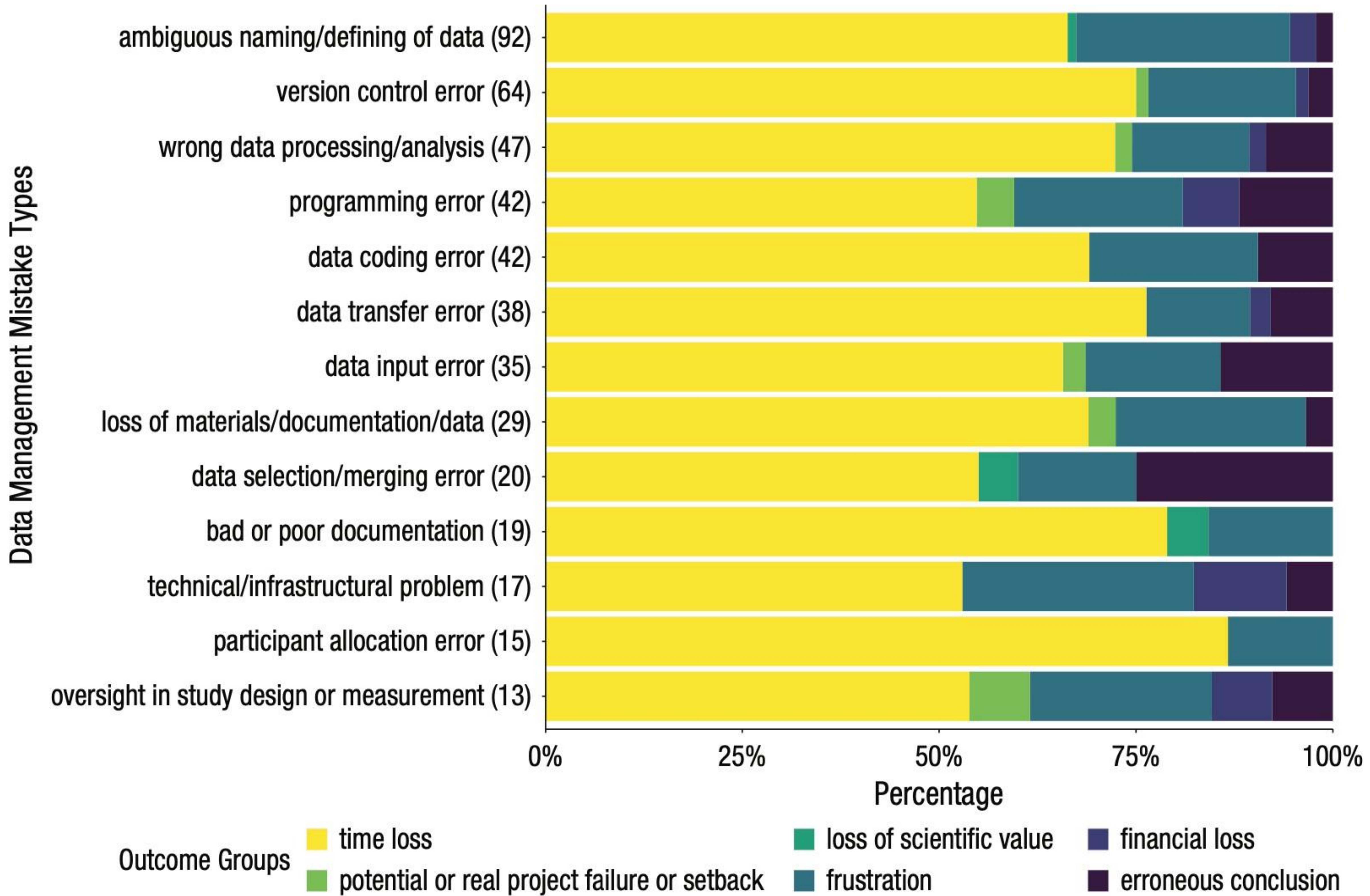
PSYC 259: Principles of Data Science

Week 3: Data Types and Transformations

Outline

1. Lecture
   - Data types
   - Factors and *forcats*
   - Data transformations with *dplyr*
2. *BREAK*
3. Tutorial (data transformations)
4. Getting started on homework

How can R data processing help us minimize mistakes?



