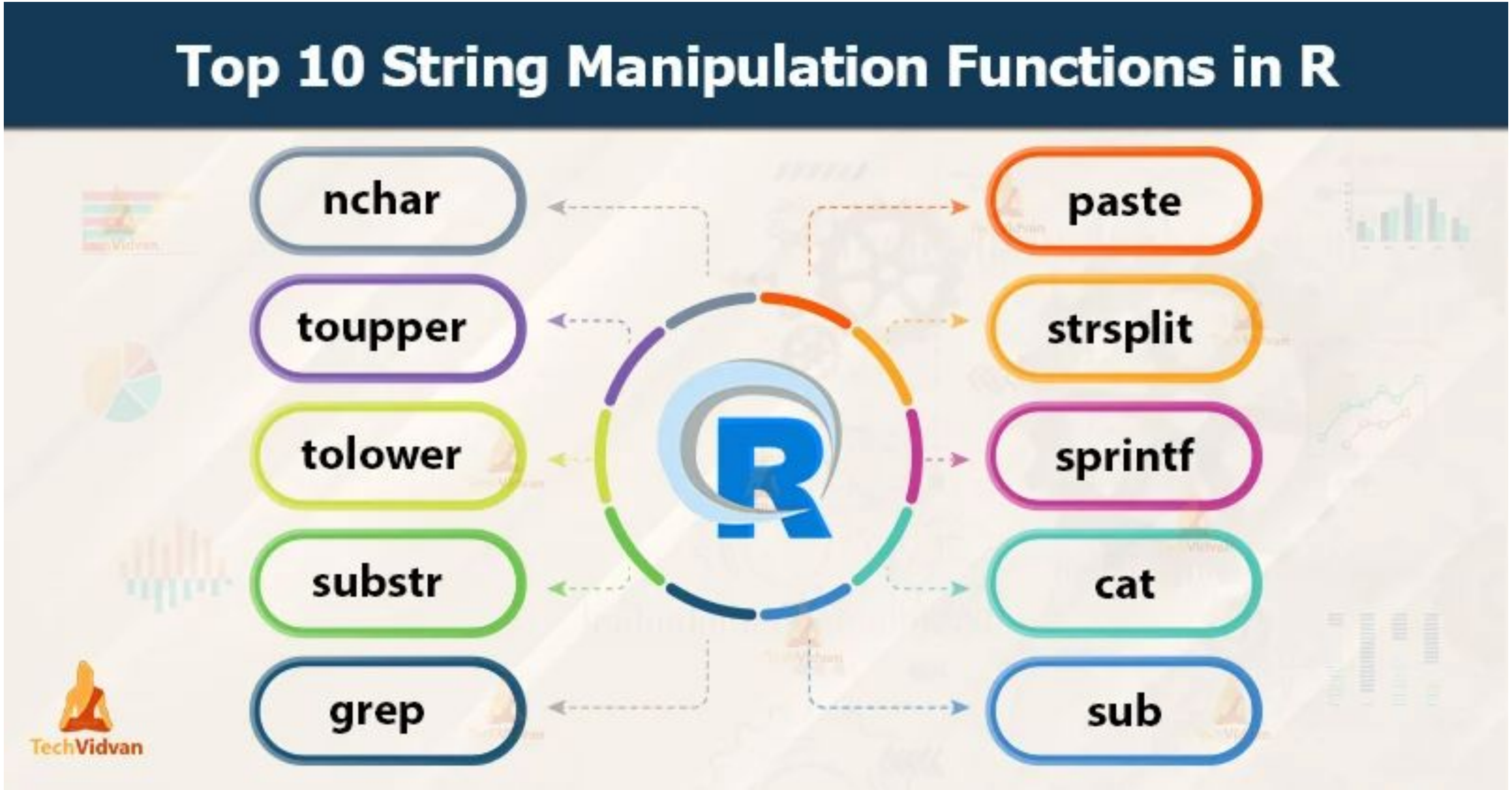
- Developed in the 1950s, and for most computer languages
- Syntax used to describe a pattern that will be matched in a string
- Each character in an expression is either:
  - regular - literal meaning of the character
  - Metacharacter - special meaning dictated by language specific regex syntax
    - Metacharacters: `.^$\|*+?{}[]()`
    - Using a metacharacter as a regular character requires “escaping” the character

Examples:

- “Summari[zs]e” would match both “Summarize” and “Summarise”
- “.a.” would match any string with an “a” contained within the string, it would match “banana” but not “apple”

