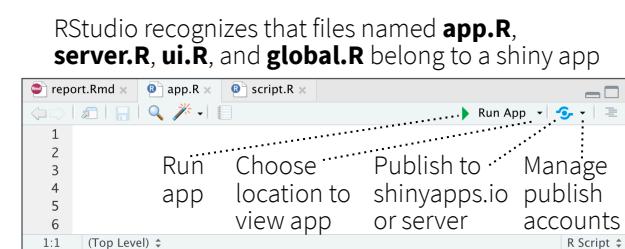
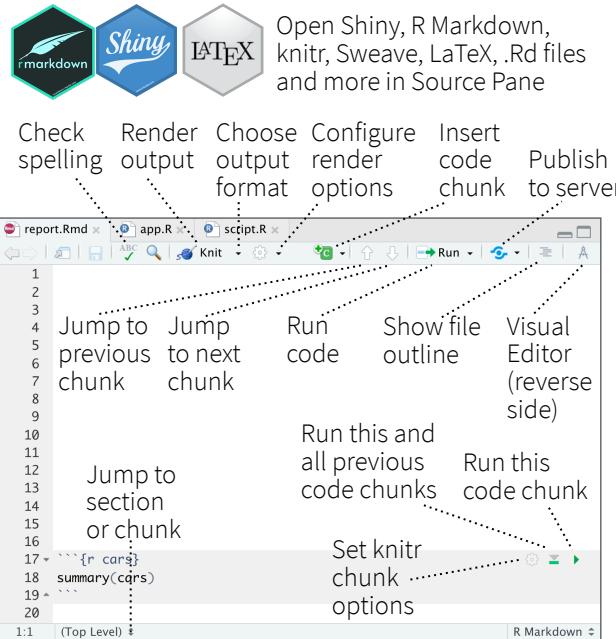


RStudio IDE :: CHEAT SHEET

Documents and Apps

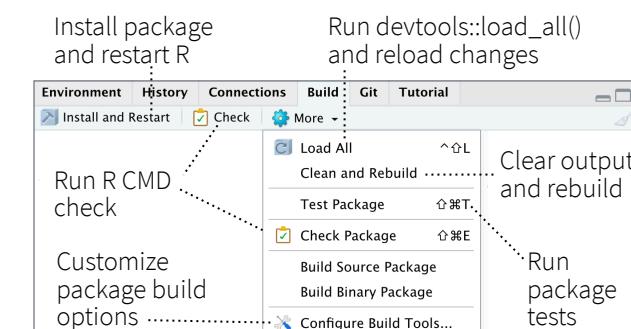


Package Development

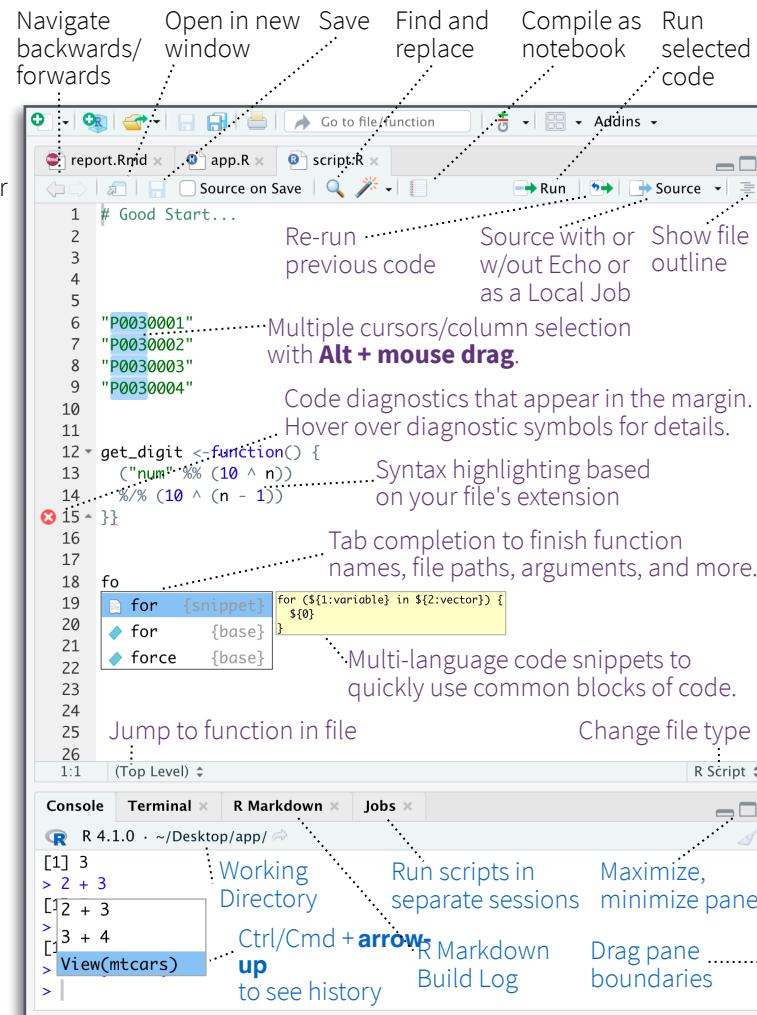
Create a new package with **File > New Project > New Directory > R Package**
Enable roxygen documentation with **Tools > Project Options > Build Tools**

Roxygen guide at **Help > Roxygen Quick Reference**

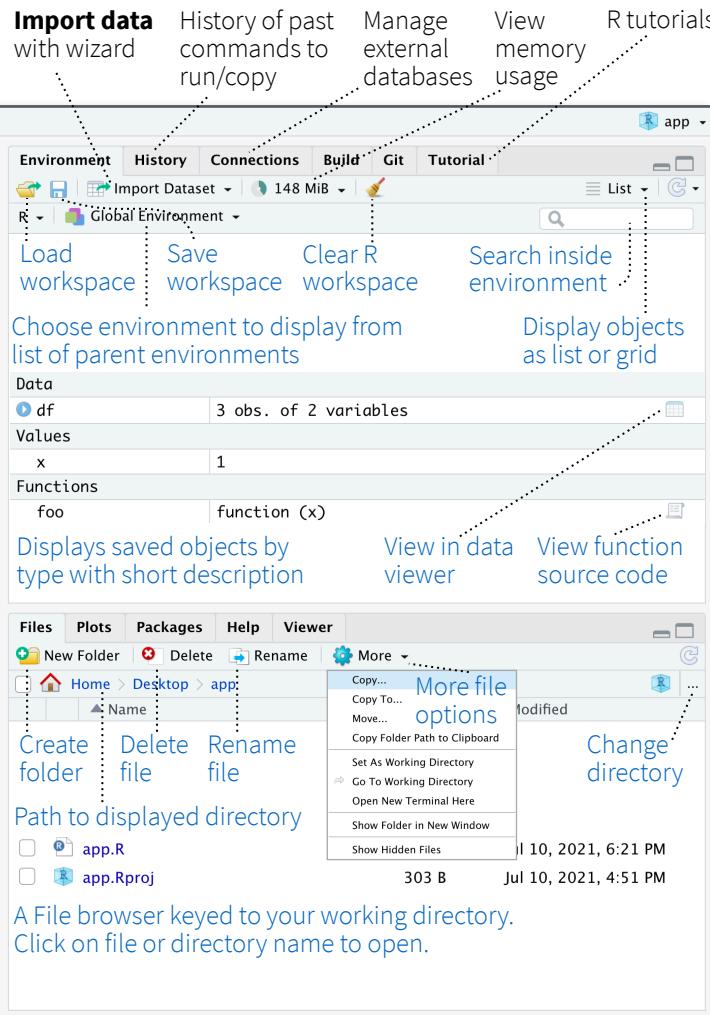
See package information in the **Build Tab**



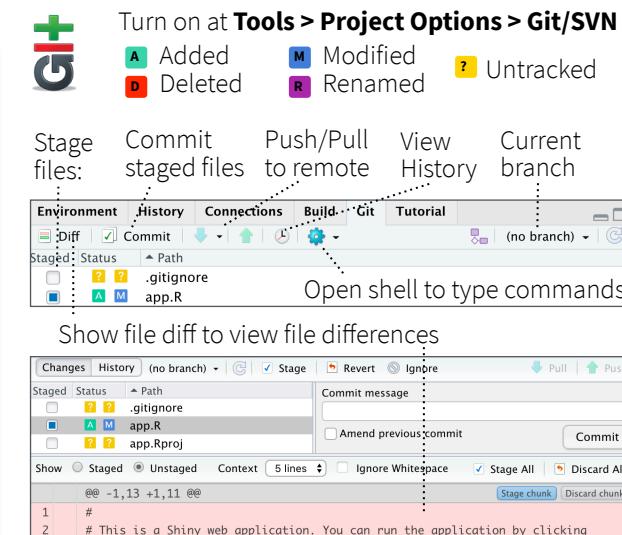
Source Editor



Tab Panes

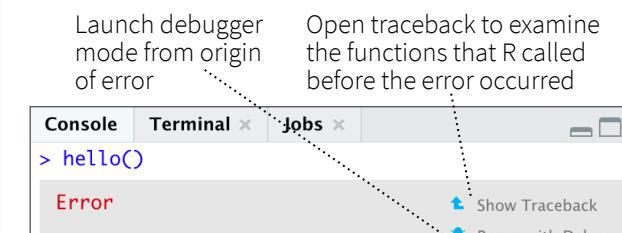


Version Control



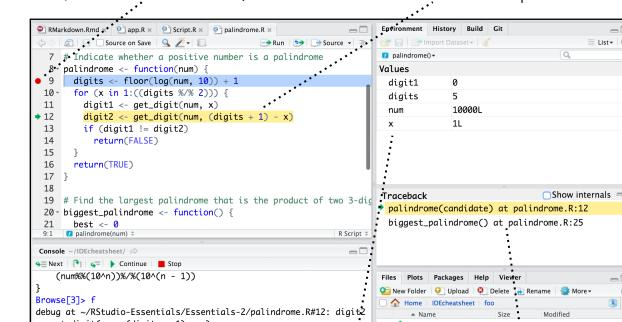
Debug Mode

Use **debug()**, **browser()**, or a breakpoint and execute your code to open the debugger mode.



Click next to line number to add/remove a breakpoint.

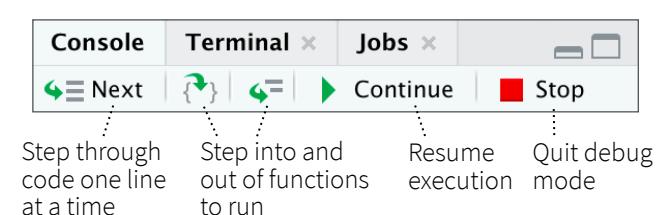
Highlighted line shows where execution has paused



Run commands in environment where execution has paused

Examine variables in executing environment

Select function in traceback to debug





Keyboard Shortcuts

RUN CODE

Search command history
Interrupt current command
Clear console

	Windows/Linux	Mac
Search command history	Ctrl+arrow-up	Cmd+arrow-up
Interrupt current command	Esc	Esc
Clear console	Ctrl+L	Ctrl+L

NAVIGATE CODE

Go to File/Function

	Windows/Linux	Mac
Go to File/Function	Ctrl+.	Ctrl+.

