



TIDY VERSE PART 1

dr Maria Kubara, WNE UW

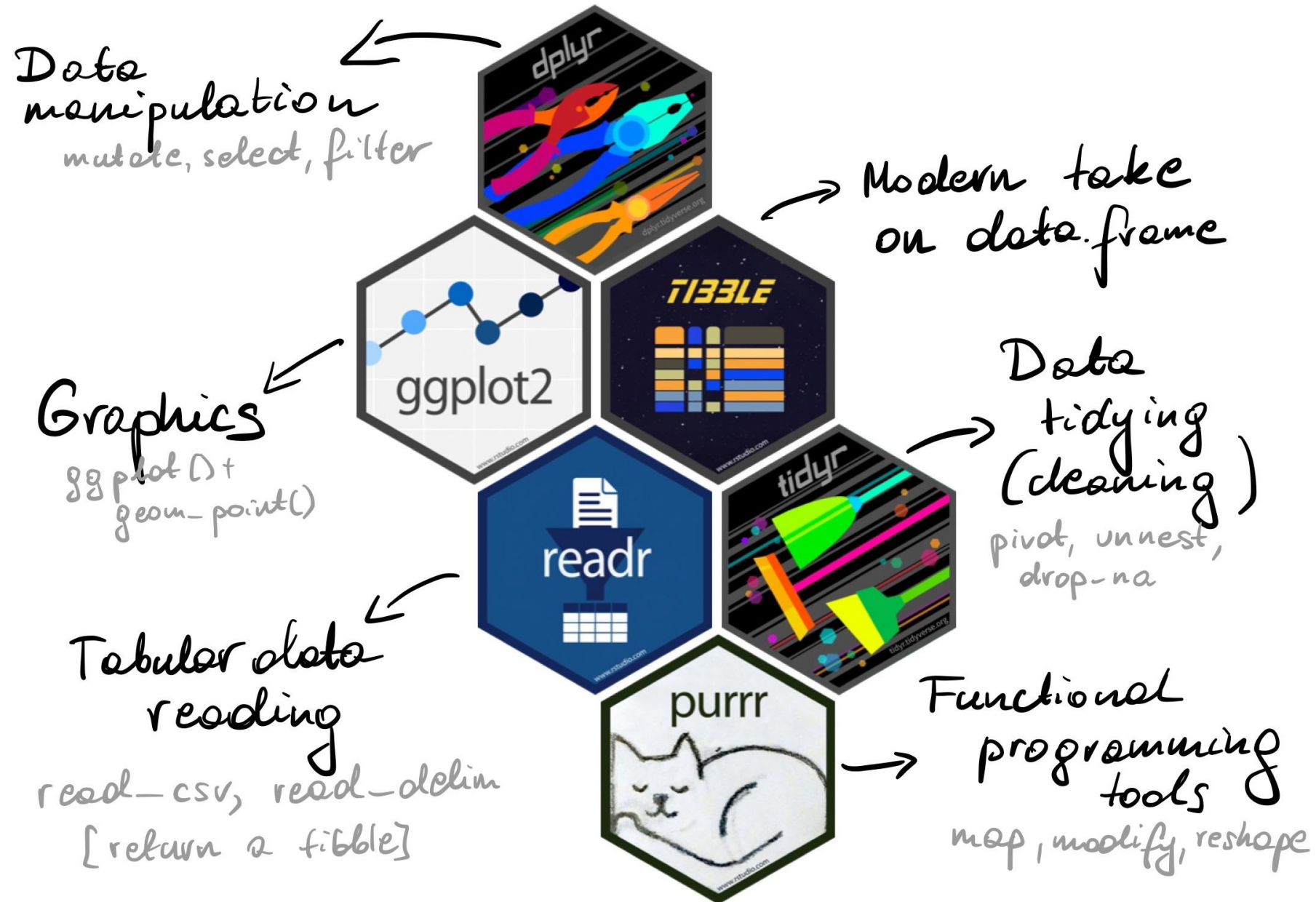
TIDYVERSE

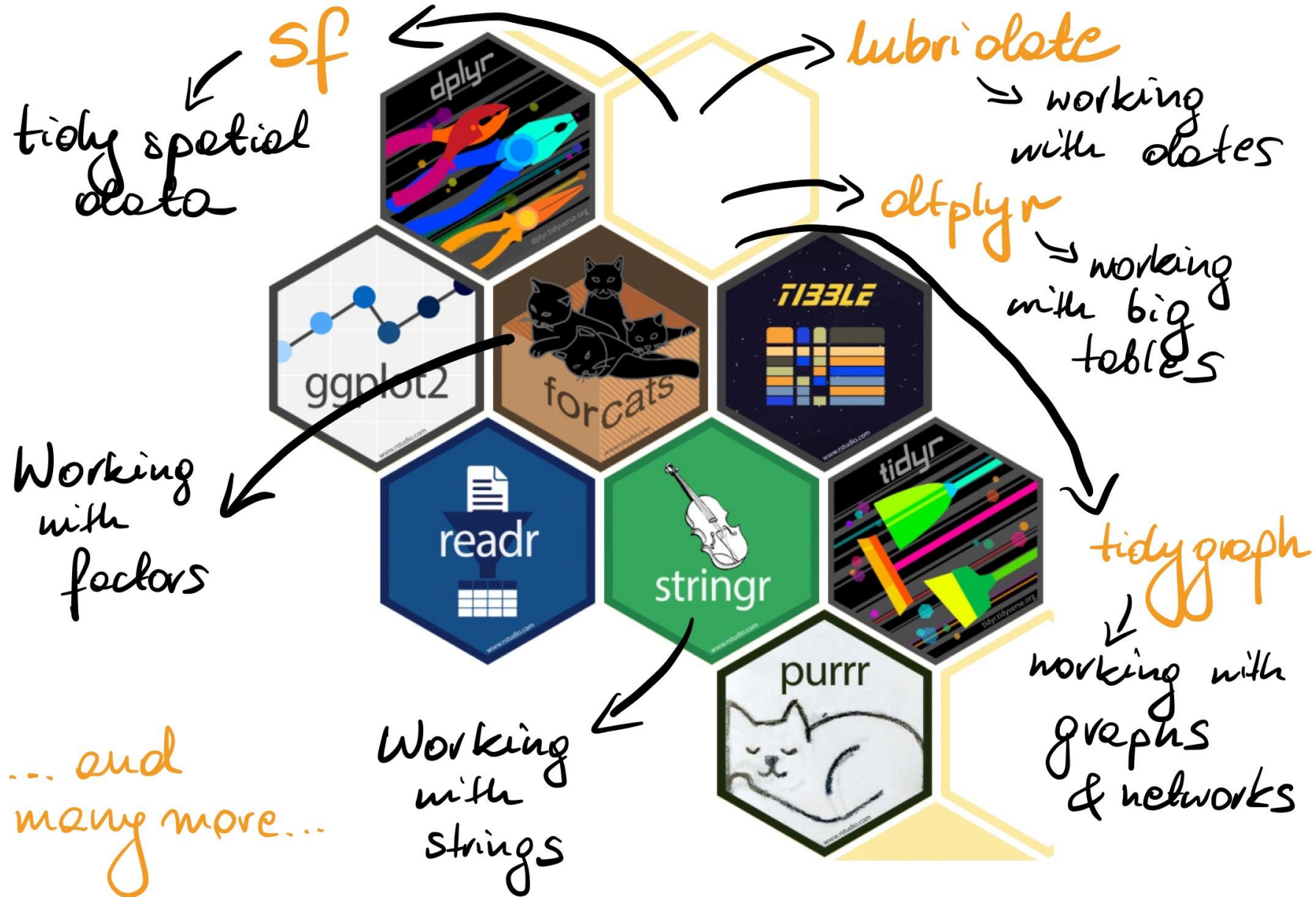
- Environment with packages for easier and tidier data processing in R
- All based on tibble structure (refreshed data.frame)
- Concerned about the tidy data
- „All packages share an underlying design philosophy, grammar, and data structures”



