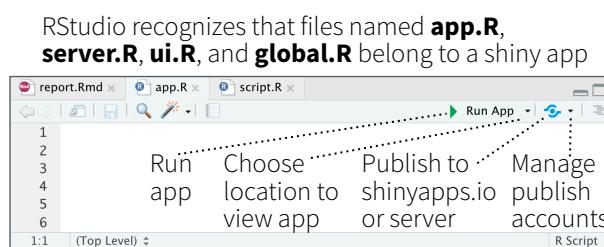
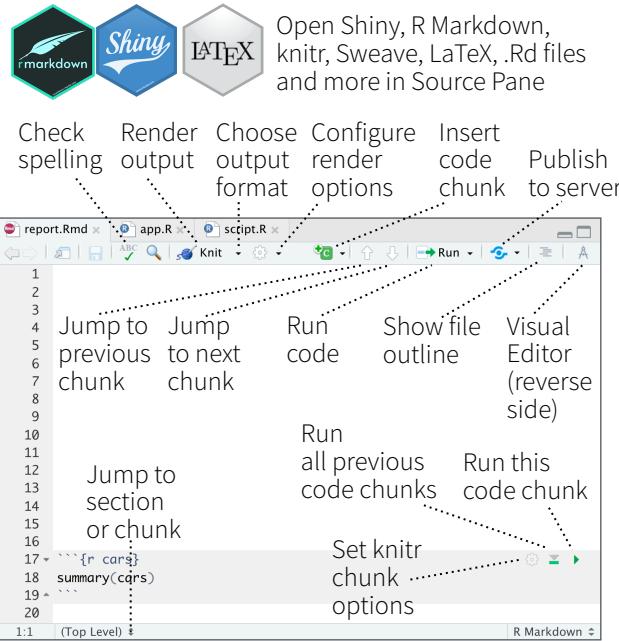


RStudio IDE :: CHEATSHEET

Documents and Apps

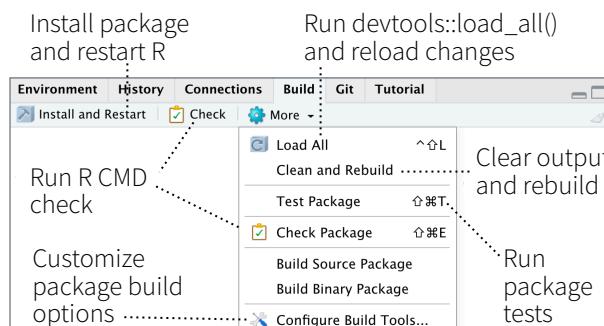


Package Development

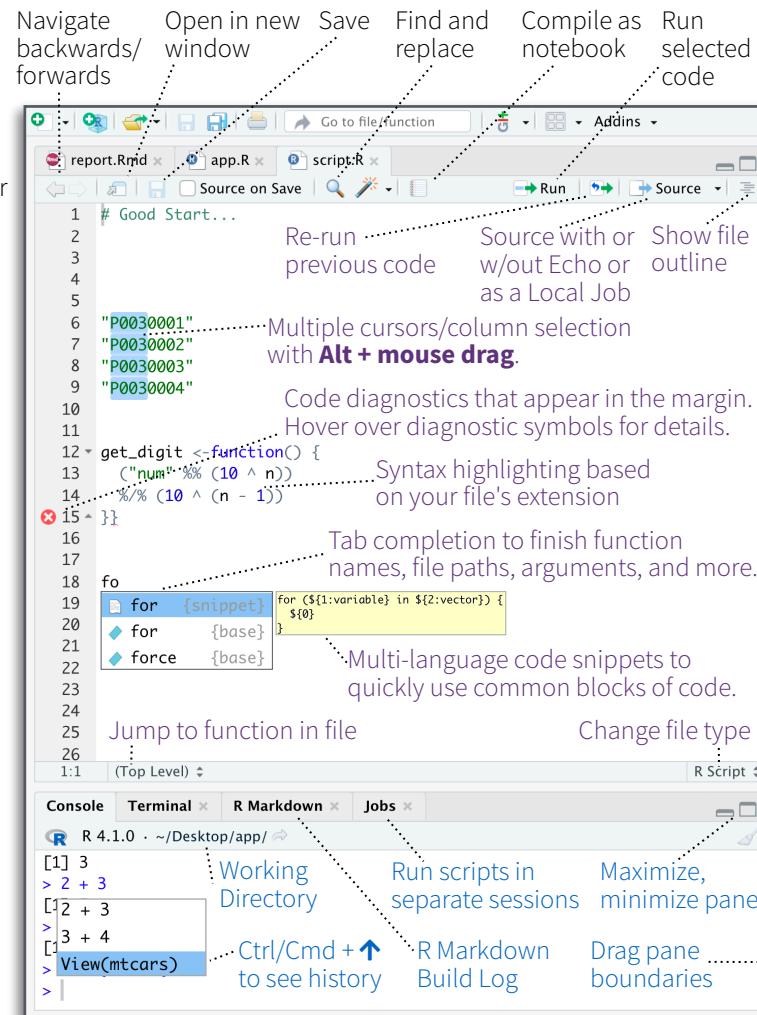
Create a new package with [File > New Project > New Directory > R Package](#)
Enable roxygen documentation with [Tools > Project Options > Build Tools](#)