WRITE CODE

Attempt completion

	Windows/Linux	Mac
Insert <- (assignment operator)	Alt+-	Option+-
Insert %>% (pipe operator)	Ctrl+Shift+M	Cmd+Shift+M
(Un)Comment selection	Ctrl+Shift+C	Cmd+Shift+C

MAKE PACKAGES

	Windows/Linux	Mac
Load All (devtools)	Ctrl+Shift+L	Cmd+Shift+L
Test Package (Desktop)	Ctrl+Shift+T	Cmd+Shift+T
Document Package	Ctrl+Shift+D	Cmd+Shift+D

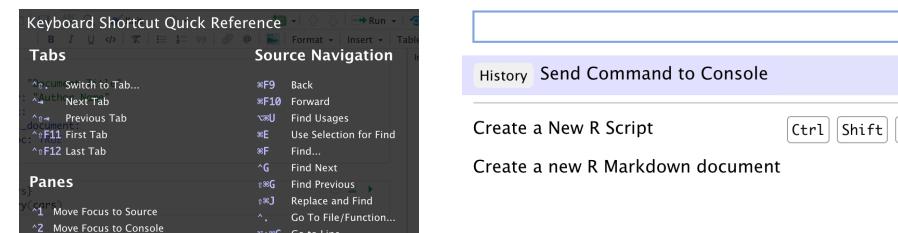
DOCUMENTS AND APPS

Knit Document (knitr)	Ctrl+Shift+K	Cmd+Shift+K
Insert chunk (Sweave & Knitr)	Ctrl+Alt+I	Cmd+Option+I
Run from start to current line	Ctrl+Alt+B	Cmd+Option+B

MORE KEYBOARD SHORTCUTS

Keyboard Shortcuts Help	Alt+Shift+K	Option+Shift+K
Show Command Palette	Ctrl+Shift+P	Cmd+Shift+P

View the Keyboard Shortcut Quick Reference with **Tools > Keyboard Shortcuts** or **Alt/Option + Shift + K**.



Search for keyboard shortcuts with **Tools > Show Command Palette** or **Ctrl/Cmd + Shift + P**.

RStudio Workbench

WHY RSTUDIO WORKBENCH?

Extend the open source server with a commercial license, support, and more:

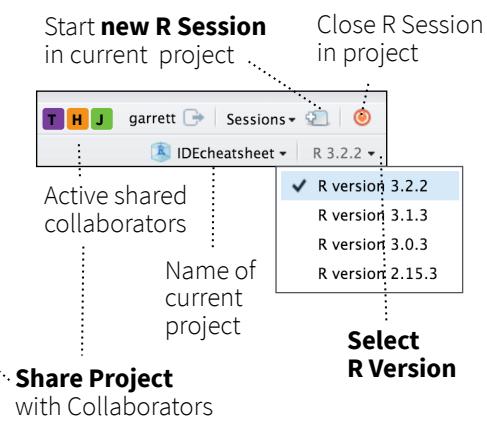
- open and run multiple R sessions at once
- tune your resources to improve performance
- administrative tools for managing user sessions
- collaborate real-time with others in shared projects
- switch easily from one version of R to a different version
- integrate with your authentication, authorization, and audit practices
- work in the RStudio IDE, JupyterLab, Jupyter Notebooks, or VS Code

Download a free 45 day evaluation at www.rstudio.com/products/workbench/evaluation/

Share Projects

File > New Project

RStudio saves the call history, workspace, and working directory associated with a project. It reloads each when you re-open a project.



Visual Editor

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.

```
{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.

```
{r cars}
summary(cars)
```

Jump to chunk or header

Block format

Check spelling

Render output

Choose output format

Choose output location

Insert code chunk

Jump to previous chunk

Jump to next chunk

Run selected lines

Publish to server

Show file outline

Back to Source Editor (front page)

File outline

Lists and block quotes

Links

Citations

Images

More formatting

Insert blocks, citations, equations, and special characters

Insert and edit tables

Set knitr chunk options

Run this and all previous code chunks

Run this code chunk

Add/Edit attributes

Run Remote Jobs

Run R on remote clusters (Kubernetes/Slurm) via the Job Launcher

Monitor launcher jobs

Launch a job

Run launcher jobs remotely

Run this and all previous code chunks

Run this code chunk

Set knitr chunk options

Add/Edit attributes

Run this and all previous code chunks

Run this code chunk

Set knitr chunk options

Run launcher jobs remotely



Data import with the tidyverse :: CHEAT SHEET

Read Tabular Data with readr

```
read_*(file, col_names = TRUE, col_types = NULL, col_select = NULL, id = NULL, locale, n_max = Inf,
      skip = 0, na = c("", "NA"), guess_max = min(1000, n_max), show_col_types = TRUE) See ?read_delim
```

A B C 1 2 3 4 5 NA	→	A B C 1 2 3 4 NA
		read_delim("file.txt", delim = " ") Read files with any delimiter. If no delimiter is specified, it will automatically guess. To make file.txt, run: write_file("A B C\n1 2 3\n4 5 NA", file = "file.txt")

A,B,C 1,2,3 4,5,NA	→	A B C 1 2 3 4 NA
		read_csv("file.csv") Read a comma delimited file with period decimal marks. write_file("A,B,C\n1,2,3\n4,5,NA", file = "file.csv")

A;B;C 1,5;2;3 4,5;5;NA	→	A B C 1.5 2 3 4.5 NA
		read_csv2("file2.csv") Read semicolon delimited files with comma decimal marks. write_file("A;B;C\n1,5;2;3\n4,5;5;NA", file = "file2.csv")

A B C 1 2 3 4 5 NA	→	A B C 1 2 3 4 NA
		read_tsv("file.tsv") Read a tab delimited file. Also read_table() . read_fwf("file.tsv", fwf_widths(c(2, 2, NA))) Read a fixed width file. write_file("A\tB\tC\n1\t2\t3\n4\t5\tNA", file = "file.tsv")

USEFUL READ ARGUMENTS

No header
read_csv("file.csv", col_names = FALSE)

Provide header
read_csv("file.csv", col_names = c("x", "y", "z"))

Read multiple files into a single table
read_csv(c("f1.csv", "f2.csv", "f3.csv"), id = "origin_file")

Skip lines
read_csv("file.csv", skip = 1)

Read a subset of lines
read_csv("file.csv", n_max = 1)

Read values as missing
read_csv("file.csv", na = c("1"))

Specify decimal marks
read_delim("file2.csv", locale = locale(decimal_mark = ","))

A;B;C 1,5;2;3
NA 2 3 4 5 NA

One of the first steps of a project is to import outside data into R. Data is often stored in tabular formats, like csv files or spreadsheets.



The front page of this sheet shows how to import and save text files into R using **readr**.



The back page shows how to import spreadsheet data from Excel files using **readxl** or Google Sheets using **googlesheets4**.

OTHER TYPES OF DATA

Try one of the following packages to import other types of files:

- **haven** - SPSS, Stata, and SAS files
- **DBI** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)
- **readr::read_lines()** - text data

Column Specification with readr

Column specifications define what data type each column of a file will be imported as. By default **readr** will generate a column spec when a file is read and output a summary.

spec(x) Extract the full column specification for the given imported data frame.

```
spec(x)
# cols(
#   age = col_integer(),
#   sex = col_character(),
#   earn = col_double()
# )
```

age is an integer

sex is a character

earn is a double (numeric)

COLUMN TYPES

Each column type has a function and corresponding string abbreviation.

- **col_logical()** - "l"
- **col_integer()** - "i"
- **col_double()** - "d"
- **col_number()** - "n"
- **col_character()** - "c"
- **col_factor(levels, ordered = FALSE)** - "f"
- **col_datetime(format = "")** - "T"
- **col_date(format = "")** - "D"
- **col_time(format = "")** - "t"
- **col_skip() - "-"**, "_"
- **col_guess() - "?"**

USEFUL COLUMN ARGUMENTS

Hide col spec message

read_*(file, show_col_types = FALSE)

Select columns to import

Use names, position, or selection helpers.
read_*(file, col_select = c(age, earn))

Guess column types

To guess a column type, **read_***() looks at the first 1000 rows of data. Increase with **guess_max**.
read_*(file, guess_max = Inf)

DEFINE COLUMN SPECIFICATION

Set a default type

```
read_csv(
  file,
  col_type = list(.default = col_double())
)
```

Use column type or string abbreviation

```
read_csv(
  file,
  col_type = list(x = col_double(), y = "l", z = "_")
)
```

Use a single string of abbreviations

```
# col types: skip, guess, integer, logical, character
read_csv(
  file,
  col_type = "_?ilc"
)
```

Save Data with readr

```
write_*(x, file, na = "NA", append, col_names, quote, escape, eol, num_threads, progress)
```

A B C 1 2 3 4 5 NA	→	A,B,C 1,2,3 4,5,NA
		write_delim(x, file, delim = " ") Write files with any delimiter. write_csv(x, file) Write a comma delimited file. write_csv2(x, file) Write a semicolon delimited file. write_tsv(x, file) Write a tab delimited file.

Import Spreadsheets with readxl

READ EXCEL FILES

A	B	C	D	E
1	x1	x2	x3	x4
2	x		z	8
3	y	7		9 10

```
read_excel(path, sheet = NULL, range = NULL)  
Read a .xls or .xlsx file based on the file extension.  
See front page for more read arguments. Also  
read_xls() and read_xlsx().  
read_excel("excel_file.xlsx")
```

READ SHEETS

A	B	C	D	E
s1	s2	s3		

s1	s2	s3
----	----	----

A	B	C	D	E
A	B	C	D	E
s1	s2	s3		

```
path <- "your_file_path.xlsx"  
path %>% excel_sheets() %>%  
  set_names() %>%  
  map_dfr(read_excel, path = path)
```

OTHER USEFUL EXCEL PACKAGES

For functions to write data to Excel files, see:

- **openxlsx**
- **writexl**

For working with non-tabular Excel data, see:

- **tidyxl**



READXL COLUMN SPECIFICATION

Column specifications define what data type each column of a file will be imported as.

Use the **col_types** argument of **read_excel()** to set the column specification.

Guess column types

To guess a column type, **read_excel()** looks at the first 1000 rows of data. Increase with the **guess_max** argument.

```
read_excel(path, guess_max = Inf)
```

Set all columns to same type, e.g. character

```
read_excel(path, col_types = "text")
```

Set each column individually

```
read_excel(  
  path,  
  col_types = c("text", "guess", "guess", "numeric"))
```

COLUMN TYPES

logical	numeric	text	date	list
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip
- guess
- logical
- numeric
- date
- list
- text

Use **list** for columns that include multiple data types. See **tidyxl** and **purrr** for list-column data.

CELL SPECIFICATION FOR READXL AND GOOGLESHEETS4

A	B	C	D	E
1	1	2	3	4
2	x		y	z
3	6	7		9 10

2	3	4
NA	y	z

Use the **range** argument of **readxl::read_excel()** or **googlesheets4::read_sheet()** to read a subset of cells from a sheet.