Install the complete tidyverse with:

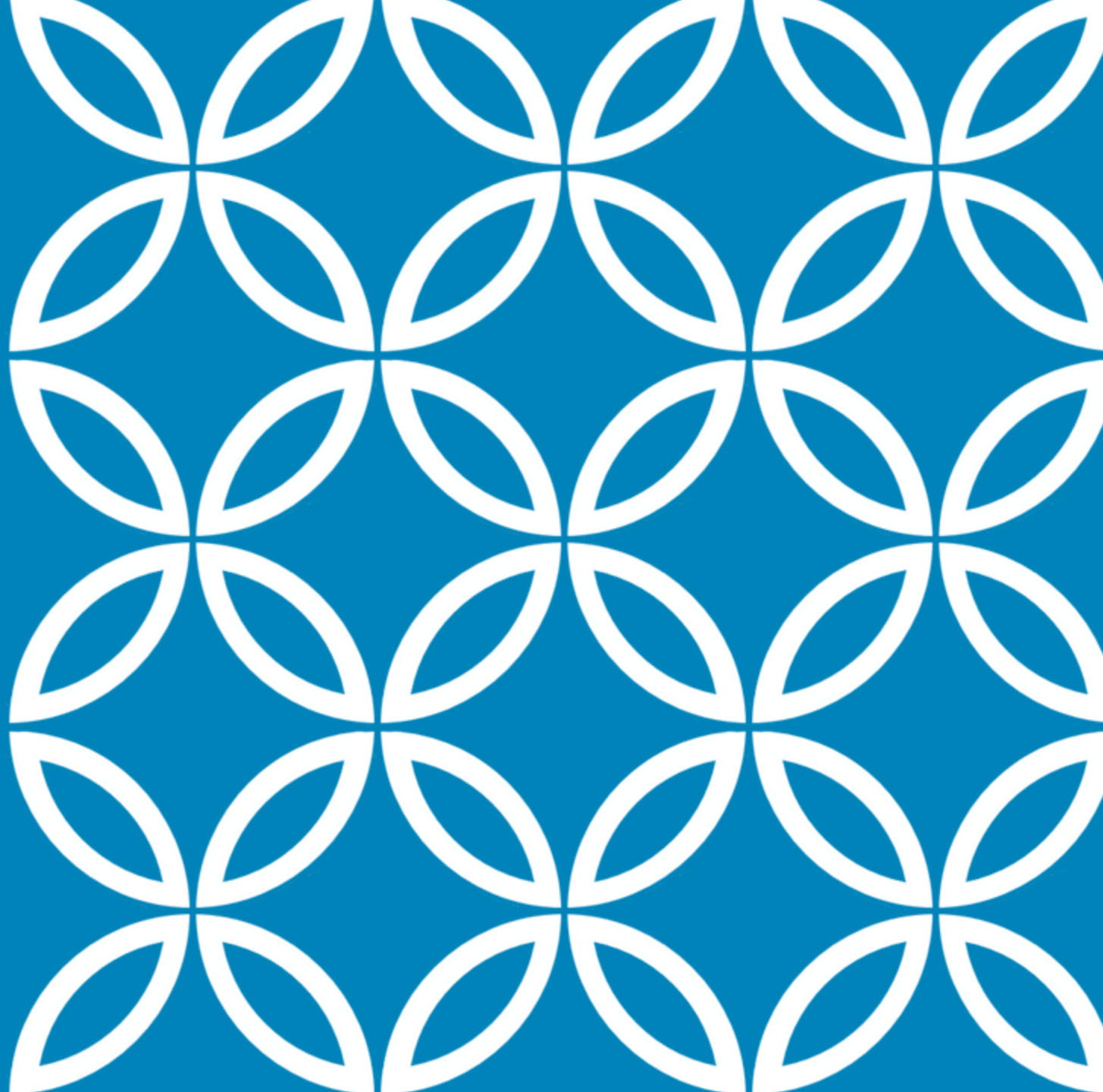
```
install.packages("tidyverse")
```





TIDY DATA

Golden standard in data analytics

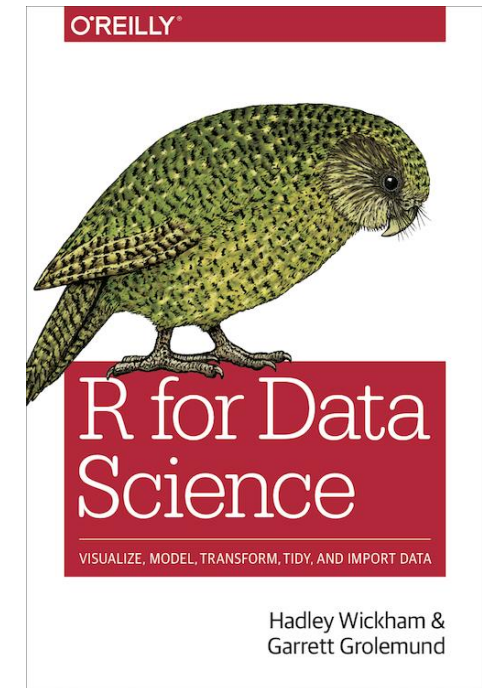


TIDYVERSE

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types.

In tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.



TIDY DATA MANIPULATION

Four fundamental **verbs** of data manipulation:

- Filter: subsetting or removing observations based on some condition.
- Transform: adding or modifying variables. These modifications can involve either a single variable (e.g., log-transformation), or multiple variables (e.g., computing density from weight and volume).
- Aggregate: collapsing multiple values into a single value (e.g., by summing or taking means).
- Sort: changing the order of observations.

BASIC DATA MANIPULATION WITH DPLYR

Filter

`filter()`

Aggregate

`summarize()`

`count()`

*Group-wise
operations:
`group_by()`
`rowwise()`*

Transform

`mutate()`

Sort

`arrange()`

CONTEXT FOR TIDYVERSE FUNCTIONS

All tidyverse functions take DATA as their first argument and work with its context. In this way we can just specify the name of a variable (column) and use it without any \$ symbols to show its source.

```
filter(iris, Sepal.Length > 2)
```

filter function „knows” that Sepal.Length is a variable within iris dataset

```