Roxygen guide at [Help > Roxygen Quick Reference](#)

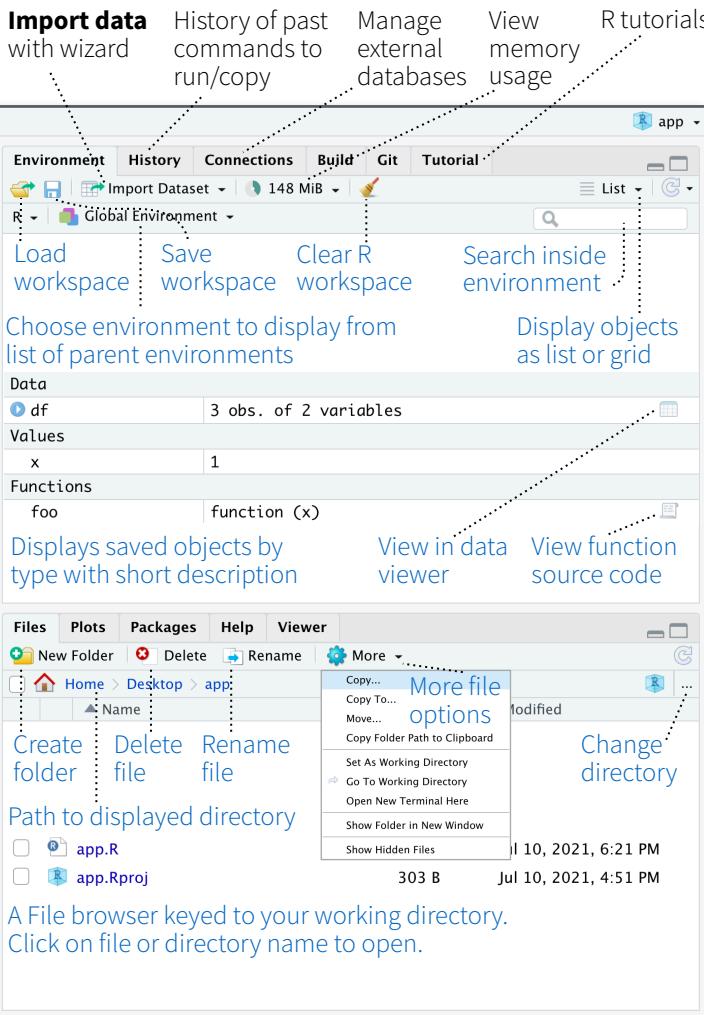
See package information in the [Build Tab](#)