```
read_excel(path, range = "Sheet1!B1:D2")  
read_sheet(ss, range = "B1:D2")
```

Also use the range argument with cell specification functions **cell_limits()**, **cell_rows()**, **cell_cols()**, and **anchored()**.



with googlesheets4

READ SHEETS

A	B	C	D	E
1	x1	x2	x3	x4
2	x		z	8
3	y	7		9 10

x1	x2	x3	x4	x5
x		z	8	NA
y	7	NA	9	10

read_sheet(ss, sheet = NULL, range = NULL)
Read a sheet from a URL, a Sheet ID, or a dribble from the googledrive package. See front page for more read arguments. Same as **range_read()**.

SHEETS METADATA

URLs are in the form:

<https://docs.google.com/spreadsheets/d/>
SPREADSHEET_ID/edit#gid=**SHEET_ID**

gs4_get(ss) Get spreadsheet meta data.

gs4_find(...) Get data on all spreadsheet files.

sheet_properties(ss) Get a tibble of properties for each worksheet. Also **sheet_names()**.

WRITE SHEETS

1	x	4
2	y	5
3	z	6

1	A	B	C	D
2				

x1	x2	x3
2	y	5
3	z	6

write_sheet(data, ss = NULL, sheet = NULL)
Write a data frame into a new or existing Sheet.

gs4_create(name, ..., sheets = NULL) Create a new Sheet with a vector of names, a data frame, or a (named) list of data frames.

sheet_append(ss, data, sheet = 1) Add rows to the end of a worksheet.

GOOGLESHEETS4 COLUMN SPECIFICATION

Column specifications define what data type each column of a file will be imported as.

Use the **col_types** argument of **read_sheet()**/ **range_read()** to set the column specification.

Guess column types

To guess a column type **read_sheet()** looks at the first 1000 rows of data. Increase with the **guess_max** argument.

```
read_sheet(path, guess_max = Inf)
```

Set all columns to same type, e.g. character

```
read_sheet(path, col_types = "c")
```

Set each column individually

```
# col types: skip, guess, integer, logical, character  
read_sheets(ss, col_types = "?ilc")
```

COLUMN TYPES

I	n	c	D	L
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip - "_" or "-"
- guess - "?"
- logical - "l"
- integer - "i"
- double - "d"
- numeric - "n"
- date - "D"
- datetime - "T"
- character - "c"
- list-column - "L"
- cell - "C" Returns list of raw cell data.

Use list for columns that include multiple data types. See **tidyxl** and **purrr** for list-column data.

FILE LEVEL OPERATIONS

googlesheets4 also offers ways to modify other aspects of Sheets (e.g. freeze rows, set column width, manage (work)sheets). Go to googlesheets4.tidyverse.org to read more.

For whole-file operations (e.g. renaming, sharing, placing within a folder), see the tidyverse package **googledrive** at googledrive.tidyverse.org.

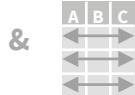
Data transformation with dplyr :: CHEAT SHEET



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).



`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

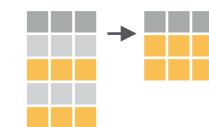
Row functions return a subset of rows as a new table.



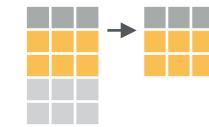
`filter(.data, ..., .preserve = FALSE)` Extract rows that meet logical criteria.
`filter(mtcars, mpg > 20)`



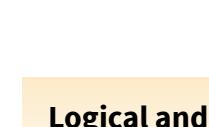
`distinct(.data, ..., .keep_all = FALSE)` Remove rows with duplicate values.
`distinct(mtcars, gear)`



`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>	

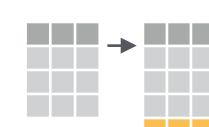
See `?base::Logic` and `?Comparison` for help.

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)`

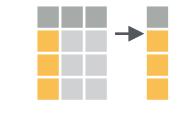
Manipulate Variables

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)`

`ends_with(match)`

`starts_with(match)`

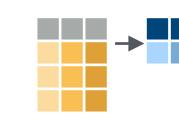
`num_range(prefix, range)`

`all_of(x)/any_of(x, ..., vars)`

- e.g. `-gear`

`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)))`

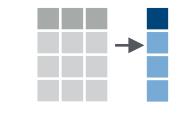
MAKE NEW VARIABLES

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()
 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 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-NAs

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.

A B C → C A B
1 a t → 1 a t
2 b u → 2 b u
3 c v → 3 c v
tibble::rownames_to_column()
Move row names into col.
a <- rownames_to_column(mtcars,
var = "C")

A B C → A B
1 a t → 1 a
2 b u → 2 b
3 c v → 3 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
A B C	E F G
a t 1	a t 3
b u 2	b u 2
c v 3	d w 1

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.

A B C D	left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join matching values from y to x.
a t 1 3 b u 2 2 c v 3 NA	

A B C D	right_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join matching values from x to y.
a t 1 3 b u 2 2 d w NA 1	

A B C D	inner_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join data. Retain only rows with matches.
a t 1 3 b u 2 2	

A B C D	full_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ..., keep = FALSE, na_matches = "na") Join data. Retain all values, all rows.
a t 1 3 b u 2 2 c v 3 NA d w NA 1	

COLUMN MATCHING FOR JOINS

A B x C B y D
a t 1 t 3
b u 2 u 2
c v 3 NA NA

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

A x B x C A y B y
a t 1 d w
b u 2 b u
c v 3 a t

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"))

A1 B1 C A2 B2
a t 1 d w
b u 2 b u
c v 3 a t

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
A B C	A B C
a t 1	a t 1
b u 2	b u 2
c v 3	d w 4

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).

X	y
A B C	A B C
a t 1	a t 1
b u 2	b u 2
c v 3	y d w 4

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.

A B C	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.

A B C	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

A B C
c v 3

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

A B C
a t 1

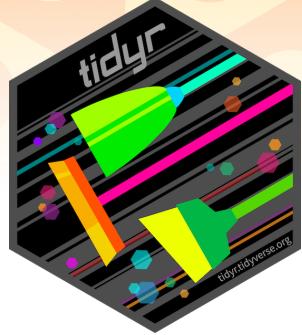
setdiff(x, y, ...)
Rows that appear in x but not y.

A B C
a t 1
b u 2

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

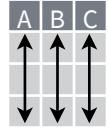
Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

Data tidying with `tidyr` :: CHEAT SHEET

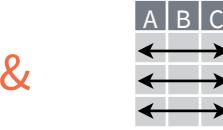


Tidy data is a way to organize tabular data in a consistent data structure across packages.

A table is tidy if:



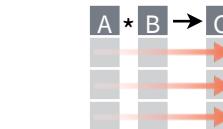
Each **variable** is in its own **column**



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



Access **variables** as **vectors**



Preserve **cases** in vectorized operations

Tibbles

AN ENHANCED DATA FRAME

Tibbles are a table format provided by the **tibble** package. They inherit the data frame class, but have improved behaviors:

- **Subset** a new tibble with `[`, a vector with `[[` and `$`.
- **No partial matching** when subsetting columns.
- **Display** concise views of the data on one screen.

`options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf)` Control default display settings.

`View()` or `glimpse()` View the entire data set.

CONSTRUCT A TIBBLE

tibble(...) Construct by columns.

`tibble(x = 1:3, y = c("a", "b", "c"))`

Both make this tibble

A tibble: 3 × 2
`x` <int> <chr>
 1 1 a
 2 2 b
 3 3 c

as_tibble(x, ...) Convert a data frame to a tibble.

enframe(x, name = "name", value = "value")

Convert a named vector to a tibble. Also `deframe()`.

is_tibble(x) Test whether x is a tibble.



Reshape Data

- Pivot data to reorganize values into a new layout.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K



country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T



country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

Split Cells

- Use these functions to split or combine cells into individual, isolated values.

table5

country	century	year
A	19	99
A	20	00
B	19	99
B	20	00



country	year
A	1999
A	2000
B	1999
B	2000

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M



country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M

table3

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M

pivot_longer(data, cols, names_to = "name", values_to = "value", values_drop_na = FALSE)

"Lengthen" data by collapsing several columns into two. Column names move to a new names_to column and values to a new values_to column.

`pivot_longer(table4a, cols = 2:3, names_to = "year", values_to = "cases")`

pivot_wider(data, names_from = "name", values_from = "value")

The inverse of `pivot_longer()`. "Widen" data by expanding two columns into several. One column provides the new column names, the other the values.

`pivot_wider(table2, names_from = type, values_from = count)`

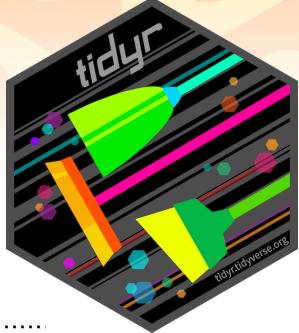
Expand Tables

Create new combinations of variables or identify implicit missing values (combinations of variables not present in the data).

x	x1	x2	x3
A	1	3	
B	1	4	
B	2	3	

expand(data, ...) Create a new tibble with all possible combinations of the values of the variables listed in ... Drop other variables.
`expand(mtcars, cyl, gear, carb)`

x	x1	x2	x3
A	1	3	
B	1	4	
B	2	3	
	</td		



Nested Data

A **nested data frame** stores individual tables as a list-column of data frames within a larger organizing data frame. List-columns can also be lists of vectors or lists of varying data types.

Use a nested data frame to:

- Preserve relationships between observations and subsets of data. Preserve the type of the variables being nested (factors and datetimes aren't coerced to character).
- Manipulate many sub-tables at once with **purrr** functions like `map()`, `map2()`, or `pmap()` or with **dplyr** `rowwise()` grouping.

CREATE NESTED DATA

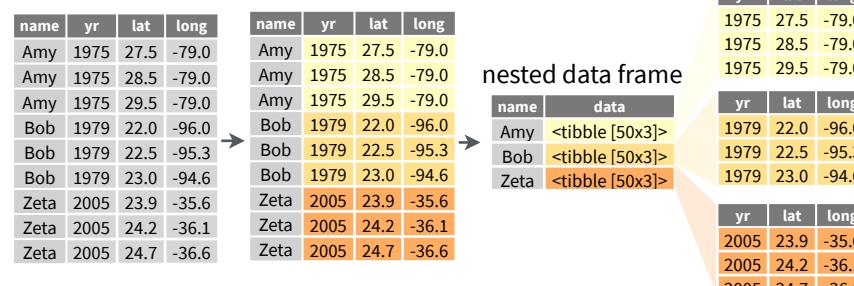
nest(data, ...) Moves groups of cells into a list-column of a data frame. Use alone or with `dplyr::group_by()`:

1. Group the data frame with `group_by()` and use `nest()` to move the groups into a list-column.

```
n_storms <- storms %>%
  group_by(name) %>%
  nest()
```

2. Use `nest(new_col = c(x, y))` to specify the columns to group using `dplyr::select()` syntax.

```
n_storms <- storms %>%
  nest(data = c(year:long))
```



Index list-columns with `[[[]]]`. `n_storms$data[[1]]`

CREATE TIBBLES WITH LIST-COLUMNS

tibble::tribble(...) Makes list-columns when needed.

