

Lecture 12: Data analysis using R and **tidyverse**

Matthew Chan

Postdoctoral Fellow, Fred Hutch
mchan3@fredhutch.org

MCB 536: Tools for Computational Biology
4 November 2025

Python vs R



- Python is designed as a general purpose programming language.
- R is designed for **statistical and mathematical** computing.
- Both have respective packages for data exploration, manipulation, and visualization.

Goals for this week's classes

- Introduce basic **R** syntax and logic
- Learn how to use **R** and **tidyverse** for data analysis and visualization of tabular data.

tidyverse - collection of packages for data science

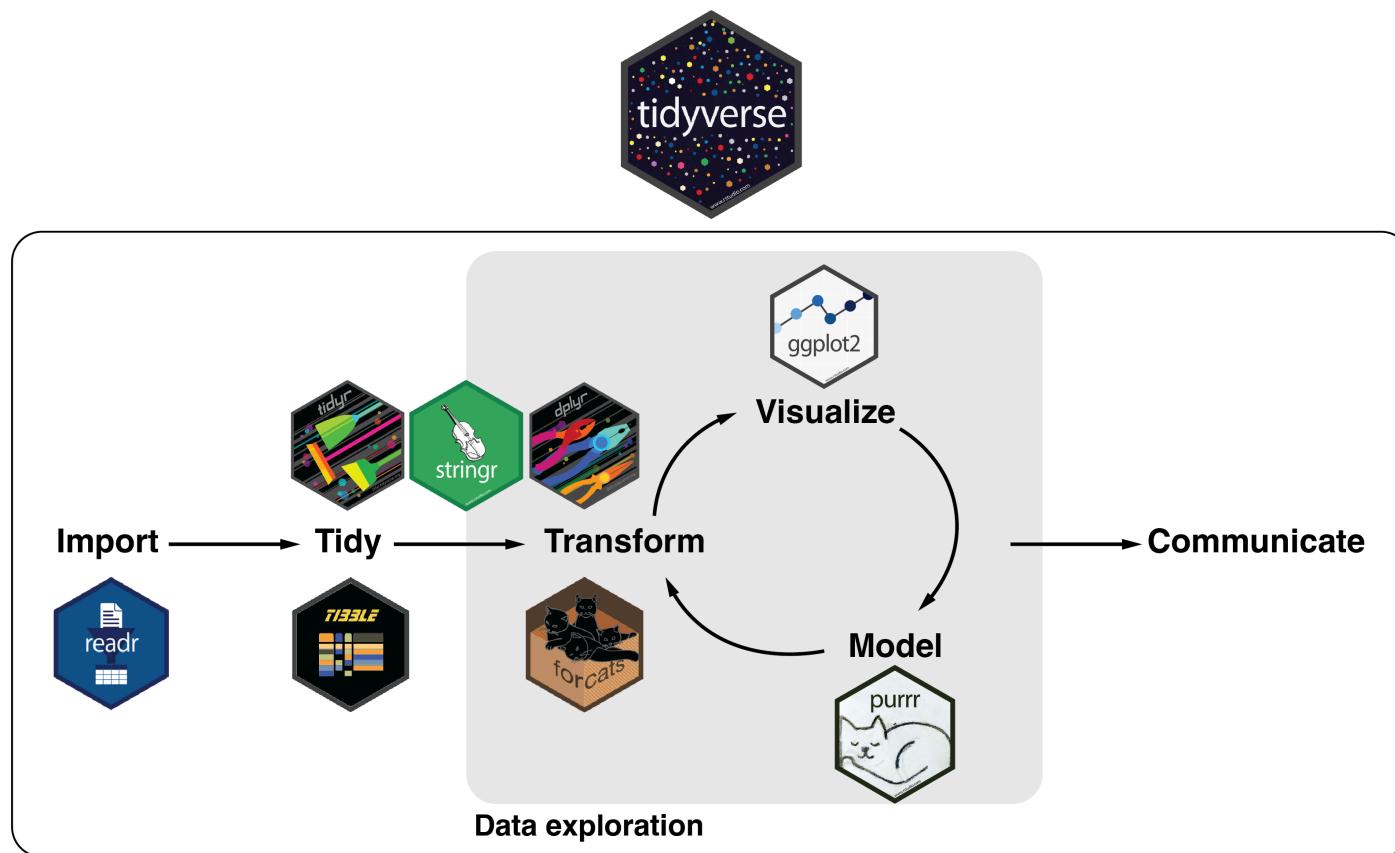


Figure adapted from [R for Data Science, 2017](#)

