Lecture 12: Data analysis using R and `tidyverse`

Matthew Chan

Postdoctoral Fellow, Fred Hutch
mchan3@fredhutch.org

MCB 536: Tools for Computational Biology
5 November 2024

Image source: Allison Horst

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

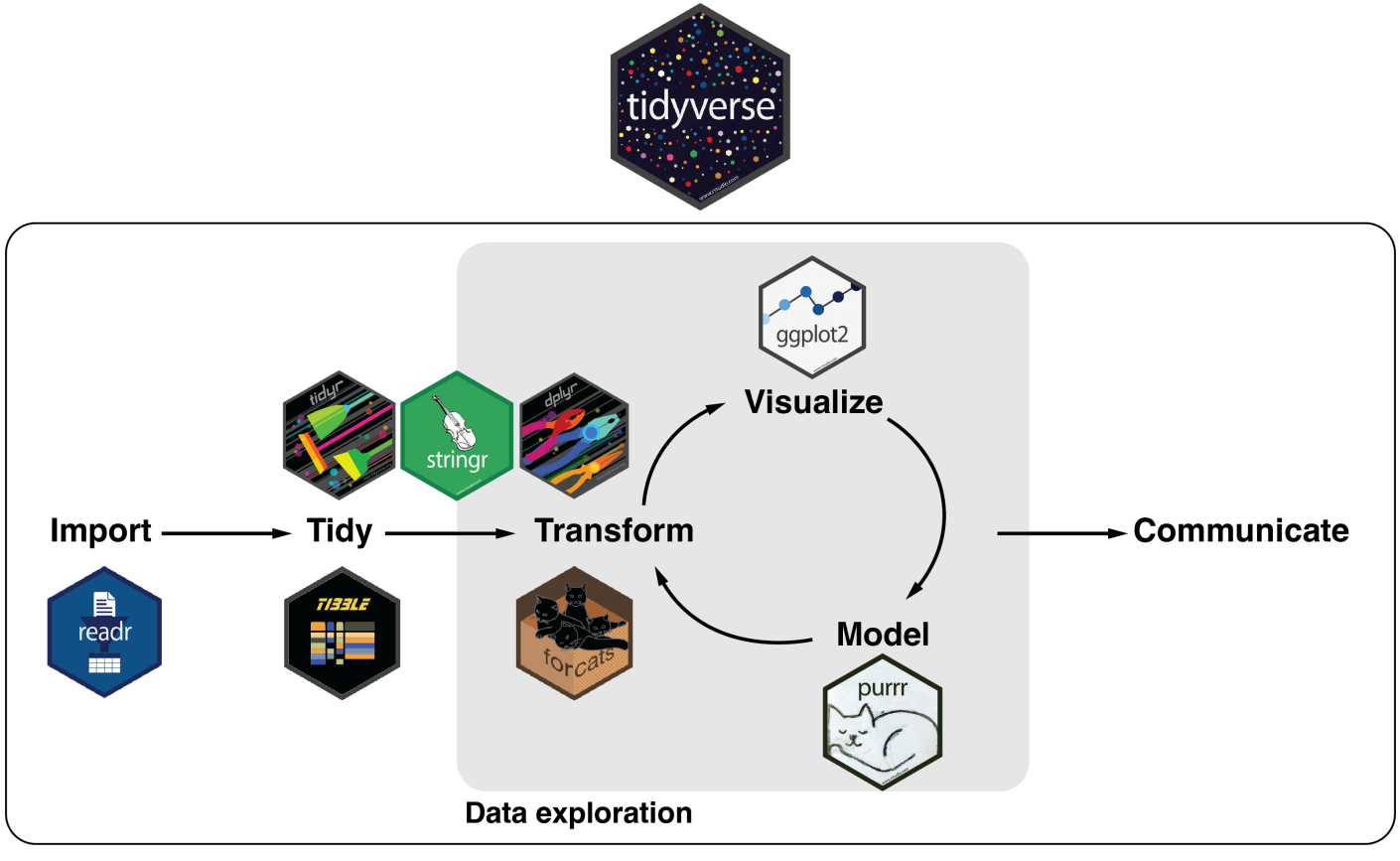


Figure adapted from R for Data Science, 2017

Loading packages

- In python:

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

- In R:

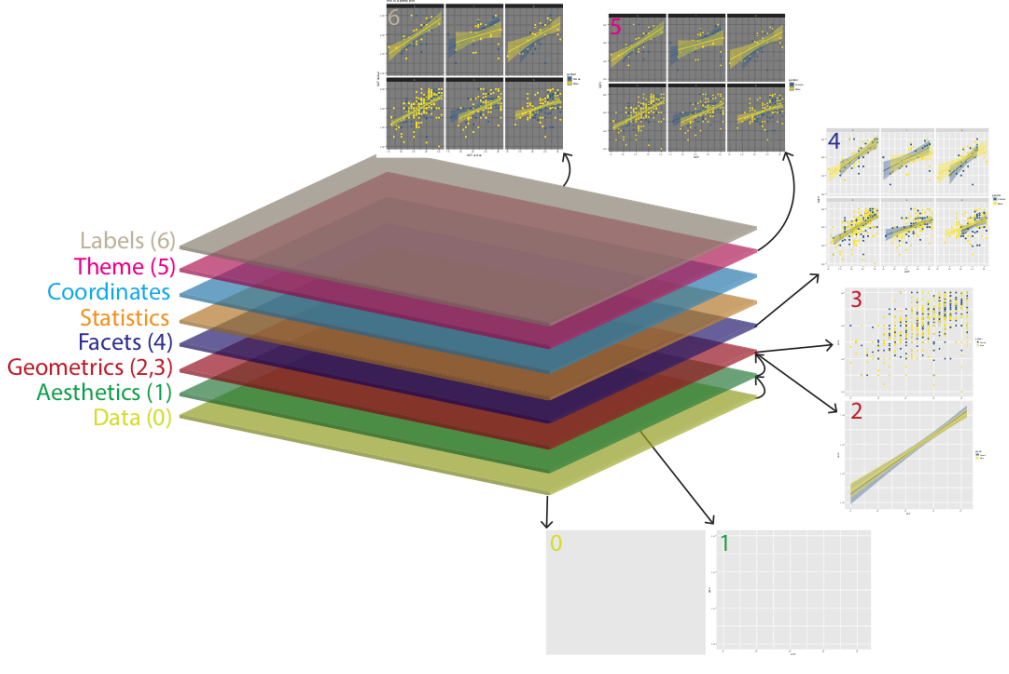
```r
library(tidyverse)
```

Plotting data: The grammer 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`

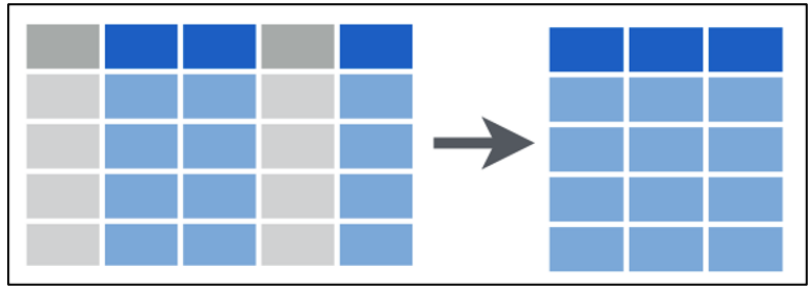
- ggplot2 documentation
- R graphics cookbook - examples and resources on all things plots in R.
- R graph gallery - Examples of different plot types and additional features/layers to add.

Select columns with `select()` function

`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, ...)
```



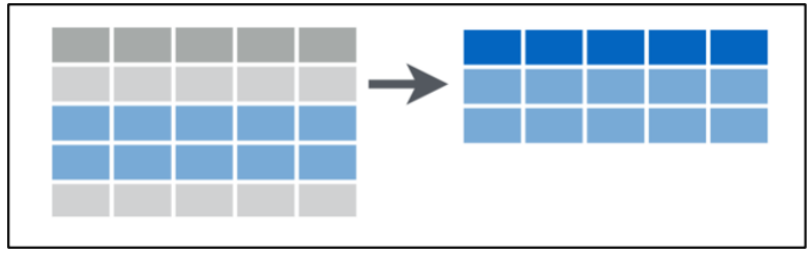
Example: Keep `mean_yfp` and `mean_rfp` columns:

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

Keep rows with `filter()` function

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`

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples < oranges
```

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples < oranges
```

```
FALSE
```

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples < oranges + kiwis
```