```
tribble(~max, ~seq,
       3, 1:3,
       4, 1:4,
       5, 1:5)
```

max	seq
3	<int [3]>
4	<int [4]>
5	<int [5]>

tibble::tibble(...) Saves list input as list-columns.

```
tibble(max = c(3, 4, 5), seq = list(1:3, 1:4, 1:5))
```

tibble::enframe(x, name="name", value="value")

Converts multi-level list to a tibble with list-cols.
`enframe(list('3'=1:3, '4'=1:4, '5'=1:5), 'max', 'seq')`

OUTPUT LIST-COLUMNS FROM OTHER FUNCTIONS

dplyr::mutate(), transmute(), and summarise() will output list-columns if they return a list.

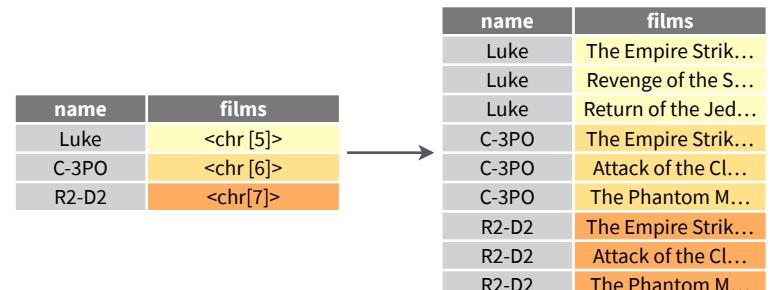
```
mtcars %>%
  group_by(cyl) %>%
  summarise(q = list(quantile(mpg)))
```

RESHAPE NESTED DATA

unnest(data, cols, ..., keep_empty = FALSE) Flatten nested columns back to regular columns. The inverse of `nest()`.
`n_storms %>% unnest(data)`

unnest_longer(data, col, values_to = NULL, indices_to = NULL)
Turn each element of a list-column into a row.

```
starwars %>%
  select(name, films) %>%
  unnest_longer(films)
```



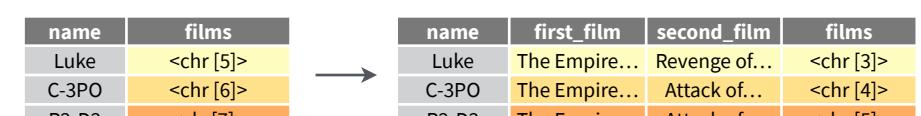
unnest_wider(data, col) Turn each element of a list-column into a regular column.

```
starwars %>%
  select(name, films) %>%
  unnest_wider(films)
```



hoist(.data, .col, ..., .remove = TRUE) Selectively pull list components out into their own top-level columns. Uses `purrr::pluck()` syntax for selecting from lists.

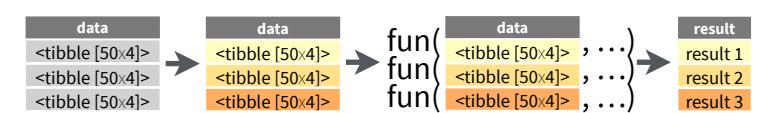
```
starwars %>%
  select(name, films) %>%
  hoist(films, first_film = 1, second_film = 2)
```



TRANSFORM NESTED DATA

A vectorized function takes a vector, transforms each element in parallel, and returns a vector of the same length. By themselves vectorized functions cannot work with lists, such as list-columns.

dplyr::rowwise(.data, ...) Group data so that each row is one group, and within the groups, elements of list-columns appear directly (accessed with `[]`, not as lists of length one. **When you use `rowwise()`, dplyr functions will seem to apply functions to list-columns in a vectorized fashion.**



Apply a function to a list-column and **create a new list-column**.

`n_storms %>%`
`rowwise() %>%`
`mutate(n = list(dim(data)))`

`dim()` returns two values per row

wrap with list to tell mutate to create a list-column

Apply a function to a list-column and **create a regular column**.

`n_storms %>%`
`rowwise() %>%`
`mutate(n = nrow(data))`

`nrow()` returns one integer per row

Collapse **multiple list-columns** into a single list-column.

`starwars %>%`
`rowwise() %>%`
`mutate(transport = list(append(vehicles, starships)))`

`append()` returns a list for each row, so col type must be list

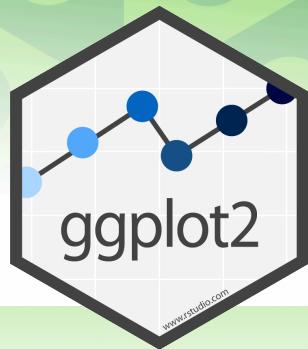
Apply a function to **multiple list-columns**.

`starwars %>%`
`rowwise() %>%`
`mutate(n_transports = length(c(vehicles, starships)))`

`length()` returns one integer per row

See **purrr** package for more list functions.

Data visualization with ggplot2 :: CHEAT SHEET



Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**—visual marks that represent data points.



To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.



Complete the template below to build a graph.

```
ggplot (data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
  stat = <STAT>, position = <POSITION>) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION> +
  <SCALE_FUNCTION> +
  <THEME_FUNCTION>
```

required
Not required, sensible defaults supplied

`ggplot(data = mpg, aes(x = cty, y = hwy))` Begins a plot that you finish by adding layers to. Add one geom function per layer.

`last_plot()` Returns the last plot.

`ggsave("plot.png", width = 5, height = 5)` Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Aes Common aesthetic values.

color and **fill** - string ("red", "#RRGGBB")

linetype - integer or string (0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotdash", 5 = "longdash", 6 = "twodash")

lineend - string ("round", "butt", or "square")

linejoin - string ("round", "mitre", or "bevel")

size - integer (line width in mm)

shape - integer/shape name or a single character ("a")




Geoms

Use a geom function to represent data points, use the geom's aesthetic properties to represent variables.
Each function returns a layer.

GRAPHICAL PRIMITIVES

```
a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))
```

- a + geom_blank()** and **a + expand_limits()**
Ensure limits include values across all plots.
- b + geom_curve(aes(yend = lat + 1, xend = long + 1, curvature = 1))** - x, yend, alpha, angle, curvature, linetype, size
- a + geom_path(lineend = "butt", linejoin = "round", linemitre = 1)** - x, y, alpha, color, group, linetype, size
- a + geom_polygon(aes(alpha = 50))** - x, y, alpha, color, fill, group, subgroup, linetype, size
- b + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1))** - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size
- a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900))** - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

common aesthetics: x, y, alpha, color, linetype, size

- b + geom_abline(aes(intercept = 0, slope = 1))**
- b + geom_hline(aes(yintercept = lat))**
- b + geom_vline(aes(xintercept = long))**
- b + geom_segment(aes(yend = lat + 1, xend = long + 1))**
- b + geom_spoke(aes(angle = 1:1155, radius = 1))**

ONE VARIABLE continuous

- ```
c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)
```
- c + geom\_area(stat = "bin")** - x, y, alpha, color, fill, linetype, size
  - c + geom\_density(kernel = "gaussian")** - x, y, alpha, color, fill, group, linetype, size, weight
  - c + geom\_dotplot()** - x, y, alpha, color, fill
  - c + geom\_freqpoly()** - x, y, alpha, color, group, linetype, size
  - c + geom\_histogram(binwidth = 5)** - x, y, alpha, color, fill, linetype, size, weight
  - c2 + geom\_qq(aes(sample = hwy))** - x, y, alpha, color, fill, linetype, size, weight

### discrete

```
d <- ggplot(mpg, aes(fl))
```

- d + geom\_bar()** - x, alpha, color, fill, linetype, size, weight

### TWO VARIABLES both continuous

```
e <- ggplot(mpg, aes(cty, hwy))
```

- e + geom\_label(aes(label = cty), nudge\_x = 1, nudge\_y = 1)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust
- e + geom\_point()** - x, y, alpha, color, fill, shape, size, stroke
- e + geom\_quantile()** - x, y, alpha, color, group, linetype, size, weight
- e + geom\_rug(sides = "bl")** - x, y, alpha, color, linetype, size
- e + geom\_smooth(method = lm)** - x, y, alpha, color, fill, group, linetype, size, weight
- e + geom\_text(aes(label = cty), nudge\_x = 1, nudge\_y = 1)** - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

### one discrete, one continuous

```
f <- ggplot(mpg, aes(class, hwy))
```

- f + geom\_col()** - x, y, alpha, color, fill, group, linetype, size
- f + geom\_boxplot()** - x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight
- f + geom\_dotplot(binaxis = "y", stackdir = "center")** - x, y, alpha, color, fill, group
- f + geom\_violin(scale = "area")** - x, y, alpha, color, fill, group, linetype, size, weight

### both discrete

```
g <- ggplot(diamonds, aes(cut, color))
```

- g + geom\_count()** - x, y, alpha, color, fill, shape, size, stroke
- e + geom\_jitter(height = 2, width = 2)** - x, y, alpha, color, fill, shape, size

### THREE VARIABLES

```
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))
```

- l + geom\_contour(aes(z = z))** - x, y, z, alpha, color, group, linetype, size, weight
- l + geom\_contour\_filled(aes(fill = z))** - x, y, alpha, color, fill, group, linetype, size, subgroup
- l + geom\_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE)** - x, y, alpha, fill
- l + geom\_tile(aes(fill = z))** - x, y, alpha, color, fill, linetype, size, width

### continuous bivariate distribution

```
h <- ggplot(diamonds, aes(carat, price))
```

- h + geom\_bin2d(binwidth = c(0.25, 500))** - x, y, alpha, color, fill, linetype, size, weight
- h + geom\_density\_2d()** - x, y, alpha, color, group, linetype, size
- h + geom\_hex()** - x, y, alpha, color, fill, size

### continuous function

```
i <- ggplot(economics, aes(date, unemploy))
```

- i + geom\_area()** - x, y, alpha, color, fill, linetype, size
- i + geom\_line()** - x, y, alpha, color, group, linetype, size
- i + geom\_step(direction = "hv")** - x, y, alpha, color, group, linetype, size

### visualizing error

```
df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
```

- j + geom\_crossbar(fatten = 2)** - x, y, ymax, ymin, alpha, color, fill, group, linetype, size
- j + geom\_errorbar()** - x, ymax, ymin, alpha, color, group, linetype, size, width  
Also **geom\_errorbarh()**.
- j + geom\_linerange()** - x, ymin, ymax, alpha, color, group, linetype, size
- j + geom\_pointrange()** - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

### maps

```
data <- data.frame(murder = USArrests$Murder, state = tolower(rownames(USArrests)))
```

```
map <- map_data("state")
```

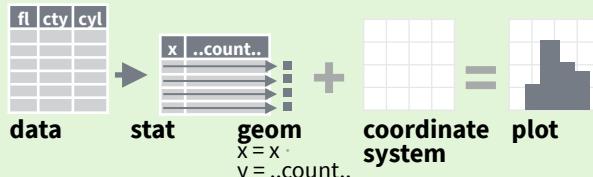
```
k <- ggplot(data, aes(fill = murder))
```

- k + geom\_map(aes(map\_id = state), map = map) + expand\_limits(x = map\$long, y = map\$lat)**  
map\_id, alpha, color, fill, linetype, size

# Stats

An alternative way to build a layer.

A stat builds new variables to plot (e.g., count, prop).



Visualize a stat by changing the default stat of a geom function, `geom_bar(stat="count")` or by using a stat function, `stat_count(geom="bar")`, which calls a default geom to make a layer (equivalent to a geom function). Use `..name..` syntax to map stat variables to aesthetics.