Loading packages

- In **python**:

```
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
```

- In **R**:

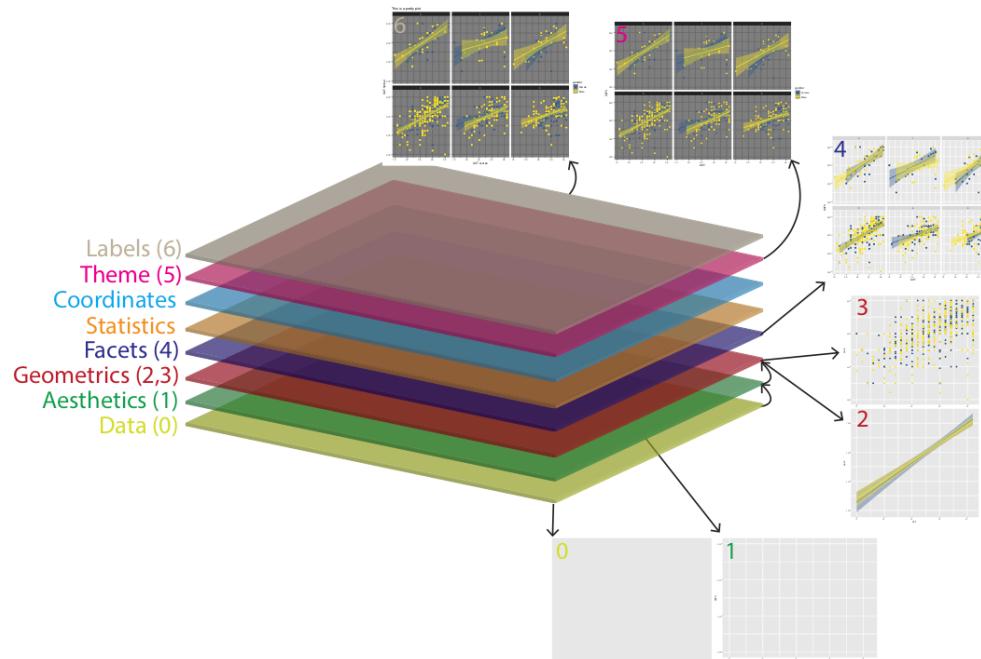
```
library(tidyverse)
```

Plotting data: The grammar of **ggplot**

ggplot2 follows its own “grammar” rules to iteratively add to or update plots.

```
ggplot(data) + aes(x = x_axis) + geometry() + layers...
```

1. **ggplot(data)** - specifies what data to plot. Usually **data** is a **tibble**.
2. **aes()** - specifies the axes or variables to plot
3. **geom** - specifies the type of plot.
4. **layers** - additional content to the plot (titles, axis titles, legend, etc) ...



Useful resources for `ggplot`

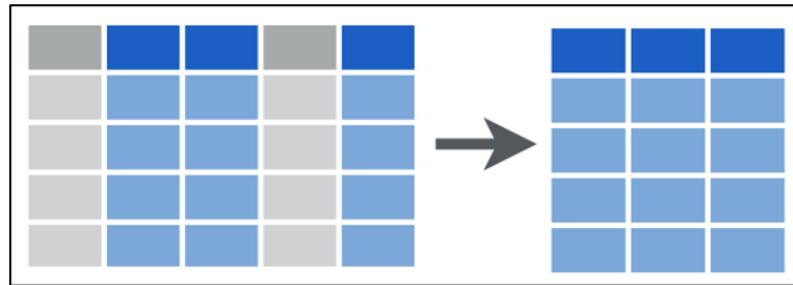
- [ggplot2 documentation](#)
- [R graphics cookbook](#) - examples and resources on all things plots in R.
- [R graph gallery](#) - Examples of different plot types and additional features/layers to add.

Tidying data: Selecting columns with `select()`

`tidyverse` has built in functions for tidying, manipulating, transforming tabular data.