Cheatsheet: https://github.com/rstudio/cheatsheets/blob/main/strings.pdf

Common Base R functions



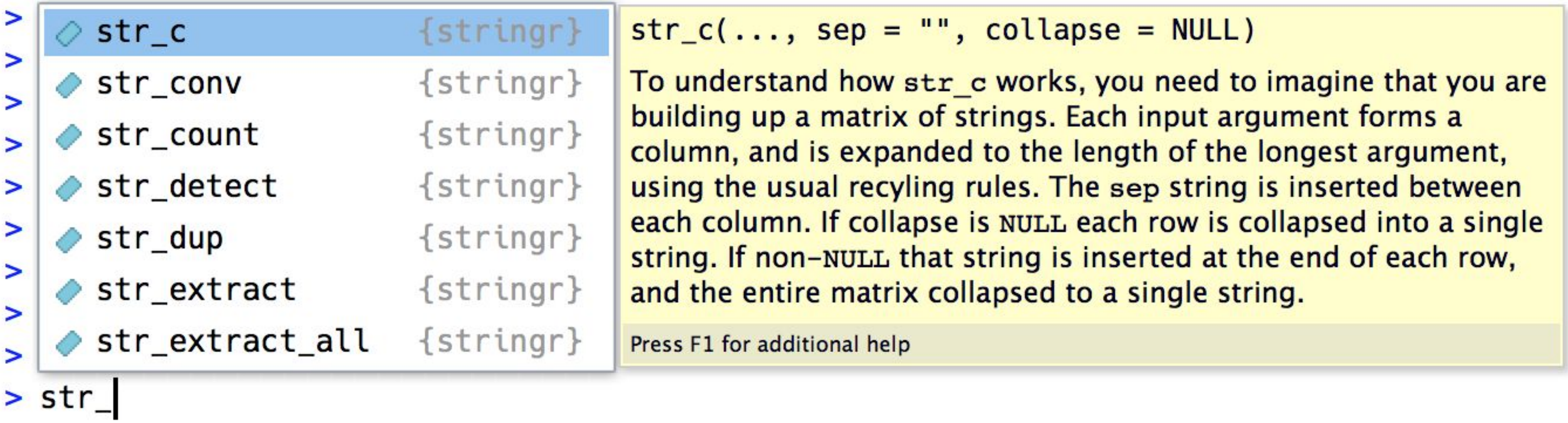
Top 10 String Manipulation Functions in R

- nchar
- toupper
- tolower
- substr
- grep
- paste
- strsplit
- sprintf
- cat
- sub

From: https://techvidvan.com/tutorials/r-string-manipulation/

Systematic Names for String Manipulation Functions

All stringr functions start with “str_”



```
str_c(..., sep = "", collapse = NULL)
```

To understand how `str_c` works, you need to imagine that you are building up a matrix of strings. Each input argument forms a column, and is expanded to the length of the longest argument, using the usual recyling rules. The `sep` string is inserted between each column. If collapse is `NULL` each row is collapsed into a single string. If non-`NULL` that string is inserted at the end of each row, and the entire matrix collapsed to a single string.

```
> str_
```

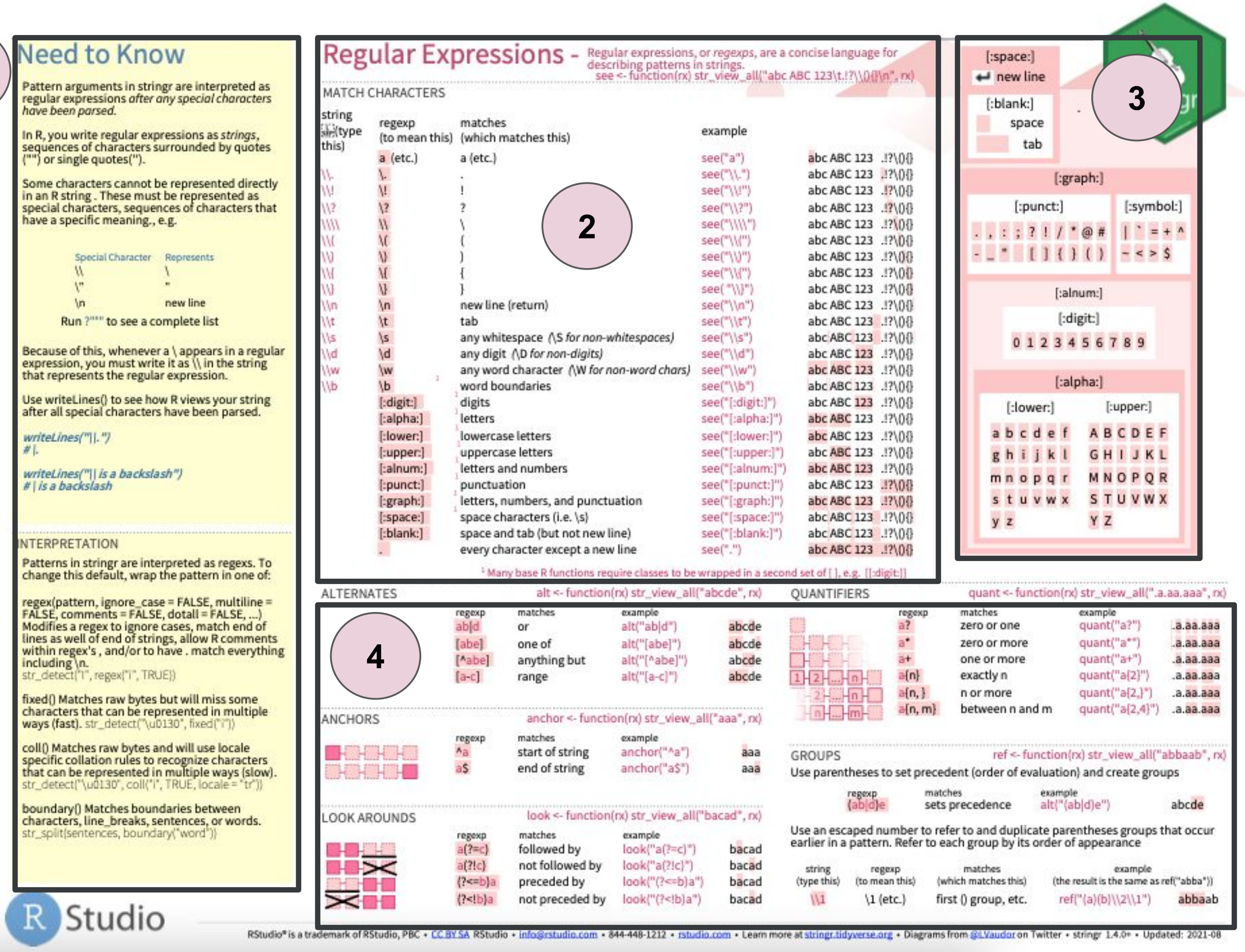
From: R for Data Science (2e) https://r4ds.hadley.nz/strings.html

Give it a try

Regex for strings with:

1. containing “A”
2. containing any number
3. ending with any uppercase letter
4. contains a question mark
5. any two repeated letters

Open Exercise.Rmd to check responses using str_view()



Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed.*

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes ("") or single quotes('').