```

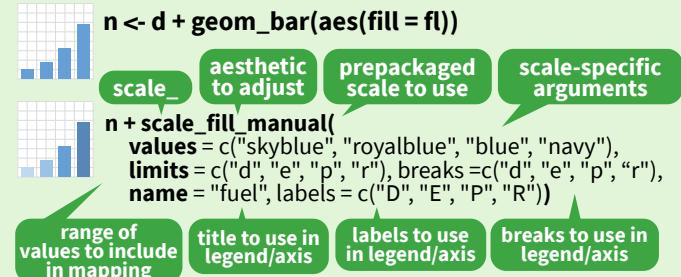
c + stat_bin(binwidth = 1, boundary = 10)
x, y | ..count.., ..ncount.., ..density.., ..ndensity..
c + stat_count(width = 1) x, y | ..count.., ..prop..
c + stat_density(adjust = 1, kernel = "gaussian")
x, y | ..count.., ..density.., ..scaled..
e + stat_bin_2d(bins = 30, drop = T)
x, y, fill | ..count.., ..density..
e + stat_bin_hex(bins = 30) x, y, fill | ..count.., ..density..
e + stat_density_2d(contour = TRUE, n = 100)
x, y, color, size | ..level..
e + stat_ellipse(level = 0.95, segments = 51, type = "t")
l + stat_contour(aes(z = z)) x, y, z, order | ..level..
l + stat_summary_hex(aes(z = z), bins = 30, fun = max)
x, y, z, fill | ..value..
l + stat_summary_2d(aes(z = z), bins = 30, fun = mean)
x, y, z, fill | ..value..
f + stat_boxplot(coef = 1.5)
x, y | ..lower.., ..middle.., ..upper.., ..width.., ..ymin.., ..ymax..
f + stat_ydensity(kernel = "gaussian", scale = "area") x, y
| ..density.., ..scaled.., ..count.., ..n.., ..violinwidth.., ..width..
e + stat_ecdf(n = 40) x, y | ..x.., ..y..
e + stat_quantile(quantiles = c(0.1, 0.9),
formula = y ~ log(x), method = "rq") x, y | ..quantile..
e + stat_smooth(method = "lm", formula = y ~ x, se = T,
level = 0.95) x, y | ..se.., ..x.., ..y.., ..ymin.., ..ymax..
ggplot() + xlim(-5, 5) + stat_function(fun = dnorm,
n = 20, geom = "point") x | ..x.., ..y..
ggplot() + stat_qq(aes(sample = 1:100))
x, y, sample | ..sample.., ..theoretical..
e + stat_sum() x, y, size | ..n.., ..prop..
e + stat_summary(fun.data = "mean_cl_boot")
h + stat_summary_bin(fun = "mean", geom = "bar")
e + stat_identity()
e + stat_unique()

```

# Scales

Override defaults with `scales` package.

**Scales** map data values to the visual values of an aesthetic. To change a mapping, add a new scale.



## GENERAL PURPOSE SCALES

Use with most aesthetics

`scale_*_continuous()` - Map cont' values to visual ones.  
`scale_*_discrete()` - Map discrete values to visual ones.  
`scale_*_binned()` - Map continuous values to discrete bins.  
`scale_*_identity()` - Use data values as visual ones.  
`scale_*_manual(values = c())` - Map discrete values to manually chosen visual ones.  
`scale_*_date(date_labels = "%m/%d")`,  
`date_breaks = "2 weeks"` - Treat data values as dates.  
`scale_*_datetime()` - Treat data values as date times.  
Same as `scale_*_date()`. See `?strptime` for label formats.

## X & Y LOCATION SCALES

Use with x or y aesthetics (x shown here)

`scale_x_log10()` - Plot x on log10 scale.  
`scale_x_reverse()` - Reverse the direction of the x axis.  
`scale_x_sqrt()` - Plot x on square root scale.

## COLOR AND FILL SCALES (DISCRETE)

`n + scale_fill_brewer(palette = "Blues")`  
For palette choices:  
RColorBrewer::display.brewer.all()  
`n + scale_fill_grey(start = 0.2,`  
`end = 0.8, na.value = "red")`

## COLOR AND FILL SCALES (CONTINUOUS)

`o <- c + geom_dotplot(aes(fill = ..x..))`  
`o + scale_fill_distiller(palette = "Blues")`  
`o + scale_fill_gradient(low = "red", high = "yellow")`  
`o + scale_fill_gradient2(low = "red", high = "blue",`  
`mid = "white", midpoint = 25)`  
`o + scale_fill_gradientn(colors = topo.colors(6))`  
Also: rainbow(), heat.colors(), terrain.colors(),  
cm.colors(), RColorBrewer::brewer.pal()

## SHAPE AND SIZE SCALES

`p <- e + geom_point(aes(shape = fl, size = cyl))`  
`p + scale_shape() + scale_size()`  
`p + scale_shape_manual(values = c(3:7))`  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25  
□○△×+×◊▽★◆⊕⊖田田田田田田田田田田田田田田田田田田田田田田田田  
`p + scale_radius(range = c(1,6))`  
`p + scale_size_area(max_size = 6)`

# Coordinate Systems

`r <- d + geom_bar()`

`r + coord_cartesian(xlim = c(0, 5))` - xlim, ylim  
The default cartesian coordinate system.

`r + coord_fixed(ratio = 1/2)`  
ratio, xlim, ylim - Cartesian coordinates with fixed aspect ratio between x and y units.

`ggplot(mpg, aes(y = fl)) + geom_bar()`  
Flip cartesian coordinates by switching x and y aesthetic mappings.

`r + coord_polar(theta = "x", direction=1)`  
theta, start, direction - Polar coordinates.

`r + coord_trans(y = "sqrt")` - x, y, xlim, ylim  
Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.

`π + coord_quickmap()`  
`π + coord_map(projection = "ortho", orientation = c(41, -74, 0))` - projection, xlim, ylim  
Map projections from the mapproj package (mercator (default), azequalarea, lagrange, etc.).

## Position Adjustments

Position adjustments determine how to arrange geoms that would otherwise occupy the same space.

`s <- ggplot(mpg, aes(fl, fill = drv))`

`s + geom_bar(position = "dodge")`  
Arrange elements side by side.

`s + geom_bar(position = "fill")`  
Stack elements on top of one another, normalize height.

`e + geom_point(position = "jitter")`  
Add random noise to X and Y position of each element to avoid overplotting.

`e + geom_label(position = "nudge")`  
Nudge labels away from points.

`s + geom_bar(position = "stack")`  
Stack elements on top of one another.

Each position adjustment can be recast as a function with manual `width` and `height` arguments:  
`s + geom_bar(position = position_dodge(width = 1))`

## Themes

`r + theme_bw()`  
White background with grid lines.

`r + theme_gray()`  
Grey background (default theme).

`r + theme_dark()`  
Dark for contrast.

`r + theme_classic()`  
r + theme\_light()

`r + theme_linedraw()`  
r + theme\_minimal()

`r + theme_void()`  
Empty theme.

`r + theme()` Customize aspects of the theme such as axis, legend, panel, and facet properties.  
`r + ggtitle("Title") + theme(plot.title.position = "plot")`  
`r + theme(panel.background = element_rect(fill = "blue"))`

# Faceting

Facets divide a plot into subplots based on the values of one or more discrete variables.

`t <- ggplot(mpg, aes(cty, hwy)) + geom_point()`

`t + facet_grid(cols = vars(fl))`  
Facet into columns based on fl.

`t + facet_grid(rows = vars(year))`  
Facet into rows based on year.

`t + facet_grid(rows = vars(year), cols = vars(fl))`  
Facet into both rows and columns.

`t + facet_wrap(vars(fl))`  
Wrap facets into a rectangular layout.

Set `scales` to let axis limits vary across facets.

`t + facet_grid(rows = vars(drv), cols = vars(fl), scales = "free")`

x and y axis limits adjust to individual facets:  
"free\_x" - x axis limits adjust  
"free\_y" - y axis limits adjust

Set `labeler` to adjust facet label:

`t + facet_grid(cols = vars(fl), labeler = label_both)`

fl: c fl: d fl: e fl: p fl: r

`t + facet_grid(rows = vars(fl), labeler = label_bquote(alpha ^ .(fl)))`

α<sup>c</sup> α<sup>d</sup> α<sup>e</sup> α<sup>p</sup> α<sup>r</sup>

# Labels and Legends

Use `labs()` to label the elements of your plot.

`t + labs(x = "New x axis label", y = "New y axis label", title = "Add a title above the plot", subtitle = "Add a subtitle below title", caption = "Add a caption below plot", alt = "Add alt text to the plot", <AES> = "New <AES> legend title")`

`t + annotate(geom = "text", x = 8, y = 9, label = "A")`  
Places a geom with manually selected aesthetics.

`p + guides(x = guide_axis(n.dodge = 2))` Avoid crowded or overlapping labels with `guide_axis(n.dodge` or `angle`).

`n + guides(fill = "none")` Set legend type for each aesthetic: colorbar, legend, or none (no legend).

`n + theme(legend.position = "bottom")`  
Place legend at "bottom", "top", "left", or "right".

`n + scale_fill_discrete(name = "Title", labels = c("A", "B", "C", "D", "E"))`  
Set legend title and labels with a scale function.

# Zooming

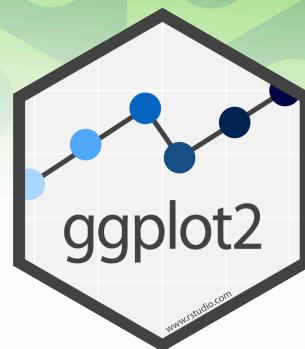
Without clipping (preferred):

`t + coord_cartesian(xlim = c(0, 100), ylim = c(10, 20))`

With clipping (removes unseen data points):

`t + xlim(0, 100) + ylim(10, 20)`

`t + scale_x_continuous(limits = c(0, 100)) + scale_y_continuous(limits = c(0, 100))`



# String manipulation with stringr :: CHEAT SHEET



The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

## Detect Matches

|                                            |                                                                                                                                                                            |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br>→<br><br>TRUE<br>TRUE<br>FALSE<br>TRUE | <b>str_detect(string, pattern, negate = FALSE)</b><br>Detect the presence of a pattern match in a string. Also <b>str_like()</b> . str_detect(fruit, "a")                  |
| <br>→<br><br>TRUE<br>TRUE<br>FALSE<br>TRUE | <b>str_starts(string, pattern, negate = FALSE)</b><br>Detect the presence of a pattern match at the beginning of a string. Also <b>str_ends()</b> . str_starts(fruit, "a") |
| <br>→<br><br>1<br>2<br>4                   | <b>str_which(string, pattern, negate = FALSE)</b><br>Find the indexes of strings that contain a pattern match. str_which(fruit, "a")                                       |
| <br>→<br><br>2 4<br>4 7<br>NA NA<br>3 4    | <b>str_locate(string, pattern)</b> Locate the positions of pattern matches in a string. Also <b>str_locate_all()</b> . str_locate(fruit, "a")                              |
| <br>→<br><br>0<br>3<br>1<br>2              | <b>str_count(string, pattern)</b> Count the number of matches in a string. str_count(fruit, "a")                                                                           |