1. `select()` to select columns in a tibble, or dataframe

```
select(dataframe, column_name1, column_name2, ...)
```



Example: Keep `mean_yfp` and `mean_rfp` columns:

```
select(data, mean_yfp, mean_rfp)
```

Tidying data: Keep rows with `filter()`

2. `filter()` to filter rows based on a condition

```
filter(dataframe, logical_expression, logical_expression, ...)
```



Example: Keep all rows that have `kozak_region` as A:

```
filter(data, kozak_region == "A")
```

Example: Keep rows with `mean_yfp` greater than 4500 and `kozak_region` as A:

```
filter(data, kozak_region == "A", mean_yfp > 4500)
```

Comparison and logical operators

Use comparisons operators for making comparisons. Comparisons expression return either **TRUE** or **FALSE**.

- `==` - equals to
- `>` - greater than
- `>=` - greater than or equals to
- `<` - less than
- `<=` - less than or equals to
- `!=` - not equals to (`!` signifies **not**)

Comparisons can be combined with logical operators to make more details comparisons.

- `|` - OR
- `&` - AND

%>% is the piping operator

The pipe operator (`%>%`) takes the output of one command, and directly inputs it into the next command.

You can do this (setting intermediate objects):

```
1 data <- read.tsv( "data/example_dataset_1.tsv" )  
2  
3 selected_columns <- select(data, strain, mean_ratio,  
4                               insert_sequence, kozak_region)  
5  
6 tidy_dataframe <- filter(selected_columns, kozak_region == "A" )
```

%>% is the piping operator

The pipe operator (`%>%`) takes the output of one command, and directly inputs it into the next command.

You can do this (setting intermediate objects):

```
1 data <- read.tsv("data/example_dataset_1.tsv")
2
3 selected_columns <- select(data, strain, mean_ratio,
4                               insert_sequence, kozak_region)
5
6 tidy_dataframe <- filter(selected_columns, kozak_region == "A")
```

But this is better:

```
1 tidy_dataframe <- data %>%
2   select(., strain, mean_ratio, insert_sequence, kozak_region) %>%
3   filter(., kozak_region == "A")
```

%>% is the piping operator

The pipe operator (`%>%`) takes the output of one command, and directly inputs it into the next command.

You can also do this (nesting functions):

```
1 data <- read.tsv("data/example_dataset_1.tsv")
2
3 tidy_dataframe <- filter(select(data, strain,
4 mean_ratio,insert_sequence, kozak_region), kozak_region == "A")
```

%>% is the piping operator

The pipe operator (`%>%`) takes the output of one command, and directly inputs it into the next command.

You can also do this (nesting functions):

```
1 data <- read.tsv("data/example_dataset_1.tsv")
2
3 tidy_dataframe <- filter(select(data, strain,
4     mean_ratio, insert_sequence, kozak_region), kozak_region == "A")
```

But this is so much better:

```
1 tidy_dataframe <- data %>%
2     select(strain, mean_ratio, insert_sequence, kozak_region) %>%
3     filter(kozak_region == "A")
```

Note: The `.` in the first argument of `filter()` and `select()` is optional. When piping, `filter()` and `select()` will assume to process the output from the previous step.

%>% is the piping operator

Piping in R is like baking

```
slice <- slice(decorate(bake(mix(ingredients))))
```

? <-    %>%

mix() %>%

bake() %>%

decorate()%>%

slice()

By @ArthurWelle / Images from flaticon.com

Source: [Arthur Welle](#)

`%>%` is the piping operator

Piping in R is like baking

```
slice <- slice(decorate(bake(mix(ingredients))))
```