How can R data processing help us minimize mistakes?

- Code *is* documentation
- Transform data with less human intervention
- But coding errors are also common…what can we do to make sure that our code works?
  - Instead of "hard" coding based on position (such as ds[1, 2]),with tidyverse we can filter and select by name and logical conditions
  - Use good coding practices to clean up variable names to make them human-readable
  - Reduce code duplication (copy-paste-tweak) with loops and functions
  - Write code that's reproducible (avoiding absolute file paths)

Base R vs. Tidyverse

- Can think of it like speaking two different languages (e.g., English and Spanish)
- Exact words, pronunciation, and ordering of words differ, but meaning of what you are communicating is the same
- Mostly going to teach Tidyverse language in this course
- But you can translate between them

Operation	base R example	dplyr function	dplyr example
select some rows	my_data[c(2,3,10),]	slice()	slice(my_data, c(2,3,10))
select some columns	my_data[,1:2] OR my_data[,c(“Var_1”, “Var_2”)]	select()	select(my_data, Var_1, Var_2)
subset	my_data[my_data$Var_2>80,] OR subset(my_data, Var_2>80)	filter()	filter(my_data, Var_2>80)
order the rows	my_data[order(my_data$Var_2),]	arrange()	arrange(my_data, Var_2)
add a column	my_data$logVar_2 <- log(my_data$Var_2) OR transform(my_data, logVar_2=log(Var_2)	mutate()	mutate(my_data, logVar_2 = log(Var_2)
define groups of data	Done within other functions.	group_by()	my_data %>% group_by(Var_3)

summarise the data	aggregate(Var_2 ~ Var_3, data =
	my_data, FUN = mean)
	OR
	tapply(my_data$Var_2,
	list(my_data$Var_3), mean)
		summarise()
		AND
		group_by()
			my_data %>%
			group_by(Var_3) %>%
			meanVar_2 = mean(Var_2)

R packages for data science

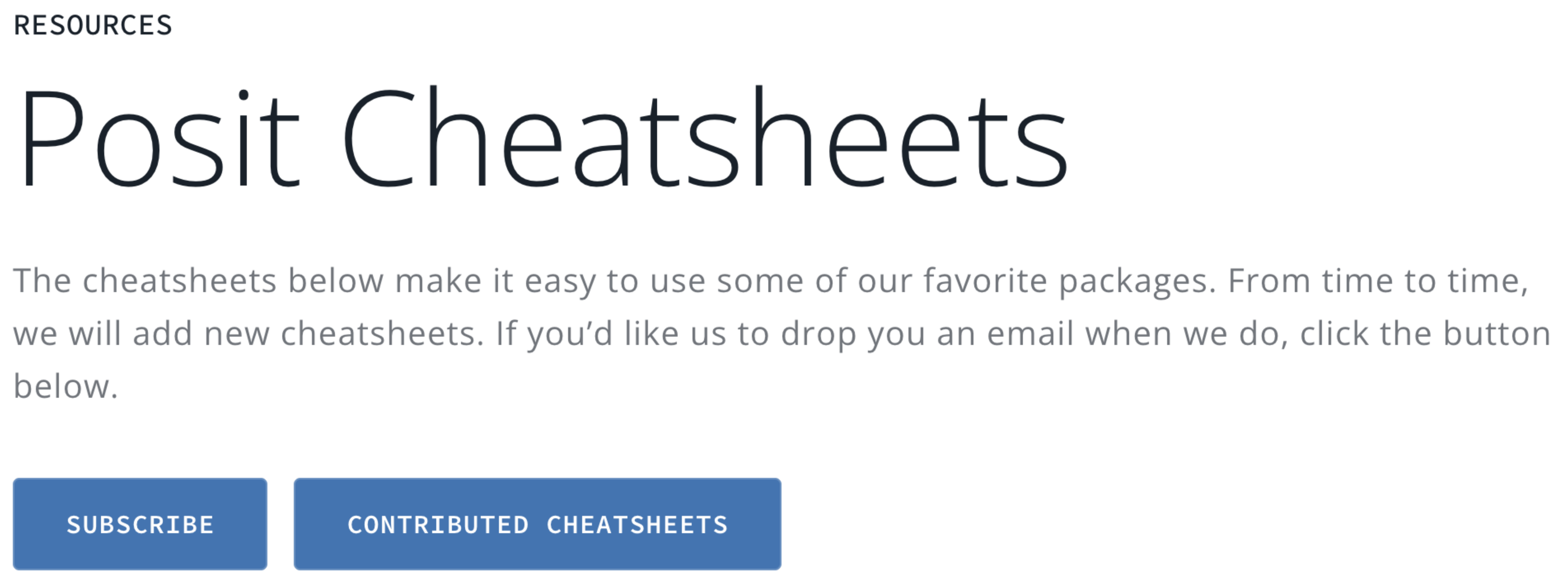
The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Cheatsheets