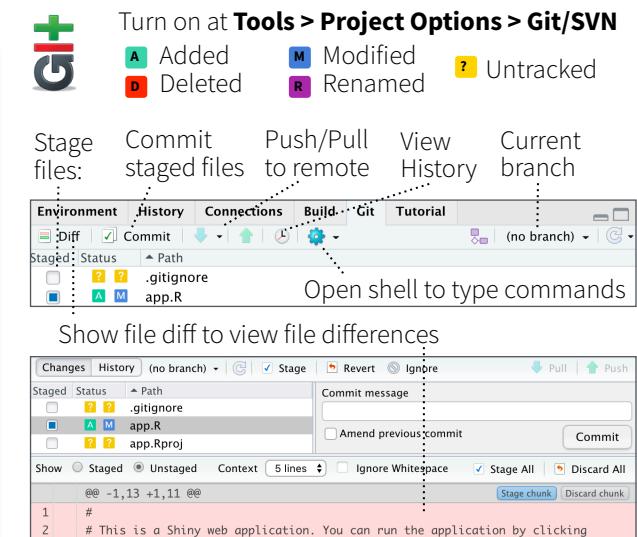
Source Editor



Tab Panes

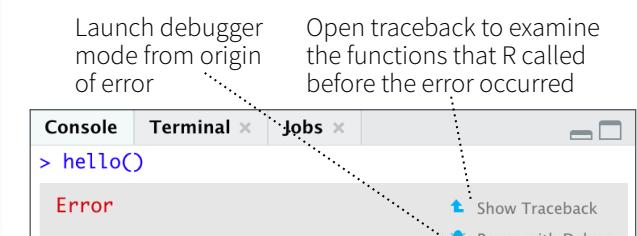


Version Control



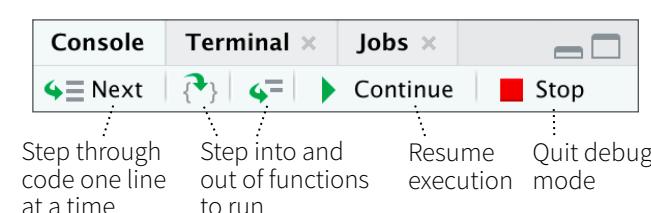
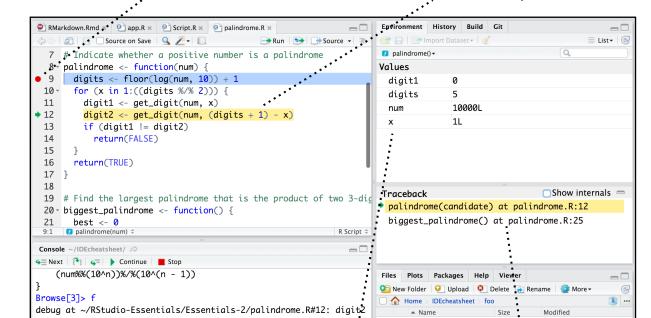
Debug Mode

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.

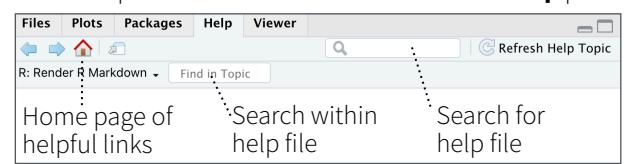
Highlighted line shows where execution has paused



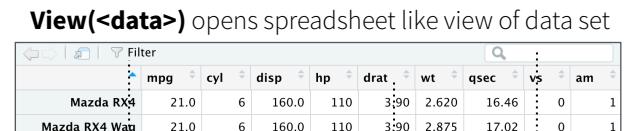
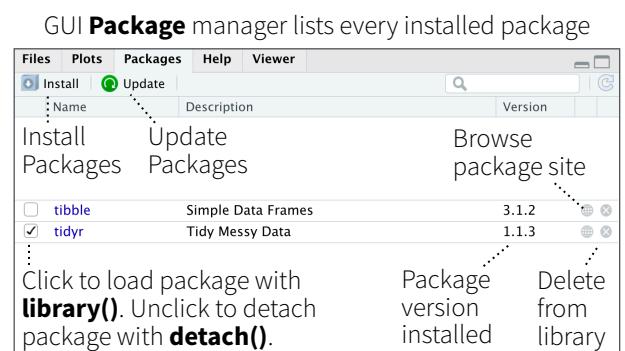
RStudio opens plots in a dedicated **Plots** pane



RStudio opens documentation in a dedicated **Help** pane



Viewer pane displays HTML content, such as Shiny apps, RMarkdown reports, and interactive visualizations



Keyboard Shortcuts

RUN CODE

Search command history
Interrupt current command
Clear console

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

Navigate Code

Go to File/Function

Ctrl+. .