? <-    `%>%`

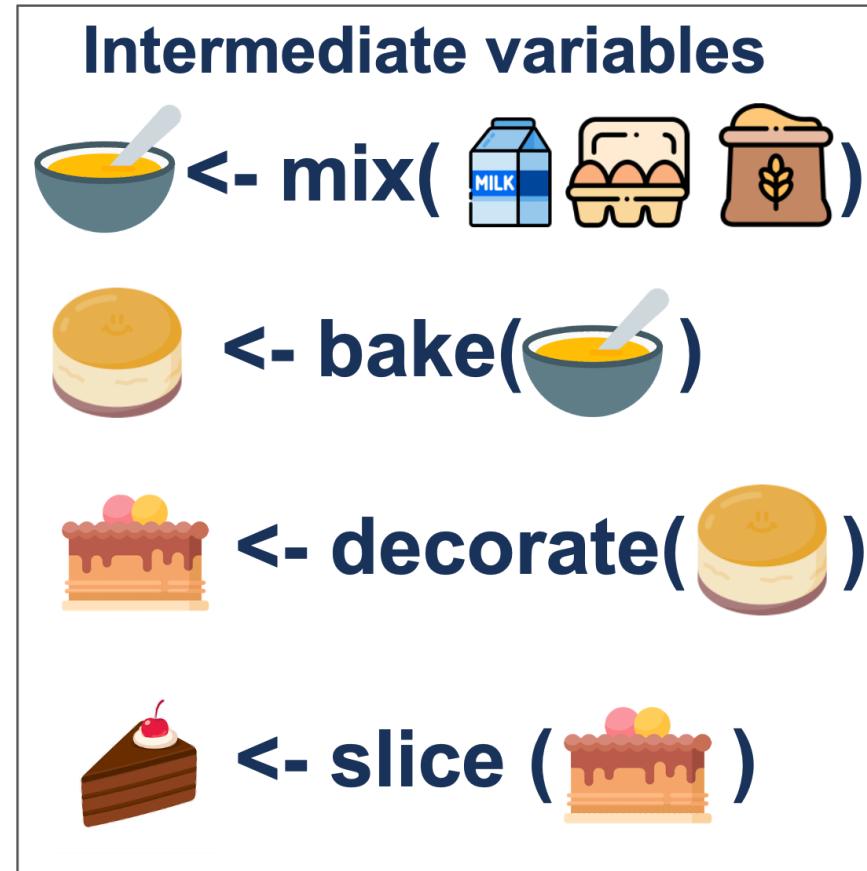
`mix() %>%`

`bake() %>%`

`decorate() %>%`

`slice()`

By @ArthurWelle / Images from flaticon.com

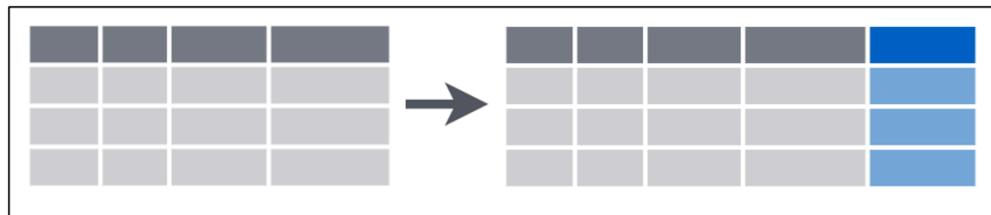


Source: [Arthur Welle](#)

Create new columns with `mutate()`

Usage:

```
mutate(new_column_name = old_column_name)
```



Example: Make a new column called `sum_fluorescence`:

```
data %>%
  mutate(sum_fluorescence = mean_yfp + mean_rfp)
```

Combine tables with **join** functions

[dplyr join documentation](#)

Variants:

1. `inner_join(x, y)` - keeps only observations from **x** that match **y**
2. `left_join(x, y)` - keeps all observations in **x**
3. `right_join(x, y)` - keeps all observations in **y**
4. `full_join(x, y)` - keeps all observations in **x** and **y**