Some characters cannot be represented directly in an R string . These must be represented as special characters, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\	\
\"	"
\n	new line

Run ?"'" to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

Use writeLines() to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
# \.

writeLines("\\ is a backslash")
# \ is a backslash
```

INTERPRETATION

Patterns in stringr are interpreted as regexs. To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...) Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's , and/or to have . match everything including \n.
str_detect("I", regex("i", TRUE))

fixed() Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). str_detect("\u0130", fixed("i"))

coll() Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow). str_detect("\u0130", coll("i", TRUE, locale = "tr"))

boundary() Matches boundaries between characters, line_breaks, sentences, or words. str_split(sentences, boundary("word"))

Regular Expressions - Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", rx)

MATCH CHARACTERS

string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	\.	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	\!	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	\?	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	\\	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	\(	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	\)	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	\{	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	\}	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	\n	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	\t	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	\s	any whitespace (\S *for non-whitespaces)*	see("\\s")	abc ABC 123 .!?\(){}
\\d	\d	any digit (\D *for non-digits)*	see("\\d")	abc ABC 123 .!?\(){}
\\w	\w	any word character (\W *for non-word chars)*	see("\\w")	abc ABC 123 .!?\(){}
\\b	\b	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:][1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:][1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:][1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:][1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:][1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:][1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:][1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:][1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

[1] Many base R functions require classes to be wrapped in a second set of [ ], e.g. [[:digit:]]

[:space:]
↵ new line

[:blank:]
space
tab

[:graph:]

[:punct:]
. , : ; ? ! / * @ #
- _ " [ ] { } ( )

[:symbol:]
| ` = + ^
~ < > $

[:alnum:]

[:digit:]
0 1 2 3 4 5 6 7 8 9

[:alpha:]

[:lower:]
a b c d e f
g h i j k l
m n o p q r
s t u v w x
y z

[:upper:]
A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z

ALTERNATES

alt <- function(rx) str_view_all("abcde", rx)

regexp	matches	example	
ab|d	or	alt("ab|d")	abcde
[abe]	one of	alt("[abe]")	abcde
[^abe]	anything but	alt("[^abe]")	abcde
[a-c]	range	alt("[a-c]")	abcde

ANCHORS

anchor <- function(rx) str_view_all("aaa", rx)

regexp	matches	example	
^a	start of string	anchor("^a")	aaa
a$	end of string	anchor("a$")	aaa

LOOK AROUNDS

look <- function(rx) str_view_all("bacad", rx)

regexp	matches	example	
a(?=c)	followed by	look("a(?=c)")	bacad
a(?!c)	not followed by	look("a(?!c)")	bacad
(?<=b)a	preceded by	look("(?<=b)a")	bacad
(?<!b)a	not preceded by	look("(?<!b)a")	bacad

QUANTIFIERS

quant <- function(rx) str_view_all(".a.aa.aaa", rx)

regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly n	quant("a{2}")	.a.aa.aaa
a{n, }	n or more	quant("a{2,}")	.a.aa.aaa
a{n, m}	between n and m	quant("a{2,4}")	.a.aa.aaa

GROUPS

ref <- function(rx) str_view_all("abbaab", rx)

Use parentheses to set precedent (order of evaluation) and create groups

regexp	matches	example	