Ctrl+. .

Write Code

Attempt completion

Insert <- (assignment operator)
Insert |> or %>% (pipe operator)
(Un)Comment selection

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

MAKE PACKAGES

Load All (devtools)
Test Package (Desktop)
Document Package

	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)
Insert chunk (Sweave & Knitr)
Run from start to current line

Ctrl+Shift+K
Ctrl+Alt+I
Ctrl+Alt+B

Cmd+Shift+K
Cmd+Option+I
Cmd+Option+B

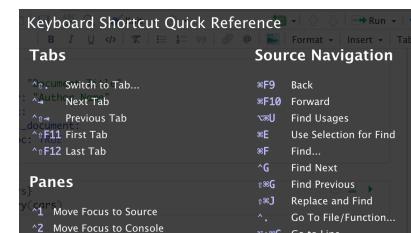
MORE KEYBOARD SHORTCUTS

Keyboard Shortcuts Help
Show Command Palette

Alt+Shift+K
Ctrl+Shift+P

Option+Shift+K
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:

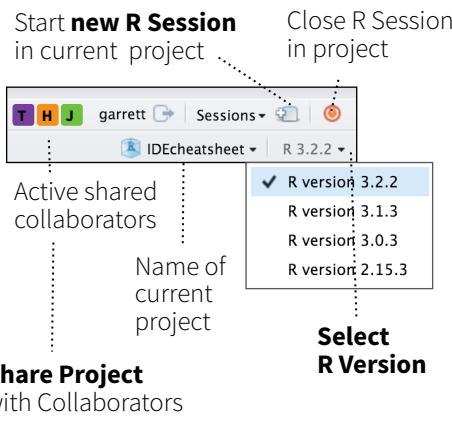
- open and run multiple R sessions at once
- tune your resources to improve performance
- administrative tools for managing user sessions
- collaborate real-time with others in shared projects
- switch easily from one version of R to a different version
- integrate with your authentication, authorization, 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

The screenshot shows the RStudio Visual Editor interface with several keyboard shortcuts overlaid. The editor is displaying an R Markdown document with code and text. Key features shown include:

- 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
- Add/Edit attributes
- Set knitr chunk options
- Run this and all previous code chunks
- Run this code chunk

Below the editor, there's a preview pane showing R code and its rendered output.

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

Run Remote Jobs

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

The screenshot shows the RStudio Job Launcher interface. It displays a list of jobs and their status (Running, Succeeded, Idle). The interface includes tabs for Console, Terminal, Jobs, and Läuncher. The Läuncher tab shows a list of launcher jobs with columns for Name, Status, Local/Remote, and submission time.

Base R Cheat Sheet

Getting Help

Accessing the help files

?mean

Get help of a particular function.

help.search('weighted mean')

Search the help files for a word or phrase.

help(package = 'dplyr')

Find help for a package.

More about an object

str(iris)

Get a summary of an object's structure.

class(iris)

Find the class an object belongs to.

Using Libraries

install.packages('dplyr')

Download and install a package from CRAN.

library(dplyr)

Load the package into the session, making all its functions available to use.

dplyr::select

Use a particular function from a package.

data(iris)

Load a built-in dataset into the environment.

Working Directory

getwd()

Find the current working directory (where inputs are found and outputs are sent).

setwd('C://file/path')

Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors			Programming					
Creating Vectors			For Loop			While Loop		
c(2, 4, 6)	2 4 6	Join elements into a vector	for (variable in sequence){	Do something	}	while (condition){	Do something	}
2:6	2 3 4 5 6	An integer sequence						
seq(2, 3, by=0.5)	2.0 2.5 3.0	A complex sequence						
rep(1:2, times=3)	1 2 1 2 1 2	Repeat a vector	for (i in 1:4){	j <- i + 10	print(j)	while (i < 5){	print(i)	i <- i + 1
rep(1:2, each=3)	1 1 1 2 2 2	Repeat elements of a vector						
Vector Functions								
sort(x)	rev(x)		If Statements			Functions		
Return x sorted.	Return x reversed.		if (condition){	Do something		function_name <- function(var){	Do something	
table(x)	unique(x)	See counts of values.	} else {	Do something different	}	return(new_variable)		
Selecting Vector Elements								
By Position			Example			Example		
x[4]	The fourth element.		if (i > 3){	print('Yes')		square <- function(x){		
x[-4]	All but the fourth.		} else {	print('No')	}	squared <- x*x		
x[2:4]	Elements two to four.					return(squared)		
x[-(2:4)]	All elements except two to four.		Reading and Writing Data					
x[c(1, 5)]	Elements one and five.		Input			Output		
By Value								
x[x == 10]	Elements which are equal to 10.		df <- read.table('file.txt')	write.table(df, 'file.txt')		Description		
x[x < 0]	All elements less than zero.					Read and write a delimited text file.		
x[x %in% c(1, 2, 5)]	Elements in the set 1, 2, 5.		df <- read.csv('file.csv')	write.csv(df, 'file.csv')		Read and write a comma separated value file. This is a special case of read.table/write.table.		
Named Vectors								
x['apple']	Element with name 'apple'.		load('file.RData')	save(df, file = 'file.Rdata')		Read and write an R data file, a file type special for R.		
Conditions			a == b	Are equal	a > b	Greater than	a >= b	Greater than or equal to
			a != b	Not equal	a < b	Less than	a <= b	Less than or equal to

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

as.logical	TRUE, FALSE, TRUE	Boolean values (TRUE or FALSE).
as.numeric	1, 0, 1	Integers or floating point numbers.
as.character	'1', '0', '1'	Character strings. Generally preferred to factors.
as.factor	'1', '0', '1', levels: '1', '0'	Character strings with preset levels. Needed for some statistical models.

Maths Functions

log(x)	Natural log.	sum(x)	Sum.
exp(x)	Exponential.	mean(x)	Mean.
max(x)	Largest element.	median(x)	Median.
min(x)	Smallest element.	quantile(x)	Percentage quantiles.
round(x, n)	Round to n decimal places.	rank(x)	Rank of elements.
signif(x, n)	Round to n significant figures.	var(x)	The variance.
cor(x, y)	Correlation.	sd(x)	The standard deviation.

Variable Assignment

```
> a <- 'apple'  
> a  
[1] 'apple'
```

The Environment

ls()	List all variables in the environment.
rm(x)	Remove x from the environment.
rm(list = ls())	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrixes

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.

	<code>m[2,]</code> - Select a row	<code>t(m)</code> Transpose
	<code>m[, 1]</code> - Select a column	<code>m %*% n</code> Matrix Multiplication
	<code>m[2, 3]</code> - Select an element	<code>solve(m, n)</code> Find x in: $m \cdot x = n$

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is collection of elements which can be of different types.

<code>l[[2]]</code>	<code>l[1]</code>	<code>l\$x</code>	<code>l['y']</code>
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

Also see the [dplyr](#) library.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

Matrix subsetting

<code>df[, 2]</code>	
<code>df[2,]</code>	
<code>df[2, 2]</code>	

List subsetting

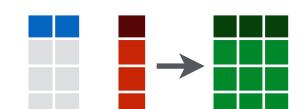
<code>df\$x</code>		<code>df[[2]]</code>	
--------------------	--	----------------------	--

Understanding a data frame

`View(df)`
See the full data frame.

`head(df)`
See the first 6 rows.

`nrow(df)`
Number of rows.



`cbind` - Bind columns.

`ncol(df)`
Number of columns.



`rbind` - Bind rows.

`dim(df)`
Number of columns and rows.

Strings

<code>paste(x, y, sep = ' ')</code>	Join multiple vectors together.
<code>paste(x, collapse = ' ')</code>	Join elements of a vector together.
<code>grep(pattern, x)</code>	Find regular expression matches in x.
<code>gsub(pattern, replace, x)</code>	Replace matches in x with a string.
<code>toupper(x)</code>	Convert to uppercase.
<code>tolower(x)</code>	Convert to lowercase.
<code>nchar(x)</code>	Number of characters in a string.