You can store these in R (for easy access!)


RESOURCES

Posit Cheatsheets

The cheatsheets below make it easy to use some of our favorite packages. From time to time, we will add new cheatsheets. If you'd like us to drop you an email when we do, click the button below.

https://posit.co/resources/cheatsheets/

Data types

Data types

- Why have pre-defined types?
  - Allows software to efficiently store data in memory
    - If a value is an integer (1, 2, 3, 4) storing it as an integer makes calculations easier compared to storing it as a double (2.34542480424624086)
  - Allows software to implement rules about transformations
    - Addition/subtraction for a date follows different rules compared with integers/double
    - “Less than” makes sense when comparing numbers, but not when comparing strings

Common data types in R reflect how values are stored

- Numeric
  - integer - 1, 2, 3
  - double - 1.12124, 5.235235
- Character - “hello”
- Logical - T/F (TRUE/FALSE)
- Date/time
- Factor
- Use typeof() function to check type of a value
- Can only apply certain operations to specific data types (e.g., cannot sum character data)

Logical statements in R

- Comparisons evaluate as T or F
  - 1 > 0 #TRUE
  - 1 == 1 #TRUE
  - 1 != 1 #FALSE
  - “s” == “S” #FALSE
  - 1 > 0 |0 > 1 #TRUE
  - 1 >0 & 0 > 1 #FALSE
  - !(1 == 1) #FALSE

Other helpful logical functions

- ifelse(logical, if_true, if_false)
  - x <- c(-1, 0, 1)
  - ifelse(x > 0, “positive”, “negative”)
  - returns: “positive”, “positive”, “negative”
- is.na() checks if a value is NA
  - x <- c(1, 2, NA)
  - is.na(x) returns: FALSE, FALSE, TRUE
- Any logical with NA returns NA

Checking/converting types

- as.factor, as.numeric, as.Date, as.character take a value and coerce it to that type
  - as.numeric(“1”) returns 1
- is.factor, is.numeric, is.character check if something is a particular type
  - is.numeric(1) #TRUE
  - is.character(as.numeric(“1”)) #FALSE

Factors

Factors in R represent categories

- x <- factor(x, levels = c(1,2,3), labels = c("rarely", "neutral", "frequently")
- levels restrict the possible set of values
- levels are *ordered*, which carries forward to output, modeling, graphics, etc.
- factors work as dummy codes; use as.numeric(factor) to treat as a continuous variable in models (if applicable)
- labels will display throughout R, which is lovely

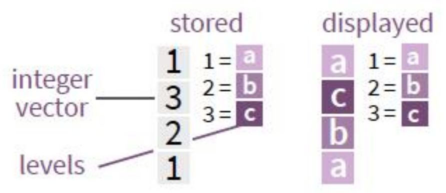
Useful *forcats* functions

- fct_count = count # of each factor level
- fct_relevel, fct_rev = reorder levels
- fct_recode, fct_collapse = reassign or combine factor levels

Factors with forcats : : CHEATSHEET

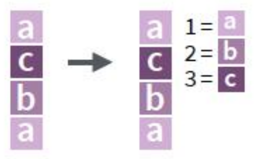
The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors



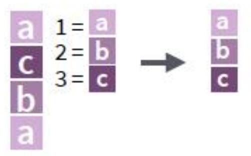
R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the levels associated with them.