(ab|d)e	sets precedence	alt("(ab|d)e")	abcde

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string (type this)	regexp (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))	
\\1	\1 (etc.)	first () group, etc.	ref("(a)(b)\\2\\1")	abbaab

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at stringr.tidyverse.org • Diagrams from @LVaudor on Twitter • stringr 1.4.0+ • Updated: 2021-08

Give it a try

Pattern	Regex
containing “A”	A
containing any number	[:digit:]
ending with any uppercase letter	[:upper:]$
contains a question mark	[?]
any two repeated letters	([:alpha:][:alpha:])\\1

Questions and switch to Rmd

Factors - a type of string

- are categorical variables, variables that have a fixed and known set of possible values
- most often interact with factors when:
  - re-ordering
  - data visualizations
  - creating sub-groups within factors

Examples:

c(“Jan”,”Feb”, “Mar”, “April”)

c(“1”,”2”,”3”)

c(“low”, “medium”, “high”)

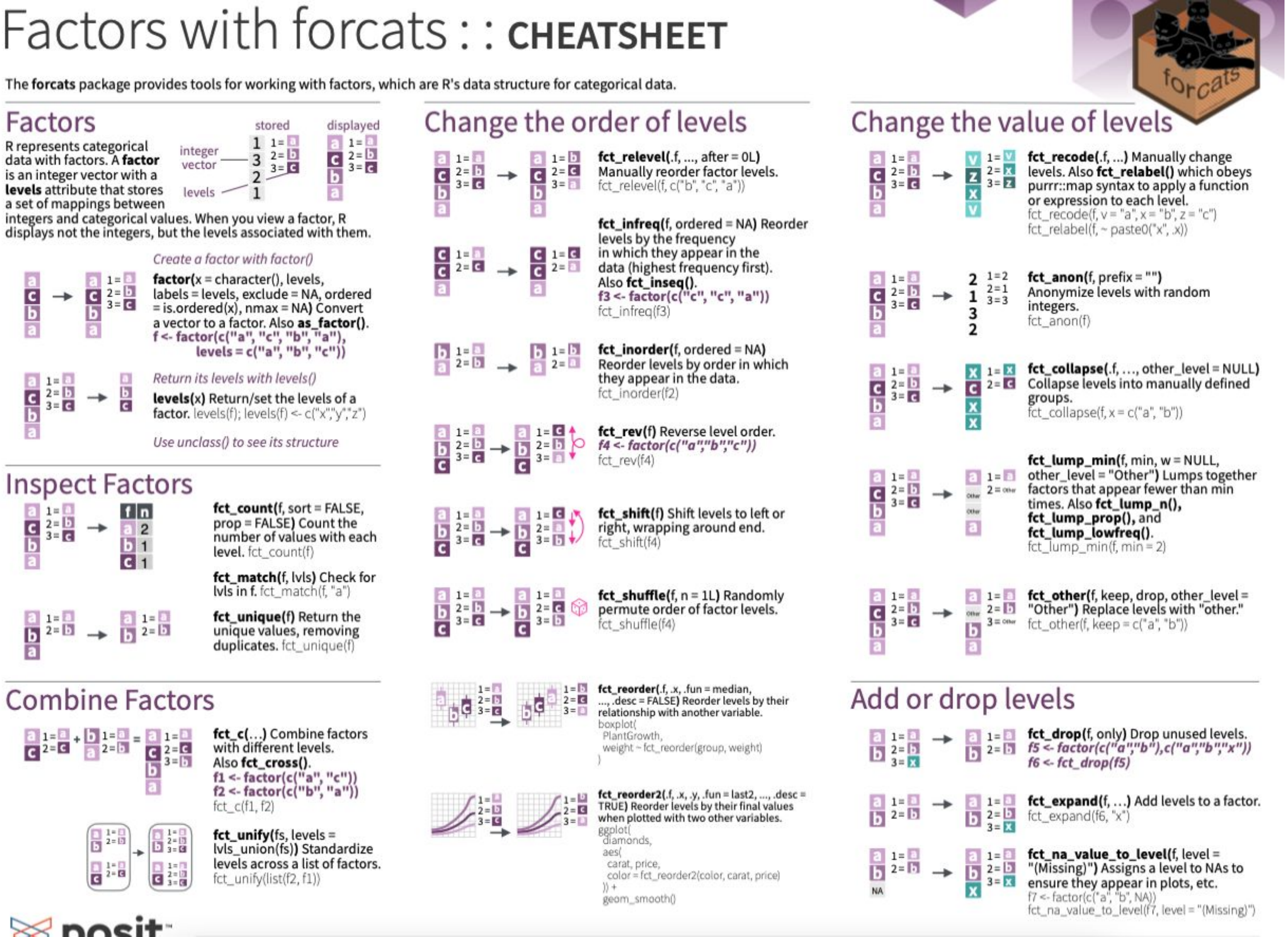
Ways to interact with factors

Manipulation	**Change order**	**Change values**	**Add/Drop levels**	**Combine factors**
Original	1. ‘Feb’ 2. ‘Jan’ 3. ‘Mar’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. ‘Jan’ 2. ‘Feb’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’
Modified	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. ‘January’ 2. ‘February’ 3. ‘March’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. Winter (‘Jan’, ‘Feb’) 2. Spring (‘Mar’)

Ways to interact with factors

Manipulation	**Change order**	**Change values**	**Add/Drop levels**	**Combine factors**
Original	1. ‘Feb’ 2. ‘Jan’ 3. ‘Mar’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. ‘Jan’ 2. ‘Feb’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’
Modified	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. ‘January’ 2. ‘February’ 3. ‘March’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. Winter (‘Jan’, ‘Feb’) 2. Spring (‘Mar’)
Forcats function				

Use the cheat sheet to find the appropriate function



Factors with forcats : : CHEATSHEET

The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors

stored
displayed
integer vector
levels

R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the levels associated with them.

Create a factor with factor()

factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA) Convert a vector to a factor. Also **as_factor()**.
f <- factor(c("a", "c", "b", "a"), levels = c("a", "b", "c"))

Return its levels with levels()

levels(x) Return/set the levels of a factor. levels(f); levels(f) <- c("x","y","z")

Use unclass() to see its structure

Inspect Factors

fct_count(f, sort = FALSE, prop = FALSE) Count the number of values with each level. fct_count(f)

fct_match(f, lvls) Check for lvls in f. fct_match(f, "a")

fct_unique(f) Return the unique values, removing duplicates. fct_unique(f)

Combine Factors

fct_c(...) Combine factors with different levels. Also **fct_cross()**.
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
fct_c(f1, f2)

fct_unify(fs, levels = lvls_union(fs)) Standardize levels across a list of factors.
fct_unify(list(f2, f1))

Change the order of levels

fct_relevel(.f, ..., after = 0L) Manually reorder factor levels.
fct_relevel(f, c("b", "c", "a"))

fct_infreq(f, ordered = NA) Reorder levels by the frequency in which they appear in the data (highest frequency first). Also **fct_inseq()**.
f3 <- factor(c("c", "c", "a"))
fct_infreq(f3)

fct_inorder(f, ordered = NA) Reorder levels by order in which they appear in the data.
fct_inorder(f2)

fct_rev(f) Reverse level order.
f4 <- factor(c("a","b","c"))
fct_rev(f4)

fct_shift(f) Shift levels to left or right, wrapping around end.
fct_shift(f4)

fct_shuffle(f, n = 1L) Randomly permute order of factor levels.
fct_shuffle(f4)

fct_reorder(.f, .x, .fun = median, ..., .desc = FALSE) Reorder levels by their relationship with another variable.