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples < oranges + kiwis
```

```
TRUE
```

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples < ((oranges + kiwis) && apples)
```

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples < ((oranges + kiwis) && apples)
```

```
FALSE
```

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples > oranges || kiwis
```

Comparison and logical operators

Suppose:

```
apples <- 10
oranges <- 7
kiwis <- 8
```

What is the value of this expression? (TRUE or FALSE)

```
apples > oranges || kiwis
```

```
TRUE
```

%>% 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):

```
data <- read.tsv("data/example_dataset_1.tsv")

selected_columns <- select(data, strain, mean_ratio,
                           insert_sequence, kozak_region)

tidy_dataframe <- filter(selected_columns, kozak_region == "A")
```

But this is better:

```
tidy_dataframe <- data %>%
    select(., strain, mean_ratio, insert_sequence, kozak_region) %>%
    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):

```
data <- read.tsv("data/example_dataset_1.tsv")

tidy_dataframe <- filter(select(data, strain,
    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):

```
data <- read.tsv("data/example_dataset_1.tsv")

tidy_dataframe <- filter(select(data, strain,
    mean_ratio,insert_sequence, kozak_region), kozak_region == "A")
```

But this is so much better:

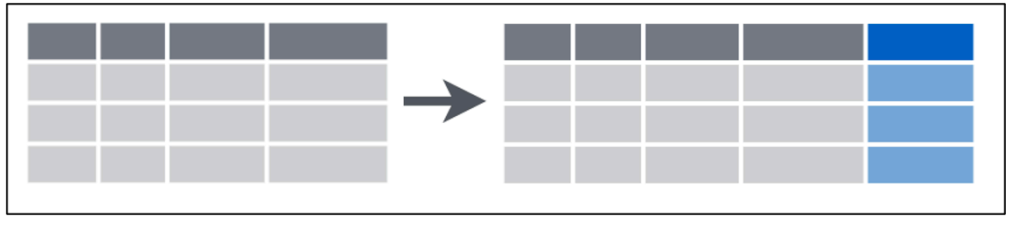
```
tidy_dataframe <- data %>%
    select(strain, mean_ratio, insert_sequence, kozak_region) %>%
    filter(kozak_region == "A")
```

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.