## Subset Strings

|                                    |                                                                                                                                                                                                                      |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br>→<br><br>4<br>2<br>3           | <b>str_sub(string, start = 1L, end = -1L)</b> Extract substrings from a character vector. str_sub(fruit, 1, 3); str_sub(fruit, -2)                                                                                   |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_subset(string, pattern, negate = FALSE)</b> Return only the strings that contain a pattern match. str_subset(fruit, "p")                                                                                      |
| <br>→<br><br>NA<br>NA              | <b>str_extract(string, pattern)</b> Return the first pattern match found in each string, as a vector. Also <b>str_extract_all()</b> to return every pattern match. str_extract(fruit, "[aeiou]")                     |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_match(string, pattern)</b> Return the first pattern match found in each string, as a matrix with a column for each () group in pattern. Also <b>str_match_all()</b> . str_match(sentences, "(a the) ([^ +])") |

## Manage Lengths

|                                    |                                                                                                                                                                                |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br>→<br><br>4<br>6<br>2<br>3      | <b>str_length(string)</b> The width of strings (i.e. number of code points, which generally equals the number of characters). str_length(fruit)                                |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_pad(string, width, side = c("left", "right", "both"), pad = " ")</b> Pad strings to constant width. str_pad(fruit, 17)                                                  |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...")</b> Truncate the width of strings, replacing content with ellipsis. str_trunc(sentences, 6) |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_trim(string, side = c("both", "left", "right"))</b> Trim whitespace from the start and/or end of a string. str_trim(str_pad(fruit, 17))                                 |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_squish(string)</b> Trim whitespace from each end and collapse multiple spaces into single spaces. str_squish(str_pad(fruit, 17, "both"))                                |

## Mutate Strings

|                   |                                                                                                                                                                   |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br>↓<br>a string | <b>str_sub()</b> <- value. Replace substrings by identifying the substrings with str_sub() and assigning into the results. str_sub(fruit, 1, 3) <- "str"          |
| <br>↓<br>a string | <b>str_replace(string, pattern, replacement)</b> Replace the first matched pattern in each string. Also <b>str_remove()</b> . str_replace(fruit, "p", "-")        |
| <br>↓<br>a string | <b>str_replace_all(string, pattern, replacement)</b> Replace all matched patterns in each string. Also <b>str_remove_all()</b> . str_replace_all(fruit, "p", "-") |
| <br>↓<br>a string | <b>str_to_lower(string, locale = "en")<sup>1</sup></b> Convert strings to lower case. str_to_lower(sentences)                                                     |
| <br>↓<br>A STRING | <b>str_to_upper(string, locale = "en")<sup>1</sup></b> Convert strings to upper case. str_to_upper(sentences)                                                     |
| <br>↓<br>A String | <b>str_to_title(string, locale = "en")<sup>1</sup></b> Convert strings to title case. Also <b>str_to_sentence()</b> . str_to_title(sentences)                     |

## Join and Split

|                                    |                                                                                                                                                                                                                                                                                                   |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br>→<br><br>4<br>1<br>3<br>2      | <b>str_c(..., sep = "", collapse = NULL)</b> Join multiple strings into a single string. str_c(letters, LETTERS)                                                                                                                                                                                  |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_flatten(string, collapse = "")</b> Combines into a single string, separated by collapse. str_flatten(fruit, ",")                                                                                                                                                                           |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_dup(string, times)</b> Repeat strings times times. Also <b>str_unique()</b> to remove duplicates. str_dup(fruit, times = 2)                                                                                                                                                                |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_split_fixed(string, pattern, n)</b> Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also <b>str_split()</b> to return a list of substrings and <b>str_split_n()</b> to return the nth substring. str_split_fixed(sentences, " ", n=3) |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_glue(..., .sep = "", .envir = parent.frame())</b> Create a string from strings and {expressions} to evaluate. str_glue("Pi is {pi}")                                                                                                                                                       |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA")</b> Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. str_glue_data(mtcars, "[rownames(mtcars)] has {hp} hp")                                                       |

## Order Strings

|                                    |                                                                                                                                                                                              |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br>→<br><br>4<br>1<br>3<br>2      | <b>str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)<sup>1</sup></b> Return the vector of indexes that sorts a character vector. fruit[str_order(fruit)] |
| <br>→<br><br>5<br>4<br>3<br>2<br>1 | <b>str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)<sup>1</sup></b> Sort a character vector. str_sort(fruit)                                             |

## Helpers

|                                                           |                                                                                                                                                                                   |
|-----------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <br>→<br>banana                                           | <b>str_conv(string, encoding)</b> Override the encoding of a string. str_conv(fruit, "ISO-8859-1")                                                                                |
| <br>→<br>banana                                           | <b>str_view_all(string, pattern, match = NA)</b> View HTML rendering of all regex matches. Also <b>str_view()</b> to see only the first match. str_view_all(sentences, "[aeiou]") |
| <br>→<br>TRUE<br>TRUE<br>FALSE<br>TRUE                    | <b>str_equal(x, y, locale = "en", ignore_case = FALSE, ...)<sup>1</sup></b> Determine if two strings are equivalent. str_equal(c("a", "b"), c("a", "c"))                          |
| This is a long sentence.<br>↓<br>This is a long sentence. | <b>str_wrap(string, width = 80, indent = 0, exdent = 0)</b> Wrap strings into nicely formatted paragraphs. str_wrap(sentences, 20)                                                |

<sup>1</sup> See [bit.ly/ISO639-1](https://bit.ly/ISO639-1) for a complete list of locales.

# Need to Know

Pattern arguments in string 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 *regexp*s, are a concise language for describing patterns in strings.

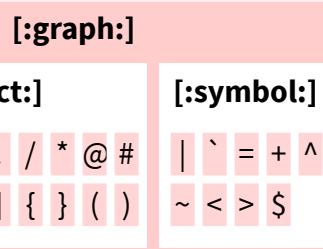
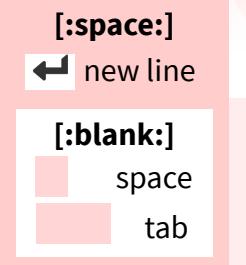
## MATCH CHARACTERS

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

see <- function(rx) str\_view\_all("abc ABC 123\\t.!?\n\\n", rx)

abc ABC 123 .!?\n  
abc ABC 123 .!?

see <- function(rx) str\_view\_all("abc ABC 123\\t.!?\n\\n", rx)



# Apply functions with purrr :: CHEAT SHEET



## Map Functions

### ONE LIST

**map(.x, .f, ...)** Apply a function to each element of a list or vector, return a list.  
`x <- list(1:10, 11:20, 21:30)  
l1 <- list(x = c("a", "b"), y = c("c", "d"))  
map(l1, sort, decreasing = TRUE)`



**map\_dbl(.x, .f, ...)**  
Return a double vector.  
`map_dbl(x, mean)`

**map\_int(.x, .f, ...)**  
Return an integer vector.  
`map_int(x, length)`

**map\_chr(.x, .f, ...)**  
Return a character vector.  
`map_chr(l1, paste, collapse = "")`