mutate(iris, Double.Sepal.Length = Sepal.Length*2)
```

A new variable Double.Sepal.Length is created in the iris dataset
which uses the values in Sepal.Length to create its own

In tidyverse there is a convention: new goes first, older comes second
Here we first specify the name of the new variable and then get its
value (based on the older info already present in the dataset)

PIPELINE OPERATOR

Handy shortcut in
RStudio to write a
pipeline: Ctrl+Shift+M



%>% pipeline operator allows for passing on the result from the previous operation as the first input for the next function (or where the dot is – for more control)

```
filter(iris, Sepal.Length > 2)
iris %>% filter(Sepal.Length > 2)
iris %>% filter(., Sepal.Length > 2)
```

Note: in R 4.0 „base pipe” has been introduced, written as `|>`. It controls the function flow in a similar way like the pipeline operator. You can use it without loading any additional package.

GROUP-WISE OPERATIONS

`group_by()` „splits” the data by given group division and allows for processing the dataset separately for given groups

Mostly used for summary table creation or by adding a variable which value differs by groups

Similarly `rowwise()` allows for processing data for each row individually (`group_by` is far more popular)

ID	VAR	GR
1	—	GR1
2	—	GR2
3	—	GR1
4	—	GR2
5	—	GR2

`group_by(GR)` ← name of the grouping variable(s)