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`

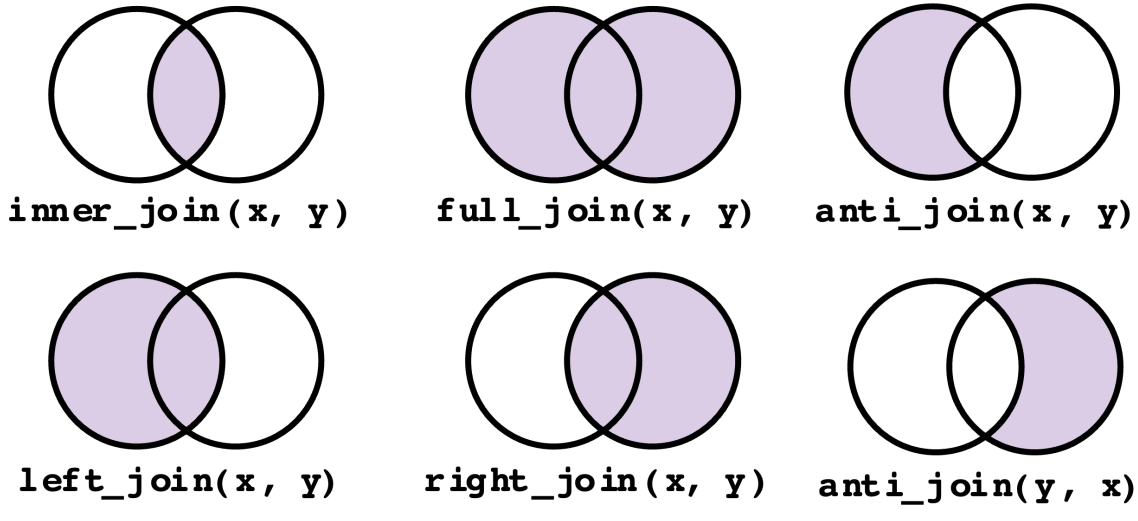
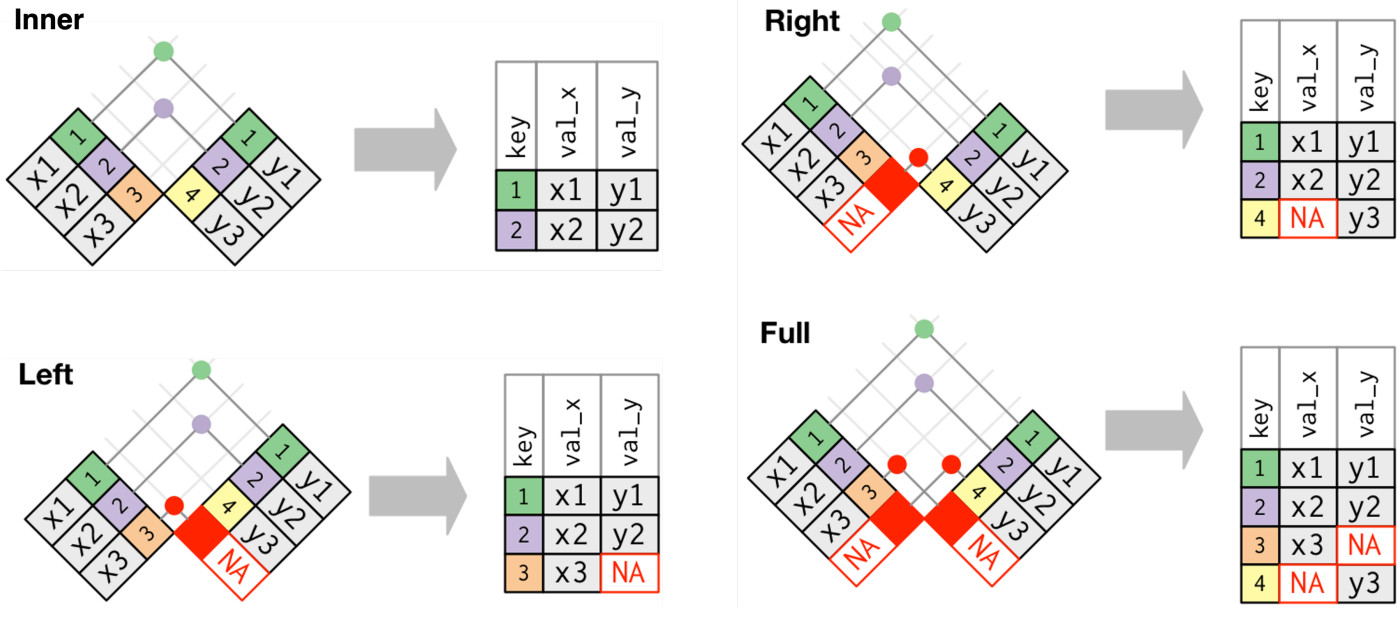


Image credit: https://tavareshugo.github.io/r-intro-tidyverse-gapminder/08-joins/index.html

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

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.

MATCH CHARACTERS

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

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:][1]	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

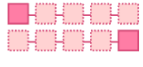
[:upper:]
A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z

ALTERNATES

alt <- function(rx) str_view("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

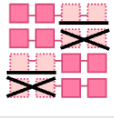
ANCHORS



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

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

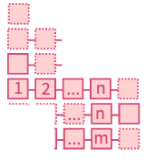
LOOK AROUNDS



look <- function(rx) str_view("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(".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("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

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

- `summarize()` - calculates statistics across row

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

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

```
data %>%
  group_by(strain) %>%
  summarize(mean_yfp = mean(yfp),
            mean_rfp = mean(rfp),
            se_yfp = sd(yfp) / sqrt(n()),
            se_rfp = sd(rfp) / sqrt(n())) %>%
  print()
```