**map\_lgl(.x, .f, ...)**  
Return a logical vector.  
`map_lgl(x, is.integer)`

**map\_dfc(.x, .f, ...)**  
Return a data frame created by column-binding.  
`map_dfc(l1, rep, 3)`

**map\_dfr(.x, .f, ..., .id = NULL)**  
Return a data frame created by row-binding.  
`map_dfr(x, summary)`

**walk(.x, .f, ...)** Trigger side effects, return invisibly.  
`walk(x, print)`

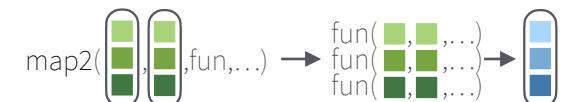
## Function Shortcuts

Use `~.` with functions like **map()** that have single arguments.

`map(l, ~ . + 2)`  
becomes  
`map(l, function(x) x + 2 )`

### TWO LISTS

**map2(.x, .y, .f, ...)** Apply a function to pairs of elements from two lists or vectors, return a list.  
`y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")  
map2(x, y, ~ .x * .y)`



**map2\_dbl(.x, .y, .f, ...)**  
Return a double vector.  
`map2_dbl(y, z, ~ .x / .y)`

**map2\_int(.x, .y, .f, ...)**  
Return an integer vector.  
`map2_int(y, z, `+`)`

**map2\_chr(.x, .y, .f, ...)**  
Return a character vector.  
`map2_chr(l1, l2, paste, collapse = "", sep = ":" )`

**map2\_lgl(.x, .y, .f, ...)**  
Return a logical vector.  
`map2_lgl(l2, l1, `%in%`)`

**map2\_dfc(.x, .y, .f, ...)**  
Return a data frame created by column-binding.  
`map2_dfc(l1, l2, ~ as.data.frame(c(.x, .y)))`

**map2\_dfr(.x, .y, .f, ..., .id = NULL)**  
Return a data frame created by row-binding.  
`map2_dfr(l1, l2, ~ as.data.frame(c(.x, .y)))`

**walk2(.x, .y, .f, ...)** Trigger side effects, return invisibly.  
`walk2(objs, paths, save)`

Use `~ .x .y` with functions like **map2()** that have two arguments.

`map2(l, p, ~ .x +.y)`  
becomes  
`map2(l, p, function(l, p) l + p)`

Use a **string** or an **integer** with any map function to index list elements by name or position. `map(l, "name")` becomes `map(l, function(x) x[["name"]])`

### MANY LISTS

**pmap(.l, .f, ...)** Apply a function to groups of elements from a list of lists or vectors, return a list.  
`pmap(list(x, y, z), ~ ..1 * (.2 + ..3))`



**pmap\_dbl(.l, .f, ...)**  
Return a double vector.  
`pmap_dbl(list(y, z), ~ .x / .y)`

**pmap\_int(.l, .f, ...)**  
Return an integer vector.  
`pmap_int(list(y, z), `+`)`

**pmap\_chr(.l, .f, ...)**  
Return a character vector.  
`pmap_chr(list(l1, l2), paste, collapse = "", sep = ":" )`

**pmap\_lgl(.l, .f, ...)**  
Return a logical vector.  
`pmap_lgl(list(l2, l1), `%in%`)`

**pmap\_dfc(.l, .f, ...)**  
Return a data frame created by column-binding.  
`pmap_dfc(list(l1, l2), ~ as.data.frame(c(.x, .y)))`

**pmap\_dfr(.l, .f, ..., .id = NULL)**  
Return a data frame created by row-binding.  
`pmap_dfr(list(l1, l2), ~ as.data.frame(c(.x, .y)))`

**pwalk(.l, .f, ...)** Trigger side effects, return invisibly.  
`pwalk(list(objs, paths), save)`

Use `~ ..1 ..2 ..3` etc with functions like **pmap()** that have many arguments.

`pmap(list(a, b, c), ~ ..3 + ..1 - ..2)`  
becomes  
`pmap(list(a, b, c), function(a, b, c) c + a - b)`

Use `~ .x .y` with functions like **imap()**. `.x` will get the list value and `.y` will get the index, or name if available.

`imap(list(a, b, c), ~ paste0(.y, ": ", .x))`  
outputs "index: value" for each item

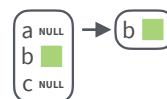


# Work with Lists

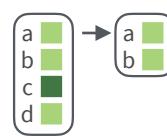
## Filter



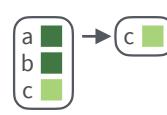
**keep(.x, .p, ...)**  
Select elements that pass a logical test.  
Conversely, **discard()**.  
`keep(x, is.na)`



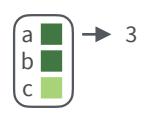
**compact(.x, .p = identity)**  
Drop empty elements.  
`compact(x)`



**head\_while(.x, .p, ...)**  
Return head elements until one does not pass.  
Also **tail\_while()**.  
`head_while(x, is.character)`



**detect(.x, .f, ..., dir = c("forward", "backward"), .right = NULL, .default = NULL)**  
Find first element to pass.  
`detect(x, is.character)`



**detect\_index(.x, .f, ..., dir = c("forward", "backward"), .right = NULL)** Find index of first element to pass.  
`detect_index(x, is.character)`



**every(.x, .p, ...)**  
Do all elements pass a test?  
`every(x, is.character)`



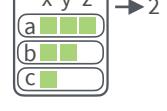
**some(.x, .p, ...)**  
Do some elements pass a test?  
`some(x, is.character)`



**none(.x, .p, ...)**  
Do no elements pass a test?  
`none(x, is.character)`

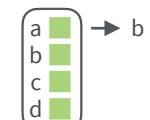


**has\_element(.x, .y)**  
Does a list contain an element?  
`has_element(x, "foo")`

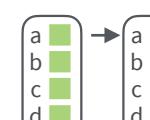


**vec\_depth(x)**  
Return depth (number of levels of indexes).  
`vec_depth(x)`

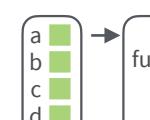
## Index



**pluck(.x, ..., .default=NULL)**  
Select an element by name or index. Also **attr\_getter()** and **chuck()**.  
`pluck(x, "b")`  
`x %>% pluck("b")`

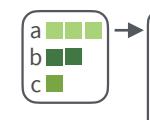


**assign\_in(x, where, value)**  
Assign a value to a location using pluck selection.  
`assign_in(x, "b", 5)`  
`x %>% assign_in("b", 5)`



**modify\_in(.x, .where, .f)**  
Apply a function to a value at a selected location.  
`modify_in(x, "b", abs)`  
`x %>% modify_in("b", abs)`

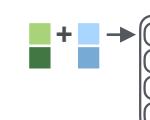
## Reshape



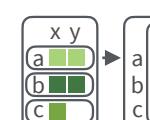
**flatten(.x)** Remove a level of indexes from a list.  
Also **flatten\_chr()** etc.  
`flatten(x)`



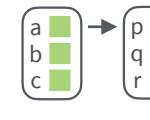
**array\_tree(array, margin = NULL)** Turn array into list.  
Also **array\_branch()**.  
`array_tree(x, margin = 3)`



**cross2(.x, .y, .filter = NULL)**  
All combinations of .x and .y.  
Also **cross()**, **cross3()**, and **cross\_df()**.  
`cross2(1:3, 4:6)`

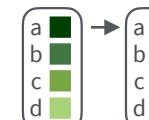


**transpose(.l, .names = NULL)**  
Transposes the index order in a multi-level list.  
`transpose(x)`

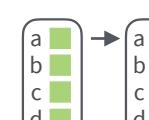


**set\_names(x, nm = x)**  
Set the names of a vector/list directly or with a function.  
`set_names(x, c("p", "q", "r"))`  
`set_names(x, tolower)`

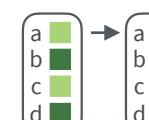
## Modify



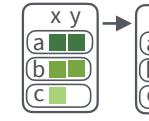
**modify(.x, .f, ...)** Apply a function to each element. Also **modify2()**, and **imodify()**.  
`modify(x, ~.+ 2)`



**modify\_at(.x, .at, .f, ...)** Apply a function to selected elements.  
Also **map\_at()**.  
`modify_at(x, "b", ~.+ 2)`

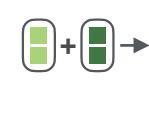


**modify\_if(.x, .p, .f, ...)** Apply a function to elements that pass a test.  
Also **map\_if()**.  
`modify_if(x, is.numeric, ~.+2)`

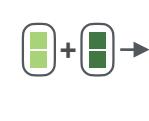


**modify\_depth(.x, .depth, .f, ...)** Apply function to each element at a given level of a list. Also **map\_depth()**.  
`modify_depth(x, 2, ~.+ 2)`

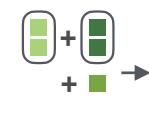
## Combine



**append(x, values, after = length(x))** Add values to end of list.  
`append(x, list(d = 1))`



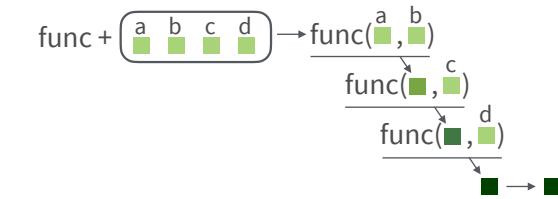
**prepend(x, values, before = 1)** Add values to start of list.  
`prepend(x, list(d = 1))`



**splice(...)** Combine objects into a list, storing S3 objects as sub-lists.  
`splice(x, y, "foo")`

## Reduce

**reduce(.x, .f, ..., .init, .dir = c("forward", "backward"))** Apply function recursively to each element of a list or vector. Also **reduce2()**.  
`reduce(x, sum)`



## List-Columns

**List-columns** are columns of a data frame where each element is a list or vector instead of an atomic value. Columns can also be lists of data frames. See **tidyverse** for more about nested data and list columns.

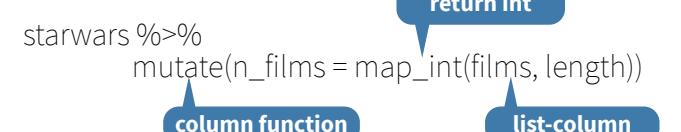
### WORK WITH LIST-COLUMNS

Manipulate list-columns like any other kind of column, using **dplyr** functions like **mutate()** and **transmute()**. Because each element is a list, use **map functions** within a column function to manipulate each element.

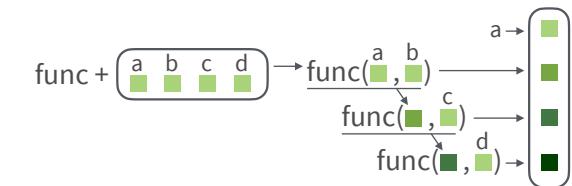
**map()**, **map2()**, or **pmap()** return lists and will **create new list-columns**.



Suffixed map functions like **map\_int()** return an atomic data type and will **simplify list-columns into regular columns**.



**accumulate(.x, .f, ..., .init)** Reduce a list, but also return intermediate results. Also **accumulate2()**.  
`accumulate(x, sum)`



# rmarkdown :: CHEAT SHEET

## What is rmarkdown?



**.Rmd files** • Develop your code and ideas side-by-side in a single document. Run code as individual chunks or as an entire document.

**Dynamic Documents** • Knit together plots, tables, and results with narrative text. Render to a variety of formats like HTML, PDF, MS Word, or MS Powerpoint.

**Reproducible Research** • Upload, link to, or attach your report to share. Anyone can read or run your code to reproduce your work.

## Workflow

- 1 Open a **new .Rmd file** in the RStudio IDE by going to *File > New File > R Markdown*.
- 2 **Embed code** in chunks. Run code by line, by chunk, or all at once.
- 3 **Write text** and add tables, figures, images, and citations. Format with Markdown syntax or the RStudio Visual Markdown Editor.
- 4 **Set output format(s) and options** in the YAML header. Customize themes or add parameters to execute or add interactivity with Shiny.
- 5 **Save and render** the whole document. Knit periodically to preview your work as you write.
- 6 **Share your work!**

## Embed Code with knitr

### CODE CHUNKS

Surround code chunks with `{{r}}` and `{{` or use the Insert Code Chunk button. Add a chunk label and/or chunk options inside the curly braces after **r**.