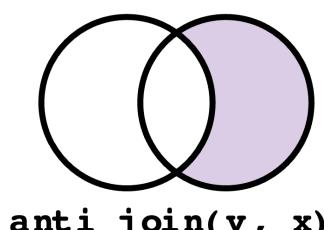
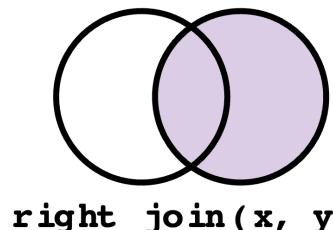
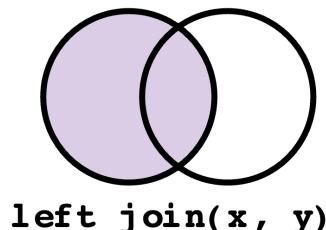
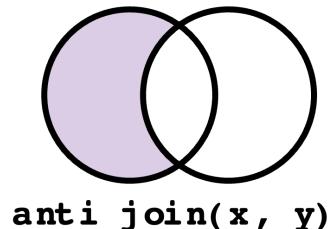
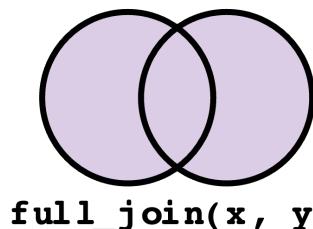
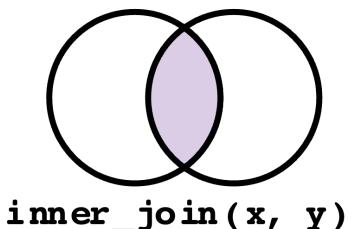
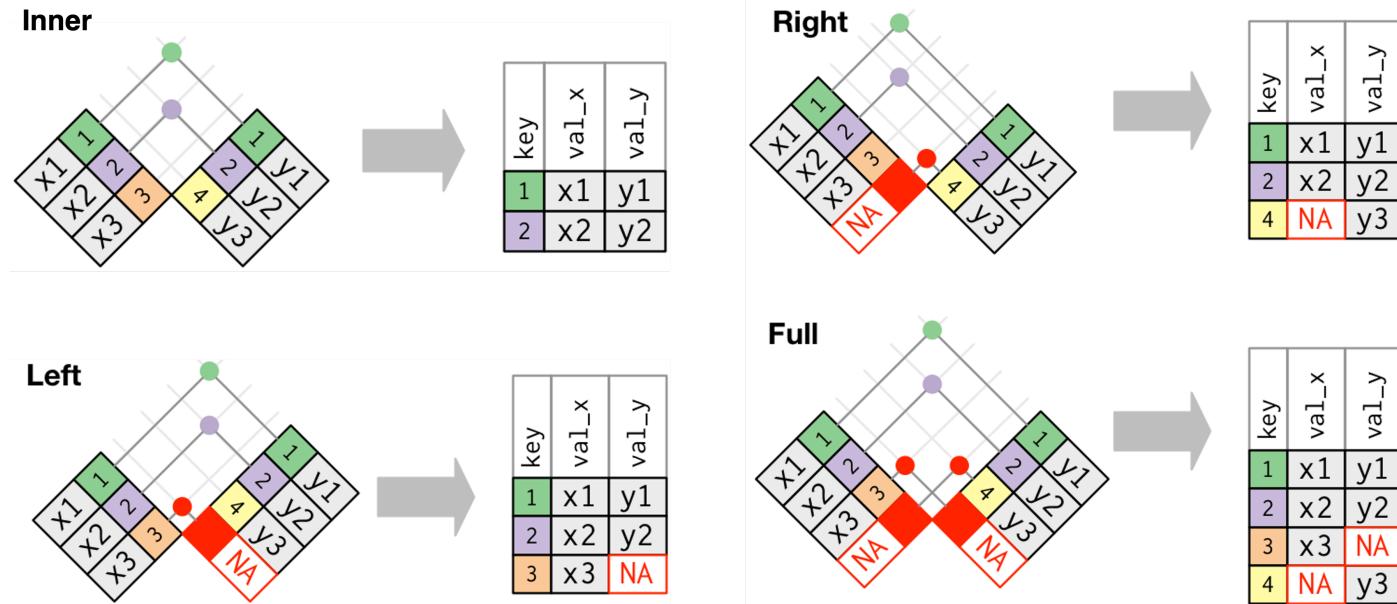


Image credit: <https://tavareshugo.github.io/r-intro-tidyverse-gapminder/08-joins/index.html>

Combine tables with **join** functions



Usage:

```
inner_join(x, y, by = "key")
left_join(x, y, by = "key")
```

Image credit: <https://tavareshugo.github.io/r-intro-tidyverse-gapminder/08-joins/index.html>

Data manipulation - **summarize()** and **group_by()**

- **summarize()** - calculates statistics across row

```
1 data %>%
2   summarize(min_yfp = min(mean_yfp), n_samples = n(mean_yfp))
```

- **group_by()** - group select rows for calculating statistics

```
1 data %>%
2   group_by(strain) %>%
3   summarize(mean_yfp = mean(yfp),
4             mean_rfp = mean(rfp),
5             se_yfp = sd(yfp) / sqrt(n()),
6             se_rfp = sd(rfp) / sqrt(n())) %>%
7   print()
```