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"))**

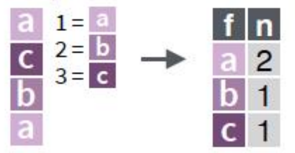
Return its levels with levels()



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

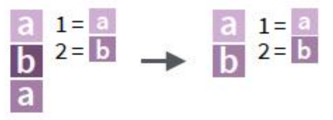
Use unclass() to see its structure

Inspect Factors



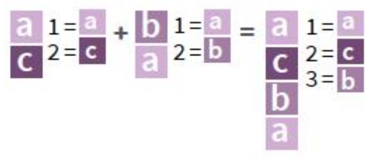
fct_count(f, sort = FALSE, prop = FALSE**)** Count the number of values with each level. fct_count(f)

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

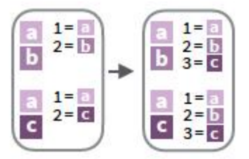


fct_unique(f**)** Return the unique values, removing duplicates. fct_unique(f)

Combine Factors

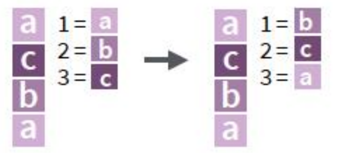


fct_c(…**)** Combine factors with different levels. Also **fct_cross()**.
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
fct_c(f1, f2)

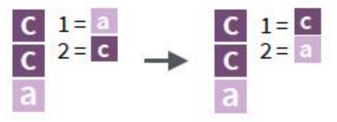


fct_unify(fs, levels = lvls_union(fs)**)** Standardize levels across a list of factors. fct_unify(list(f2, f1))

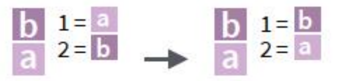
Change the order of levels



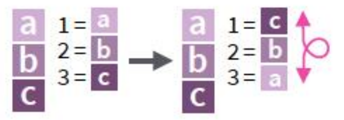
fct_relevel(.f, ..., after = 0L**)** Manually reorder factor levels. fct_relevel(f, c("b", "c", "a"))



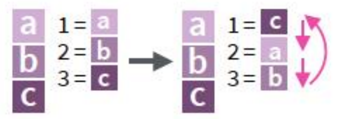
fct_infreq(f, ordered = NA**)** Reorder levels by the frequency in which they appear in the data (highest frequency first). Also **fct_inseq()**.
f3 <- factor(c("c", "c", "a"))
fct_infreq(f3)



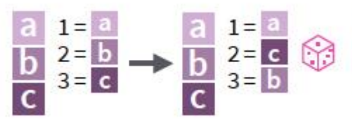
fct_inorder(f, ordered = NA**)** Reorder levels by order in which they appear in the data. fct_inorder(f2)



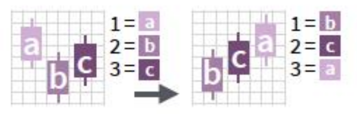
fct_rev(f**)** Reverse level order.
f4 <- factor(c("a","b","c"))
fct_rev(f4)



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

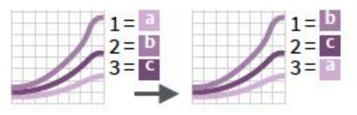


fct_shuffle(f, n = 1L**)** Randomly permute order of factor levels. fct_shuffle(f4)



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

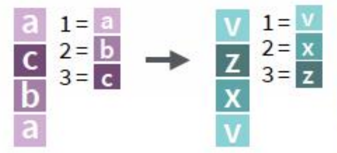
```
boxplot(
 PlantGrowth,
 weight ~ fct_reorder(group, weight)
)
```



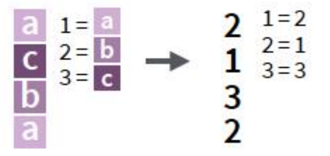
fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE**)** Reorder levels by their final values when plotted with two other variables.