```
```{r chunk-label, include=FALSE}
summary(mtcars)
```
```

### SET GLOBAL OPTIONS

Set options for the entire document in the first chunk.

```
```{r include=FALSE}
knitr::opts_chunk$message = FALSE
```
```

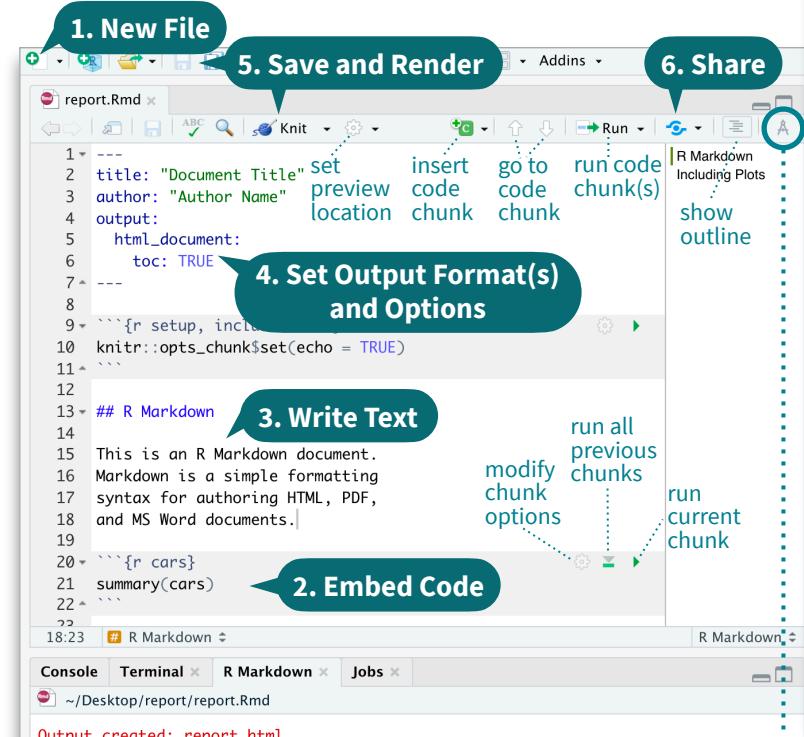
### INLINE CODE

Insert `r <code>` into text sections. Code is evaluated at render and results appear as text.

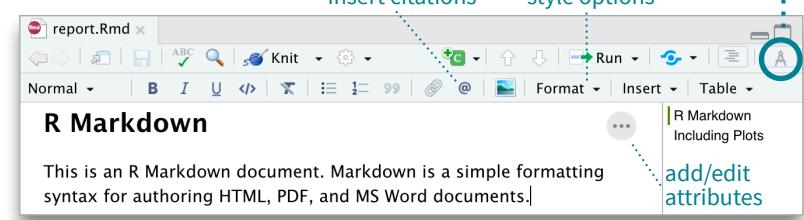
"Built with `r getRversion()`" --> "Built with 4.1.0"



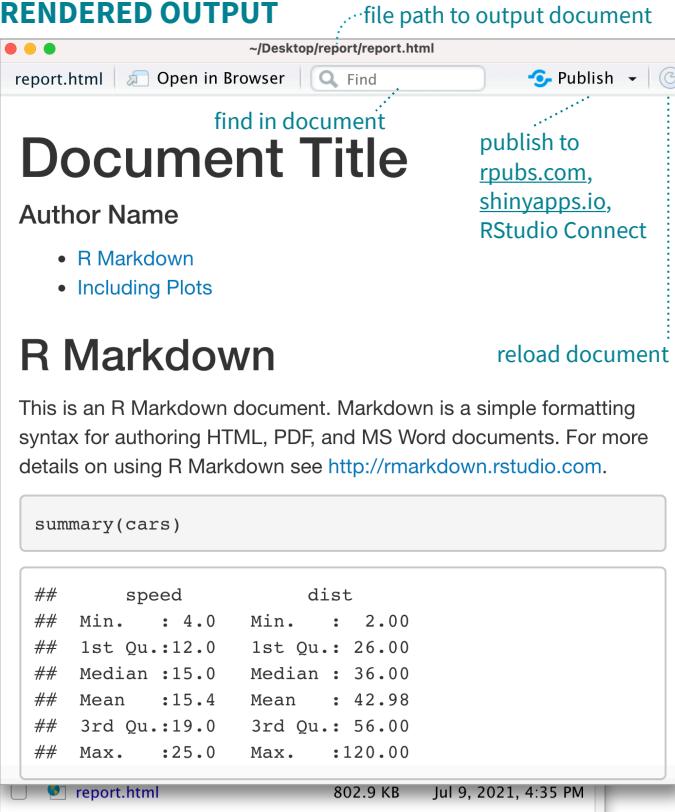
### SOURCE EDITOR



### VISUAL EDITOR



### RENDERED OUTPUT



## Write with Markdown

The syntax on the left renders as the output on the right.

Plain text.

Plain text.

End a line with two spaces to start a new paragraph.

End a line with two spaces to start a new paragraph.

Also end with a backslash\ to make a new line.

Also end with a backslash\ to make a new line.

**italics\*** and **\*\*bold\*\***

**italics** and **bold**

superscript<sup>2</sup>/subscript<sub>2</sub>

superscript<sup>2</sup>/subscript<sub>2</sub>

~~strikethrough~~

strikethrough

escaped: `\* \\_ \|`

escaped: \* \\_ |

endash: --, emdash: ---

endash: -, emdash: --

### Header 1

### Header 2

...

### Header 6

- unordered list

• item 2

- item 2a (indent 1 tab)

• item 2b

1. ordered list

1. item 2

- item 2a (indent 1 tab)

• item 2b

<link url>

[This is a link.](link url)

[This is another link][id].

This is another link.

<http://www.rstudio.com/>

This is a link.

This is another link.



## Insert Citations

Create citations from a bibliography file, a Zotero library, or from DOI references.

### BUILD YOUR BIBLIOGRAPHY

- Add BibTeX or CSL bibliographies to the YAML header.

---

title: "My Document"  
bibliography: references.bib  
link-citations: TRUE

---

- If Zotero is installed locally, your main library will automatically be available.

- Add citations by DOI by searching "from DOI" in the **Insert Citation** dialog.

### INSERT CITATIONS

- Access the **Insert Citations** dialog in the Visual Editor by clicking the @ symbol in the toolbar or by clicking **Insert > Citation**.

- Add citations with markdown syntax by typing **[@cite]** or **@cite**.

## Insert Tables

Output data frames as tables using **kable**(data, caption).

```
```{r}
data <- faithful[1:4, ]
knitr::kable(data,
             caption = "Table with kable")
...```

```

Other table packages include **flextable**, **gt**, and **kableExtra**.

Table with kable	
eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

Results	
Plots	Tables
more text	

Set Output Formats and their Options in YAML

Use the document's YAML header to set an **output format** and customize it with **output options**.

```
---
```

```
title: "My Document"
```

```
author: "Author Name"
```

```
output:
```

```
  html_document:
```

```
    toc: TRUE
```

```
---
```

Indent format 2 characters,
indent options 4 characters

OUTPUT FORMAT	CREATES
html_document	.html
pdf_document*	.pdf
word_document	Microsoft Word (.docx)
powerpoint_presentation	Microsoft Powerpoint (.pptx)
odt_document	OpenDocument Text
rtf_document	Rich Text Format
md_document	Markdown
github_document	Markdown for Github
ioslides_presentation	ioslides HTML slides
slidy_presentation	Slidy HTML slides
beamer_presentation*	Beamer slides

* Requires LaTeX, use `tinytex::install_tinytex()`
Also see `flexdashboard`, `bookdown`, `distill`, and `blogdown`.

IMPORTANT OPTIONS	DESCRIPTION	HTML	PDF	MS Word	MS PPT
<code>anchor_sections</code>	Show section anchors on mouse hover (TRUE or FALSE)	X			
<code>citation_package</code>	The LaTeX package to process citations ("default", "natbib", "biblatex")	X			
<code>code_download</code>	Give readers an option to download the .Rmd source code (TRUE or FALSE)	X			
<code>code_folding</code>	Let readers to toggle the display of R code ("none", "hide", or "show")	X			
<code>css</code>	CSS or SCSS file to use to style document (e.g. "style.css")	X			
<code>dev</code>	Graphics device to use for figure output (e.g. "png", "pdf")	X X			
<code>df_print</code>	Method for printing data frames ("default", "kable", "tibble", "paged")	X X X X			
<code>fig_caption</code>	Should figures be rendered with captions (TRUE or FALSE)	X X X X			
<code>highlight</code>	Syntax highlighting ("tango", "pygments", "kate", "zenburn", "textmate")	X X X			
<code>includes</code>	File of content to place in doc ("in_header", "before_body", "after_body")	X X			
<code>keep_md</code>	Keep the Markdown .md file generated by knitting (TRUE or FALSE)	X X X X			
<code>keep_tex</code>	Keep the intermediate TEX file used to convert to PDF (TRUE or FALSE)	X			
<code>latex_engine</code>	LaTeX engine for producing PDF output ("pdflatex", "xelatex", or "lualatex")	X			
<code>reference_docx/_doc</code>	docx/pptx file containing styles to copy in the output (e.g. "file.docx", "file.pptx")	X X			
<code>theme</code>	Theme options (see Bootswatch and Custom Themes below)	X			
<code>toc</code>	Add a table of contents at start of document (TRUE or FALSE)	X X X X			
<code>toc_depth</code>	The lowest level of headings to add to table of contents (e.g. 2, 3)	X X X X			
<code>toc_float</code>	Float the table of contents to the left of the main document content (TRUE or FALSE)	X			

Use `?<output format>` to see all of a format's options, e.g. `?html_document`

More Header Options

PARAMETERS

Parameterize your documents to reuse with new inputs (e.g., data, values, etc.).

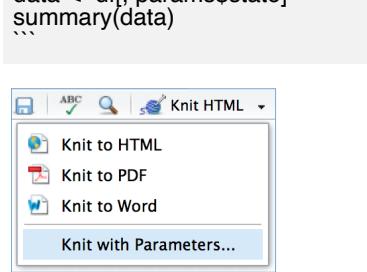
1. **Add parameters** in the header as sub-values of `params`.

```
---
```

```
params:
```

```
  state: "hawaii"
```

```
---
```



2. **Call parameters** in code using `params$<name>`.

3. **Set parameters** with Knit with Parameters or the `params` argument of `render()`.

REUSABLE TEMPLATES

1. **Create a new package** with a `inst/rmarkdown/templates` directory.

2. **Add a folder** containing `template.yaml` (below) and `skeleton.Rmd` (template contents).

```
---
```

```
name: "My Template"
```

```
---
```

3. **Install** the package to access template by going to **File > New R Markdown > From Template**.

BOOTSWATCH THEMES

Customize HTML documents with Bootswatch themes from the `bslib` package using the theme output option.

Use `bslib::bootswatch_themes()` to list available themes.



```
---
```

```
title: "Document Title"
```

```
author: "Author Name"
```

```
output:
```

```
  html_document:
```

```
    theme:
```

```
      bootswatch: solar
```

```
---
```

CUSTOM THEMES

Customize individual HTML elements using `bslib` variables. Use `?bs_theme` to see more variables.

```
---
```

```
output:
```

```
  html_document:
```

```
    theme:
```

```
      bg: "#121212"
```

```
      fg: "#E4E4E4"
```

```
      base_font:
```

```
      google: "Prompt"
```

```
---
```

More on `bslib` at pkgs.rstudio.com/bslib/.

STYLING WITH CSS AND SCSS

Add CSS and SCSS to your document by adding a path to a file with the `css` option in the YAML header.

```
---
```

```
title: "My Document"
```

```
author: "Author Name"
```

```
output:
```

```
  html_document:
```

```
    css: "style.css"
```

```
---
```

Apply CSS styling by writing HTML tags directly or:

- Use markdown to apply style attributes inline.
- Bracketed Span
A [green]{.my-color} word.
- Fenced Div
:::{.my-color}
All of these words are green.
:::
- Use the Visual Editor. Go to **Format > Div/Span** and add CSS styling directly with Edit Attributes.

.my-css-tag ...
This is a div with some text in it.

Render

When you render a document, rmarkdown:

1. Runs the code and embeds results and text into an .md file with knitr.
2. Converts the .md file into the output format with Pandoc.



Save, then **Knit** to preview the document output. The resulting HTML/PDF/MS Word/etc. document will be created and saved in the same directory as the .Rmd file.

Use `rmarkdown::render()` to render/knit in the R console. See `?render` for available options.



Share

Publish on RStudio Connect

to share R Markdown documents securely, schedule automatic updates, and interact with parameters in real time.

rstudio.com/products/connect/



INTERACTIVITY

Turn your report into an interactive Shiny document in 4 steps:

1. Add `runtime: shiny` to the YAML header.
2. Call Shiny input functions to embed input objects.
3. Call Shiny render functions to embed reactive output.
4. Render with `rmarkdown::run()` or click **Run Document** in RStudio IDE.

```
---
```

```
output: html_document
```

```
runtime: shiny
```

```
---
```

```
---
```

```
---
```

```
---
```

```
---
```

```
---
```

```
---
```

```
---
```

```
---
```

How many cars?	
5	
speed	dist
1	4.00
2	10.00
3	4.00
4	22.00
5	16.00

Also see Shiny Prerendered for better performance.

rmarkdown.rstudio.com/authoring_shiny_prerendered

Embed a complete app into your document with `shiny::shinyAppDir()`. More at bookdown.org/yihui/rmarkdown/shiny-embedded.html.