ID	VAR	GR
1	—	GR1
3	—	GR1

ID	VAR	GR
2	—	GR2
4	—	GR2
5	—	GR2

← data is now processed separately

`modify(in groups)`
mutate, filter, etc.
and `ungroup()`
to go back to full data

ID	VAR	GR	NEW
1	—	GR1	*
2	—	GR2	#
3	—	GR1	*
4	—	GR2	#
5	—	GR2	#

`Summarize()`
to get a summary table by group

GR	SUM.N
GR1	2
GR2	3

OTHER HELPFUL DPLYR FUNCTIONS

Dplyr function	Description
<code>rename()</code>	Changes the name of a given variable (new name first)
<code>distinct()</code>	Remove duplicated rows in the dataset
<code>slice()</code>	Select rows by positions (specify vector of positions which you'd like to get)
<code>add_row()</code>	Add new row to data (arguments go as names of variables and their value)
<code>bind_cols()</code>	Bind two datasets by columns (like <code>cbind</code>)
<code>bind_rows()</code>	Bind two datasets by rows (like <code>rbind</code>)
<code>left_join()/right_join()</code> <code>inner_join()/full_join()</code>	Merging functions which take their names exactly from SQL syntax
<code>%in%</code>	Operator for the <code>filter()</code> function, allows for checking if a value is within the values given in a longer vector e.g. <code>gender %in% c("male","female")</code>
<code>glimpse</code>	Get an overview of the data (like <code>summary/str</code> function)

Data transformation with dplyr : : CHEAT SHEET



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**



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

Summarise 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



summarise(.data, ...)
Compute table of summaries.
`summarise(mtcars, 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()**.
`count(mtcars, 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) %>%
summarise(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 tidyverse cheat sheet for list-column workflow.



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

ungroup(x, ...) Returns ungrouped copy of table.
`ungroup(g_mtcars)`

Manipulate Cases

EXTRACT CASES

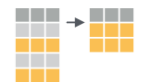
Row functions return a subset of rows as a new table.



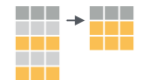
filter(.data, ..., preserve = FALSE) Extract rows that meet logical criteria.
`filter(mtcars, mpg > 20)`



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



slice(.data, ..., preserve = FALSE) Select rows by position.
`slice(mtcars, 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.
`slice_sample(mtcars, 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.
`slice_min(mtcars, mpg, prop = 0.25)`



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

Logical and boolean operators to use with filter()

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

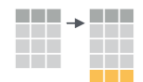
See **?base::Logic** and **?Comparison** for help.

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.
`arrange(mtcars, mpg)`
`arrange(mtcars, desc(mpg))`

ADD CASES



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

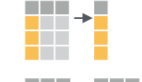
Manipulate Variables

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.
`pull(mtcars, wt)`



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



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

Use these helpers with select() and across()

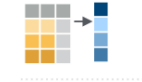
e.g. `select(mtcars, mpg:cyl)`

contains(match)	num_range(prefix, range)	; e.g. <code>mpg:cyl</code>
ends_with(match)	all_of(x)/any_of(x, ..., vars)	-; e.g. <code>-gear</code>
starts_with(match)	matches(match)	everything()

MANIPULATE MULTIPLE VARIABLES AT ONCE



across(.cols, .funs, ..., .names = NULL) Summarise or mutate multiple columns in the same way.
`summarise(mtcars, across(everything(), mean))`

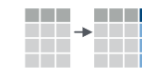


c_across(.cols) Compute across columns in row-wise data.
`transmute(rowwise(UKgas), total = sum(c_across(1:2)))`

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()**, **add_count()**, and **add_tally()**.
`mutate(mtcars, gpm = 1 / mpg)`



transmute(.data, ...) Compute new column(s), drop others.
`transmute(mtcars, gpm = 1 / mpg)`



rename(.data, ...) Rename columns. Use **rename_with()** to rename with a function.
`rename(cars, distance = dist)`

Vectorized Functions

TO USE WITH MUTATE ()

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

vectorized function

OFFSET

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

CUMULATIVE AGGREGATE

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

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

summary function

COUNT

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

POSITION

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

LOGICAL

mean() - proportion of TRUE's
sum() - # of TRUE's

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 <- rownames_to_column(mtcars,
 var = "C")

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

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

Combine Tables

COMBINE VARIABLES

X + Y =

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.

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.

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"))

COMBINE CASES

X + Y =

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).

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

X + Y =

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.

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).