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

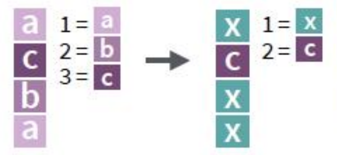
Change the value of levels



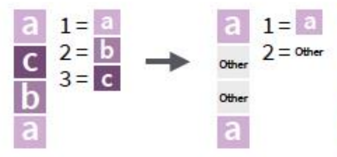
fct_recode(.f, ...**)** Manually change levels. Also **fct_relabel()** which obeys purrr::map syntax to apply a function or expression to each level.
fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))



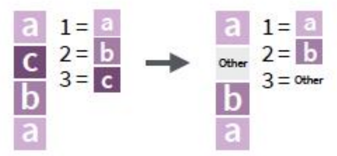
fct_anon(f, prefix = ""**)** Anonymize levels with random integers. fct_anon(f)



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

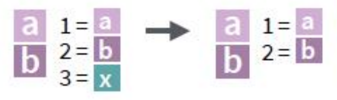


fct_lump_min(f, min, w = NULL, other_level = "Other"**)** Lumps together factors that appear fewer than min times. Also **fct_lump_n()**, **fct_lump_prop()**, and **fct_lump_lowfreq()**. fct_lump_min(f, min = 2)

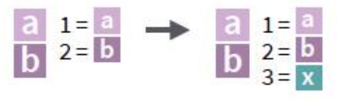


fct_other(f, keep, drop, other_level = "Other"**)** Replace levels with "other." fct_other(f, keep = c("a", "b"))

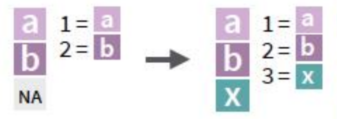
Add or drop levels



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)")

Data transformations with *dplyr*

General *dplyr* notes

- All *dplyr* functions take a data argument; data argument could be:
  - <u>First argument:</u> select(ds, id)
  - OR
  - <u>Piped in:</u> ds %>% select(id)
- All *dplyr* transformations are temporary unless saved back to the dataset
  - ds <- ds %>% select(id)

General *dplyr* notes

- All *dplyr* transformations can be chained together with pipes %>%
  - ds %>% filter(id > 0) %>% select(id:por_x) %>% mutate(por_x = por_x + 5) %>% arrange(id)
  - Pipe keyboard shortcuts
    - Windows = Ctrl + Shift + M
    - Mac = Cmd + Shift + M

Using *dplyr* helps to avoid inflexible, inefficientcode

- Instead of “hard” coding based onposition (such as ds[1, 2]), *dplyr* allows you to filter and select by name and logical conditions
- Instead of saving multiple subsets of data to calculate summaries, use group_by in *dplyr* to summarize within groups
- Instead of typing out long lists of column names and functions, *dplyr* helper functions let you select columns in a variety of ways and apply multiple transformations to selected functions at once

General *dplyr* notes

- All *dplyr* transformations have loads of powerful options that you might want
  - Read the documentation and examples

Data transformation with dplyr : : **CHEATSHEET**

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes

x |> f(y) becomes **f(x, y)**

Summarize Cases

Apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

summary function

summarize(.data, ...)
Compute table of summaries.
mtcars |> summarize(avg = mean(mpg))

count(.data, ..., wt = NULL, sort = FALSE, name = NULL) Count number of rows in each group defined by the variables in ... Also **tally()**, **add_count()**, **add_tally()**.
mtcars |> count(cyl)

Group Cases

Use **group_by**(.data, ..., .add = FALSE, .drop = TRUE) to create a "grouped" copy of a table grouped by columns in ... dplyr functions will manipulate each "group" separately and combine the results.

```
mtcars |>
  group_by(cyl) |>
  summarize(avg = mean(mpg))
```