```
boxplot(
 PlantGrowth,
 weight ~ fct_reorder(group, weight)
)
```

fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE) Reorder levels by their final values when plotted with two other variables.

```
ggplot(
 diamonds,
 aes(
  carat, price,
  color = fct_reorder2(color, carat, price)
 )) +
 geom_smooth()
```

Change the value of levels

fct_recode(.f, ...) Manually change levels. Also **fct_relabel()** which obeys purrr::map syntax to apply a function or expression to each level.
fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))

fct_anon(f, prefix = "") Anonymize levels with random integers.
fct_anon(f)

fct_collapse(.f, ..., other_level = NULL) Collapse levels into manually defined groups.
fct_collapse(f, x = c("a", "b"))

fct_lump_min(f, min, w = NULL, other_level = "Other") Lumps together factors that appear fewer than min times. Also **fct_lump_n()**, **fct_lump_prop()**, and **fct_lump_lowfreq()**.
fct_lump_min(f, min = 2)

fct_other(f, keep, drop, other_level = "Other") Replace levels with "other."
fct_other(f, keep = c("a", "b"))

Add or drop levels

fct_drop(f, only) Drop unused levels.
f5 <- factor(c("a","b"),c("a","b","x"))
f6 <- fct_drop(f5)

fct_expand(f, ...) Add levels to a factor.
fct_expand(f6, "x")

fct_na_value_to_level(f, level = "(Missing)") Assigns a level to NAs to ensure they appear in plots, etc.
f7 <- factor(c("a", "b", NA))
fct_na_value_to_level(f7, level = "(Missing)")