Pivoting tibbles to long and wide format

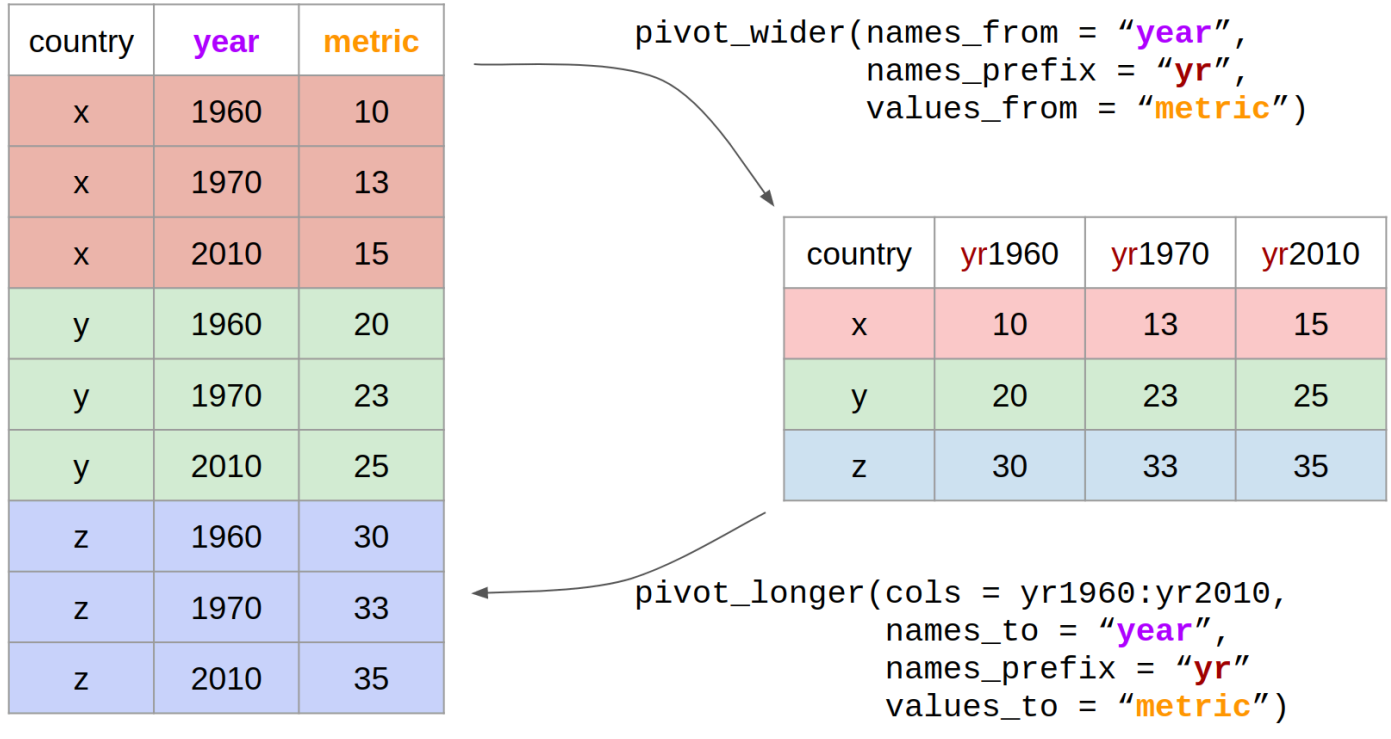


Image source: https://tavareshugo.github.io/r-intro-tidyverse-gapminder/09-reshaping/index.html

Working with multiple files

- Use `list.files` and `enframe` to go from a list of files to a `tibble` quickly.
- `map()` applys 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**

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))
```

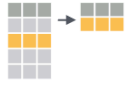
ungroup(x, …**)** Returns ungrouped copy of table.

```
g_mtcars <- mtcars |> group_by(cyl)
ungroup(g_mtcars)
```

Manipulate Cases

EXTRACT CASES

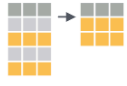
Row functions return a subset of rows as a new table.



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

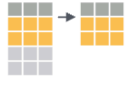
distinct(.data, …, .keep_all = FALSE**)** Remove rows with duplicate values.
mtcars |> distinct(gear)

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)



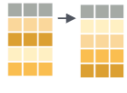
slice_head(.data, …, n, prop**)** and **slice_tail()** Select the first or last rows.
mtcars |> slice_head(n = 5)

Logical and boolean operators to use with filter()

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

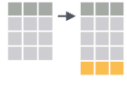
See **?base::Logic** and **?Comparison** for help.

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)

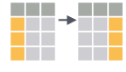
Manipulate Variables

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)



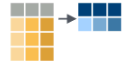
relocate(.data, …, .before = NULL, .after = NULL**)** Move columns to new position.
mtcars |> relocate(mpg, cyl, .after = last_col())

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))

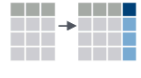
c_across(.cols**)** Compute across columns in row-wise data.

```
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)