Use **rowwise**(.data, ...) to group data into individual rows. dplyr functions will compute results for each row. Also apply functions to list-columns. See tidyr cheat sheet for list-column workflow.

```
starwars |>
  rowwise() |>
  mutate(film_count = length(films))
```

ungroup(x, ...) Returns ungrouped copy of table.
g_mtcars <- mtcars |> group_by(cyl)
ungroup(g_mtcars)

Manipulate Cases

EXTRACT CASES

Row functions return a subset of rows as a new table.

filter(.data, ..., .preserve = FALSE) Extract rows that meet logical criteria.
mtcars |> filter(mpg > 20)

distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values.
mtcars |> distinct(gear)

slice(.data, ..., .preserve = FALSE) Select rows by position.
mtcars |> slice(10:15)

slice_sample(.data, ..., n, prop, weight_by = NULL, replace = FALSE) Randomly select rows. Use n to select a number of rows and prop to select a fraction of rows.
mtcars |> slice_sample(n = 5, replace = TRUE)

slice_min(.data, order_by, ..., n, prop, with_ties = TRUE) and **slice_max()** Select rows with the lowest and highest values.
mtcars |> slice_min(mpg, prop = 0.25)

slice_head(.data, ..., n, prop) and **slice_tail()** Select the first or last rows.
mtcars |> slice_head(n = 5)

Logical and boolean operators to use with filter()

==	<	<=	is.na()	%in%	|	xor()
!=	>	>=	!is.na()	!	&	

See **?base::Logic** and **?Comparison** for help.

ARRANGE CASES

arrange(.data, ..., .by_group = FALSE) Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.
mtcars |> arrange(mpg)
mtcars |> arrange(desc(mpg))

ADD CASES

add_row(.data, ..., .before = NULL, .after = NULL) Add one or more rows to a table.
cars |> add_row(speed = 1, dist = 1)

Manipulate Variables

EXTRACT VARIABLES

Column functions return a set of columns as a new vector or table.

pull(.data, var = -1, name = NULL, ...) Extract column values as a vector, by name or index.
mtcars |> pull(wt)

select(.data, ...) Extract columns as a table.
mtcars |> select(mpg, wt)

relocate(.data, ..., .before = NULL, .after = NULL) Move columns to new position.
mtcars |> relocate(mpg, cyl, .after = last_col())

Use these helpers with select() and across()
e.g. mtcars |> select(mpg:cyl)

contains(match)	**num_range**(prefix, range)	**:**, e.g., mpg:cyl
ends_with(match)	**all_of(x)/any_of**(x, ..., vars)	**!**, e.g., !gear
starts_with(match)	**matches**(match)	**everything()**

MANIPULATE MULTIPLE VARIABLES AT ONCE

df <- tibble(x_1 = c(1, 2), x_2 = c(3, 4), y = c(4, 5))

across(.cols, .funs, ..., .names = NULL) Summarize or mutate multiple columns in the same way.
df |> summarize(across(everything(), mean))

c_across(.cols) Compute across columns in row-wise data.
```
df |>
  rowwise() |>
  mutate(x_total = sum(c_across(1:2)))
```

MAKE NEW VARIABLES

Apply **vectorized functions** to columns. Vectorized functions take vectors as input and return vectors of the same length as output (see back).

vectorized function

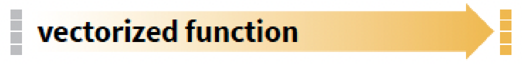
mutate(.data, ..., .keep = "all", .before = NULL, .after = NULL) Compute new column(s). Also **add_column()**.
mtcars |> mutate(gpm = 1 / mpg)
mtcars |> mutate(gpm = 1 / mpg, .keep = "none")

rename(.data, ...) Rename columns. Use **rename_with()** to rename with a function.
mtcars |> rename(miles_per_gallon = mpg)

Vectorized Functions

TO USE WITH MUTATE ()

mutate() applies vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.