posit

Ways to interact with factors

Manipulation	Change order	Change values	Add/Drop levels	Combine factors
Original	1. ‘Feb’ 2. ‘Jan’ 3. ‘Mar’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. ‘Jan’ 2. ‘Feb’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’
Modified	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. ‘January’ 2. ‘February’ 3. ‘March’	1. ‘Jan’ 2. ‘Feb’ 3. ‘Mar’	1. Winter (‘Jan’, ‘Feb’) 2. Spring (‘Mar’)
Forcats function	fct_relevel()	fct_recode()	fct_drop()	fct_collapse()

Questions and switch back to Rmd

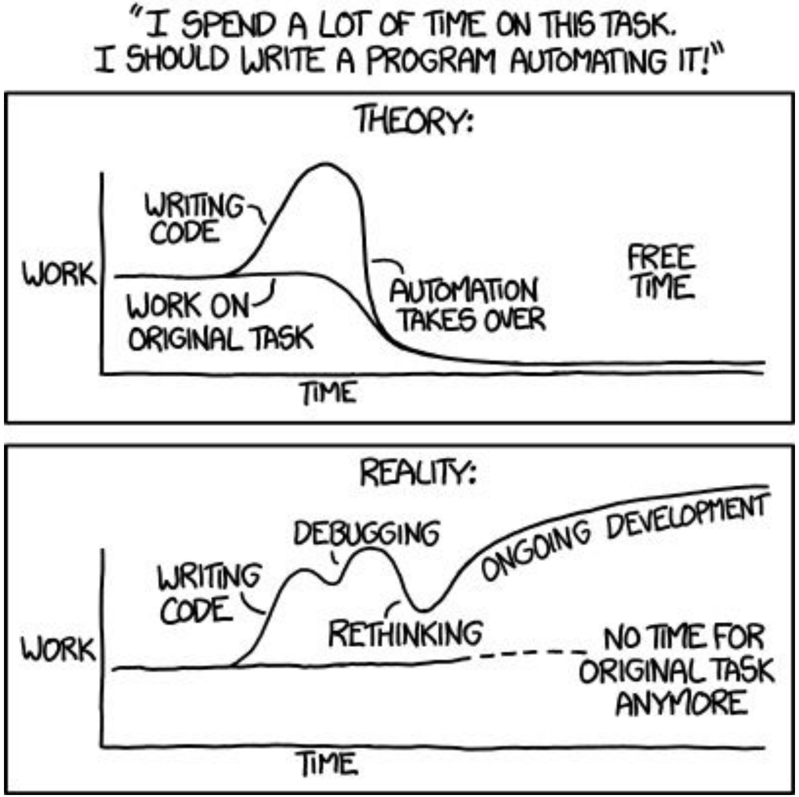
Functions: A way to automate a process

Writing a function can be a time saver or sink

Knowing what ***could*** be a function vs what ***should*** be a function comes with experience

You're always going to be wrong at some point

What are some potentially good candidate tasks?



HTTPS://XKCD.COM/1319/

Functions: A way to automate a process

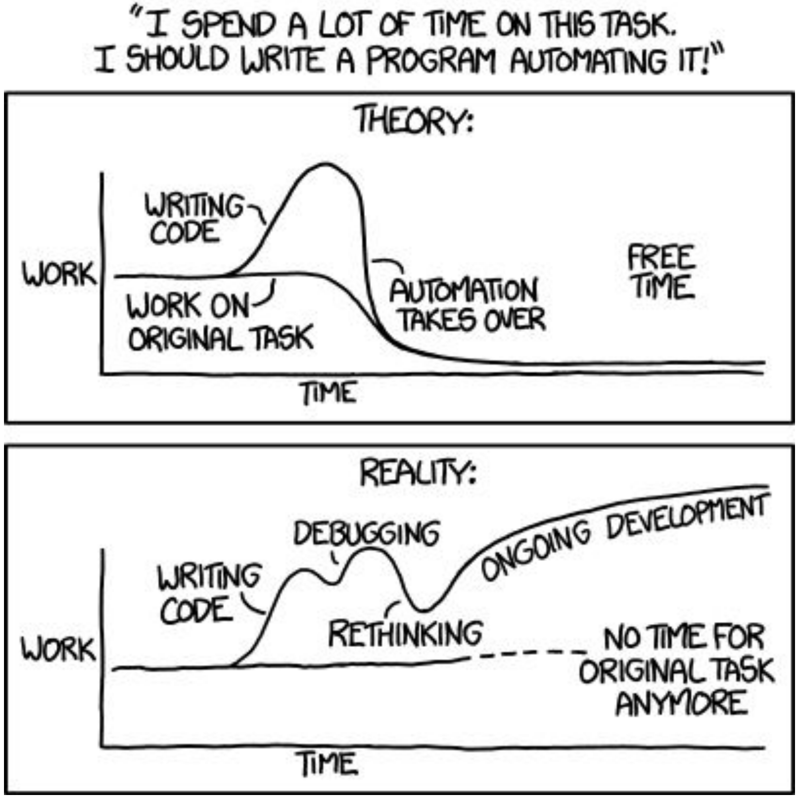
Writing a function can be a time saver or sink

Knowing what ***could*** be a function vs what ***should*** be a function comes with experience

You're always going to be wrong at some point

What are some potentially good candidate tasks?

- Processing standardized data
- ODEs/Dynamic Models
- Common data visualizations

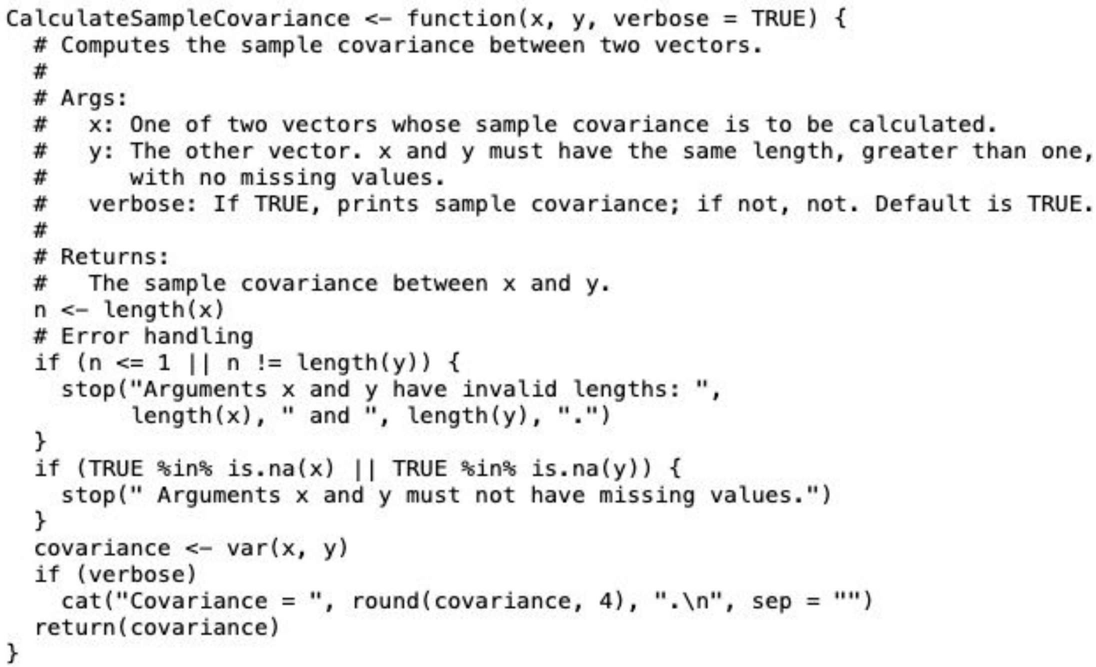


HTTPS://XKCD.COM/1319/

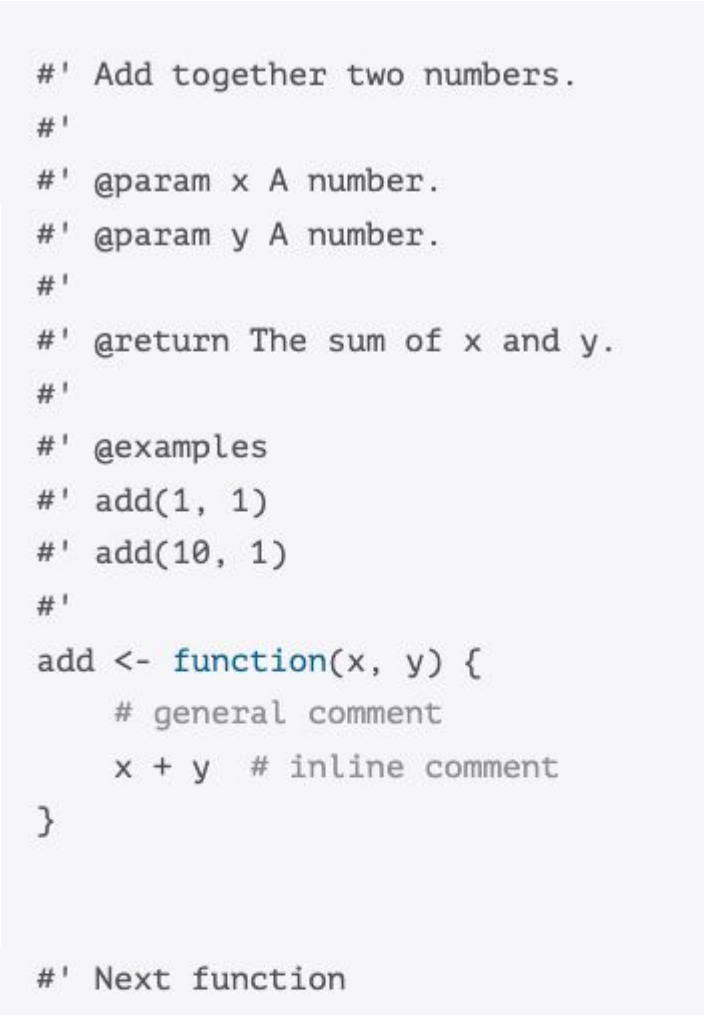
Anatomy of a function