Pivoting tibbles to long and wide format

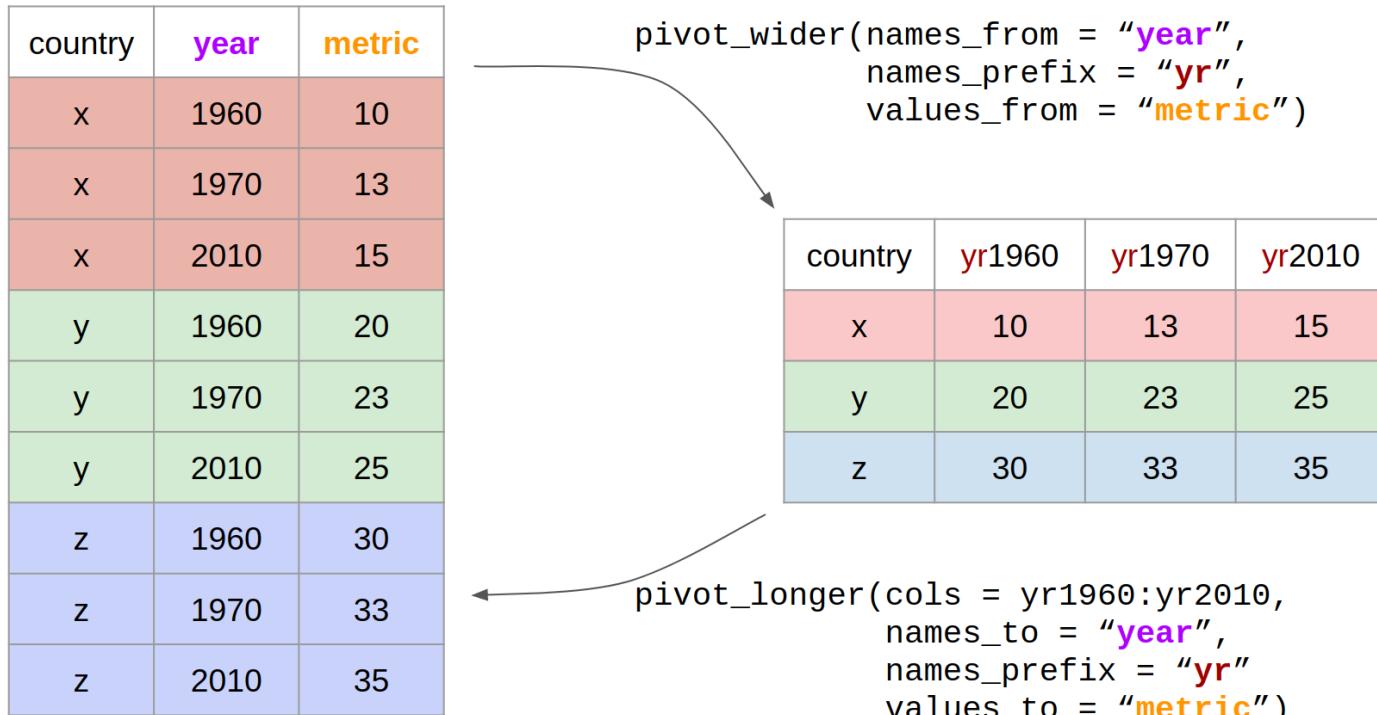


Image source: <https://tavareshugo.github.io/r-intro-tidyverse-gapminder/09-reshaping/index.html>

Working with multiple files

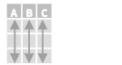
- Use `list.files` and `enframe` to go from a list of files to a `tibble` quickly.
- `map()` applies a function to all elements in the tibble

```
data <- list.files("data/channel_data/", full.names = T) %>%
  enframe("number", "file") %>%
  mutate(data = map(file, read_tsv)) %>%
  unnest(data) %>%
  mutate(channel = str_extract(file, "[^/]+(?=\\.tsv)")) %>%
  select(-number, -file) %>%
  print()
```


Data transformation with dplyr :: CHEATSHEET



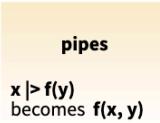
dplyr functions work with pipes and expect **tidy data**. In tidy data:



Each **variable** is in its own **column**



Each **observation**, or **case**, is in its own **row**



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.



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.



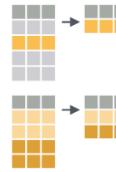
ungroup(x, ...) Returns ungrouped copy of table.

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



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