OFFSET

dplyr::**lag()** - offset elements by 1
dplyr::**lead()** - offset elements by -1

CUMULATIVE AGGREGATE

dplyr::**cumall()** - cumulative all()
dplyr::**cumany()** - cumulative any()
cummax() - cumulative max()
dplyr::**cummean()** - cumulative mean()
cummin() - cumulative min()
cumprod() - cumulative prod()
cumsum() - cumulative sum()

RANKING

dplyr::**cume_dist()** - proportion of all values <=
dplyr::**dense_rank()** - rank w ties = min, no gaps
dplyr::**min_rank()** - rank with ties = min
dplyr::**ntile()** - bins into n bins
dplyr::**percent_rank()** - min_rank scaled to [0,1]
dplyr::**row_number()** - rank with ties = "first"

MATH

+, - , *, /, ^, %/%, %% - arithmetic ops
log(), log2(), log10() - logs
<, <=, >, >=, !=, == - logical comparisons
dplyr::**between()** - x >= left & x <= right
dplyr::**near()** - safe == for floating point numbers

MISCELLANEOUS

dplyr::**case_when()** - multi-case if_else()

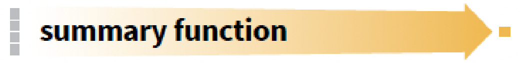
```
starwars |>
  mutate(type = case_when(
    height > 200 | mass > 200 ~ "large",
      species == "Droid"       ~ "robot",
      TRUE                     ~ "other")
  )
```

dplyr::**coalesce()** - first non-NA values by element across a set of vectors
dplyr::**if_else()** - element-wise if() + else()
dplyr::**na_if()** - replace specific values with NA
pmax() - element-wise max()
pmin() - element-wise min()

Summary Functions

TO USE WITH SUMMARIZE ()

summarize() applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.



COUNT

dplyr::**n()** - number of values/rows
dplyr::**n_distinct()** - # of uniques
sum(!is.na()) - # of non-NAs

POSITION

mean() - mean, also **mean(!is.na())**
median() - median

LOGICAL

mean() - proportion of TRUEs
sum() - # of TRUEs

ORDER

dplyr::**first()** - first value
dplyr::**last()** - last value
dplyr::**nth()** - value in nth location of vector

RANK

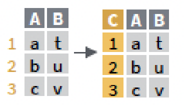
quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

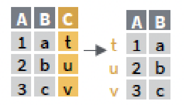
IQR() - Inter-Quartile Range
mad() - median absolute deviation
sd() - standard deviation
var() - variance

Row Names

Tidy data does not use rownames, which store a variable outside of the columns. To work with the rownames, first move them into a column.



tibble::**rownames_to_column()**
Move row names into col.
a <- mtcars |>
rownames_to_column(var = "C")

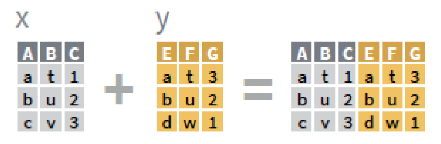


tibble::**column_to_rownames()**
Move col into row names.
a |> column_to_rownames(var = "C")

Also tibble::**has_rownames()** and tibble::**remove_rownames()**.

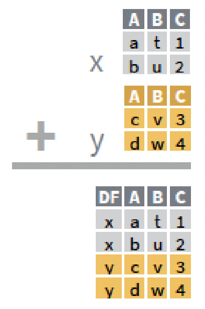
Combine Tables

COMBINE VARIABLES



bind_cols(..., .name_repair) Returns tables placed side by side as a single table. Column lengths must be equal. Columns will NOT be matched by id (to do that look at Relational Data below), so be sure to check that both tables are ordered the way you want before binding.

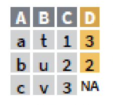
COMBINE CASES



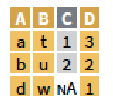
bind_rows(..., .id = NULL) Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured).

RELATIONAL DATA

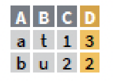
Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.