```
FunctionName <- function(arguement1,
arguement2, ...){
  #some analysis
  return(outputOfFunction)
}
```

Document with Comments!



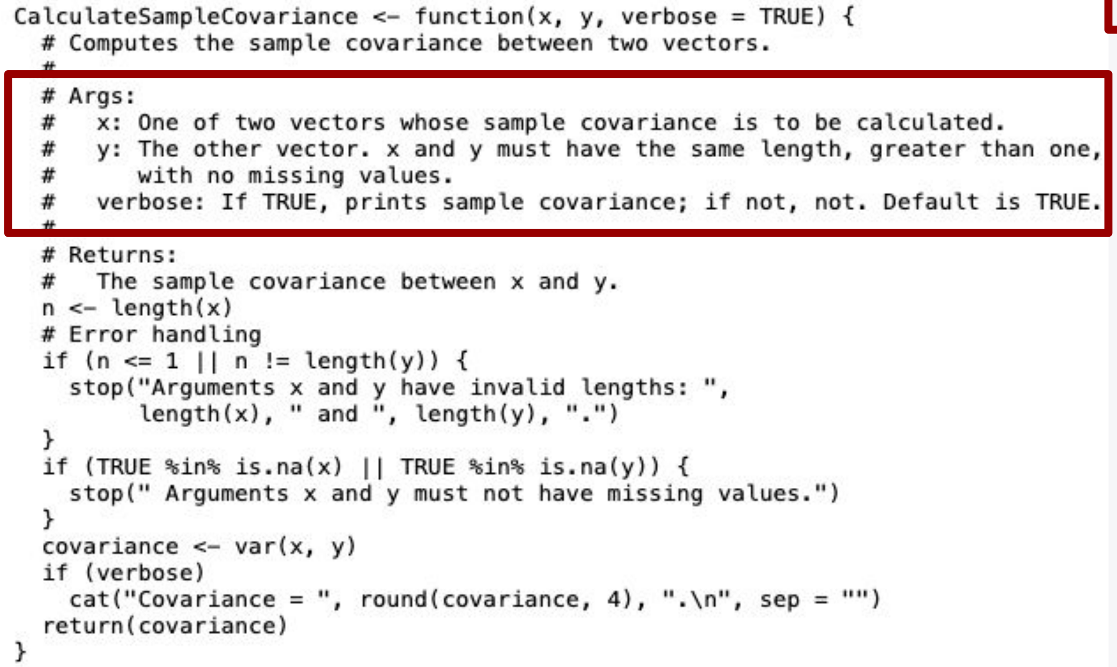
```
CalculateSampleCovariance <- function(x, y, verbose = TRUE) {
  # Computes the sample covariance between two vectors.
  #
  # Args:
  #   x: One of two vectors whose sample covariance is to be calculated.
  #   y: The other vector. x and y must have the same length, greater than one,
  #      with no missing values.
  #   verbose: If TRUE, prints sample covariance; if not, not. Default is TRUE.
  #
  # Returns:
  #   The sample covariance between x and y.
  n <- length(x)
  # Error handling
  if (n <= 1 || n != length(y)) {
    stop("Arguments x and y have invalid lengths: ",
         length(x), " and ", length(y), ".")
  }
  if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) {
    stop(" Arguments x and y must not have missing values.")
  }
  covariance <- var(x, y)
  if (verbose)
    cat("Covariance = ", round(covariance, 4), ".\n", sep = "")
  return(covariance)
}
```



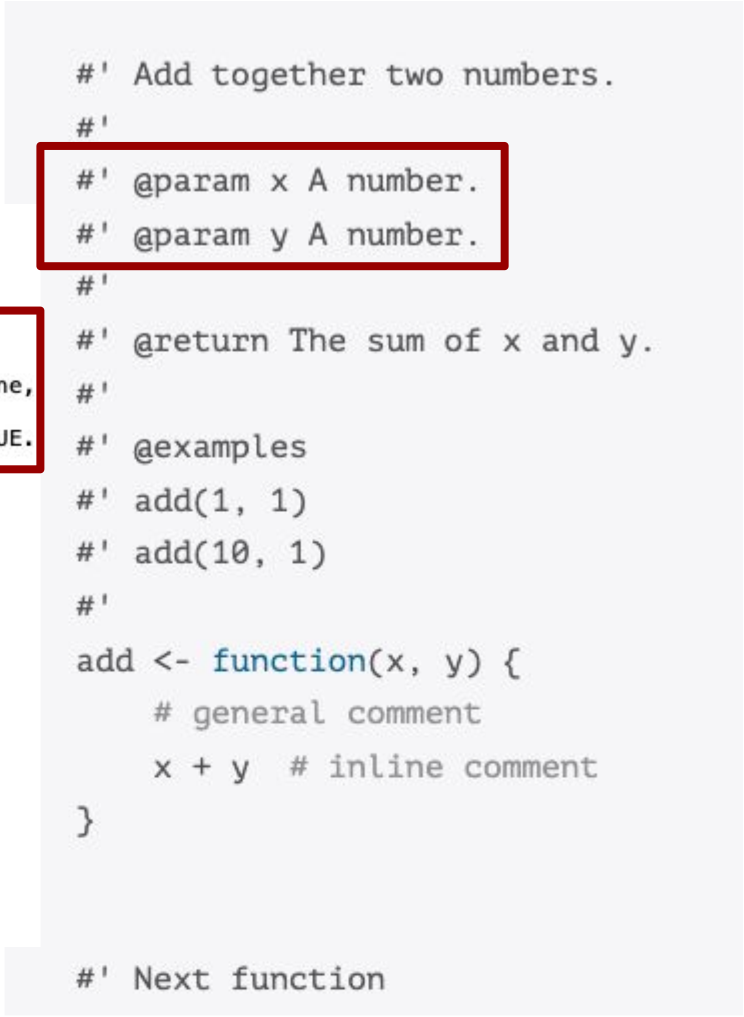
```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#'
#' @return The sum of x and y.
#'
#' @examples
#' add(1, 1)
#' add(10, 1)
#'
add <- function(x, y) {
    # general comment
    x + y  # inline comment
}


#' Next function
```

Document with Comments!



```
CalculateSampleCovariance <- function(x, y, verbose = TRUE) {
  # Computes the sample covariance between two vectors.
  #
  # Args:
  #   x: One of two vectors whose sample covariance is to be calculated.
  #   y: The other vector. x and y must have the same length, greater than one,
  #      with no missing values.
  #   verbose: If TRUE, prints sample covariance; if not, not. Default is TRUE.
  #
  # Returns:
  #   The sample covariance between x and y.
  n <- length(x)
  # Error handling
  if (n <= 1 || n != length(y)) {
    stop("Arguments x and y have invalid lengths: ",
         length(x), " and ", length(y), ".")
  }
  if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) {
    stop(" Arguments x and y must not have missing values.")
  }
  covariance <- var(x, y)
  if (verbose)
    cat("Covariance = ", round(covariance, 4), ".\n", sep = "")
  return(covariance)
}
```



```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#'
#' @return The sum of x and y.
#'
#' @examples
#' add(1, 1)
#' add(10, 1)
#'
add <- function(x, y) {
    # general comment
    x + y  # inline comment
}



#' Next function
```

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

Soft coding: values or functions are set in parameters that is reference by the code

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

Soft coding: values or functions are set in parameters that is reference by the code

```
travel.data %>%
filter((!AddressState %in% c("ALBERTA", "BC", "NA","CANADA", "IN TRANSIT TO
CANADA“)))
```

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

Soft coding: values or functions are set in parameters that is reference by the code

```
travel.data %>%
filter((!AddressState %in% c("ALBERTA", "BC", "NA","CANADA", "IN TRANSIT TO
CANADA“)))
```

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

Soft coding: values or functions are set in parameters that is reference by the code

This is an example of hard coding

```
travel.data %>%
filter((!AddressState %in% c("ALBERTA", "BC", "NA","CANADA", "IN TRANSIT TO
CANADA“)))
```

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

Soft coding: values or functions are set in parameters that is reference by the code

How would you convert to soft coding?

```
 travel.data %>%
 filter((!AddressState %in% c("ALBERTA", "BC", "NA","CANADA", "IN TRANSIT TO
CANADA“)))
```

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

Soft coding: values or functions are set in parameters that is reference by the code

Converted to soft coding

```
drop.names <- c("ALBERTA", "BC", "NA","CANADA", "IN TRANSIT TO CANADA“)

travel.data %>%
 filter((!AddressState %in% drop.names))
```

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

```
travel.data %>%
 filter((!AddressState %in% c("ALBERTA", "BC",
"NA","CANADA", "IN TRANSIT TO CANADA“)))
```

Use when:

Soft coding: values or functions are set in parameters that is reference by the code

```
drop.names <- c("ALBERTA", "BC",
"NA","CANADA", "IN TRANSIT TO CANADA“)

travel.data %>%
 filter((!AddressState %in% drop.names))
```

Use when:

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

```
travel.data %>%
filter((!AddressState %in% c("ALBERTA", "BC",
"NA","CANADA", "IN TRANSIT TO CANADA“)))
```

Use when:

Value:

- never changes
- is referenced only once
- Is short, and direct embedding makes the code easier to understand

Soft coding: values or functions are set in parameters that is reference by the code

```
drop.names <- c("ALBERTA", "BC",
"NA","CANADA", "IN TRANSIT TO CANADA“)

travel.data %>%
 filter((!AddressState %in% drop.names))
```

Use when:

Value:

- changes often
- is referenced multiple times
- is ugly, making it hard to read code

Hard vs soft coding

Hard coding: values or functions are directly embedded into the code

```
travel.data %>%
filter((!AddressState %
"NA","CANADA", "IN T
```

Soft coding: values or functions are set in parameters that is reference by the code

```
ERTA", "BC",
RANSIT TO CANADA“)

%in% drop.names))
```

Take home:

1. Soft coding is essential for flexible functions
2. Soft coding is strategy to make code easier to read and/or modify, but comes at the cost of increased nesting structure

Use when:

Value:

- never changes
- is referenced only once
- Is short, and direct embedding makes the code easier to understand

Value:

- changes often
- is referenced multiple times
- is ugly, making it hard to read code

Questions and switch back to Rmd