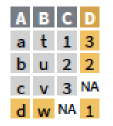
left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join matching values from y to x.



right_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join matching values from x to y.

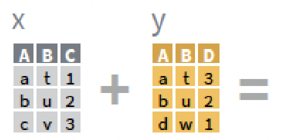


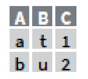
inner_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join data. Retain only rows with matches.



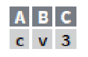
full_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join data. Retain all values, all rows.

Use a "**Filtering Join**" to filter one table against the rows of another.



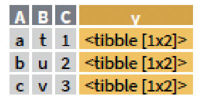


semi_join(x, y, by = NULL, copy = FALSE, ..., na_matches = "na") Return rows of x that have a match in y. Use to see what will be included in a join.



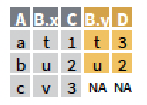
anti_join(x, y, by = NULL, copy = FALSE, ..., na_matches = "na") Return rows of x that do not have a match in y. Use to see what will not be included in a join.

Use a "**Nest Join**" to inner join one table to another into a nested data frame.

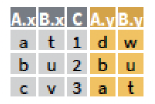


nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, ...) Join data, nesting matches from y in a single new data frame column.

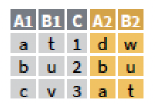
COLUMN MATCHING FOR JOINS



Use **by = c("col1", "col2", ...)** to specify one or more common columns to match on.
left_join(x, y, by = "A")

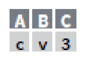


Use a named vector, **by = c("col1" = "col2")**, to match on columns that have different names in each table.
left_join(x, y, by = c("C" = "D"))

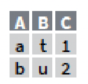


Use **suffix** to specify the suffix to give to unmatched columns that have the same name in both tables.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

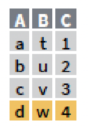
SET OPERATIONS



intersect(x, y, ...)
Rows that appear in both x and y.



setdiff(x, y, ...)
Rows that appear in x but not y.



union(x, y, ...)
Rows that appear in x or y, (duplicates removed). **union_all()** retains duplicates.

Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

Data wrangling with *dplyr* package **filter**, **select**, **pull,** and **arrange**

- Subset by rows (*filter*) or column (*select*)
  - ds %>% filter(id > 0): select rows id > 0
  - ds %>% select(id): select the column id
  - filter and select return tibbles (unless you reassign to ds)
- *Pull* grabs values and returns as a vector
  - ds %>% filter(id > 0)%>% pull(id): returns a vector of ids (that are greater than 0)
- *Arrange* sorts by columns

Data wrangling with *dplyr* package **rename** and **mutate**

- *Rename* changes column names
  - ds %>% rename(old_column = new_column)
- *Mutate* changes the values in a column
  - ds %>% mutate(id = id + 5)
- *Mutate can* also create new columns
  - ds %>% mutate(id_plus_5 = id + 5)

Data wrangling with *dplyr* package **summarize** and **group_by**

- *summarize* collapses data down to a single row
  - ds %>% summarize(mean_porx = mean(por_x))
- *group_by* makes transformations apply within groups (e.g., factors)
  - ds %>% group_by(id, condition) %>% summarize(mean_porx = mean(por_x))

Within select(), helper functions can make it easy to select **columns**

- Powerful, flexible options for finding columns
  - starts_with and ends_with (contains a string)
    - select (starts_with(“neo_”))
  - -variable (take everything except variable)
    - select(-id)
  - var1:var4 (take everything from var1 to var4)
    - select(neo_01:neo_10)
  - where(is.factor) (take variables that are a factor)
    - select(where(is.factor))
- Don’t try to use these to select rows!

across() is a powerful helper to use with mutate and summarize

- across(column_selection, function_to_apply)
  - summarize(across(var1:var4, mean)) will summarize by taking the mean of var1 to var4
  - savesyou from typing: summarize(var1 = mean(var1), var2 =mean(var2), var3 = mean(var3), var4 = mean(var4)
  - can apply multiple functions: summarize(across(starts_with("item"), list(mean = mean, sd =sd))

Tutorial: Data transformations