Factors

<code>factor(x)</code>	
<code>cut(x, breaks = 4)</code>	Turn a numeric vector into a factor but 'cutting' into sections.

Statistics

<code>lm(x ~ y, data=df)</code>	Linear model.
<code>glm(x ~ y, data=df)</code>	Generalised linear model.
<code>summary</code>	Get more detailed information out a model.
<code>prop.test</code>	Test for a difference between proportions.
<code>t.test(x, y)</code>	Preform a t-test for difference between means.
<code>pairwise.t.test</code>	Preform a t-test for paired data.
<code>aov</code>	Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	<code>rnorm</code>	<code>dnorm</code>	<code>pnorm</code>	<code>qnorm</code>
Poisson	<code>rpois</code>	<code>dpois</code>	<code>ppois</code>	<code>qpois</code>
Binomial	<code>rbinom</code>	<code>dbinom</code>	<code>pbinom</code>	<code>qbinom</code>
Uniform	<code>runif</code>	<code>dunif</code>	<code>unif</code>	<code>qunif</code>

Plotting

<code>plot(x)</code>	Values of x in order.
<code>plot(x, y)</code>	Values of x against y.
<code>hist(x)</code>	Histogram of x.

Dates

See the [lubridate](#) library.



Data import with the tidyverse :: CHEATSHEET

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	5	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	5	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	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	5	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")`

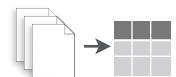
USEFUL READ ARGUMENTS

A	B	C
1	2	3
4	5	NA

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

x	y	z
A	B	C
1	2	3
4	5	NA

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")`

1	2	3
4	5	NA

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

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

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

A	B	C
NA	2	3
4	5	NA

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

A;B;C	1;5;2;3;0	
A	B	C
1	5	2
4	5	NA
		0

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

A	B	C
1	2	3
4	5	NA

write_delim(x, file, delim = " ") Write files with any delimiter.

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

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.

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(),
#   edu = col_character(),
#   earn = col_double()
# )
```

age is an integer

edu 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()** - "?"

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"
)
```

Import Spreadsheets with readxl

READ EXCEL FILES

	A	B	C	D	E
1	x1	x2	x3	x4	x5
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
A	B	C	D	E

- To **read multiple sheets**:
1. Get a vector of sheet names from the file path.
 2. Set the vector names to be the sheet names.
 3. Use purrr::map() and purrr::list_rbind() to read multiple files into one data frame.

```
path <- "your_file_path.xlsx"  
path >  
  excel_sheets() |>  
  set_names() |>  
  map(read_excel, path = path) |>  
  list_rbind()
```

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 **tidy** 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

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	x5
2	x		z	8	
3	y	7		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
1	1	x
2	y	5

1	A	B	C	D
1				

x1	x2	x3
1	x1	x2
2	y	5

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.

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 **tidy** 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.

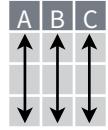
For whole-file operations (e.g. renaming, sharing, placing within a folder), see the tidyverse package **googledrive** at googledrive.tidyverse.org.

Data tidying with `tidyr` :: CHEATSHEET



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/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/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	

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

complete(data, ..., fill = list()) Add missing possible combinations of values of variables listed in ... Fill remaining variables with NA.

```
complete(mtcars, cyl, gear, carb)
```

x	x1	x2

<tbl_r cells="3" ix="1" maxc



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))
```

name	yr	lat	long
Amy	1975	27.5	-79.0
Amy	1975	28.5	-79.0
Amy	1975	29.5	-79.0
Bob	1979	22.0	-96.0
Bob	1979	22.5	-95.3
Bob	1979	23.0	-94.6
Zeta	2005	23.9	-35.6
Zeta	2005	24.2	-36.1
Zeta	2005	24.7	-36.6

name	yr	lat	long
Amy	1975	27.5	-79.0
Amy	1975	28.5	-79.0
Amy	1975	29.5	-79.0
Bob	1979	22.0	-96.0
Bob	1979	22.5	-95.3
Bob	1979	23.0	-94.6
Zeta	2005	23.9	-35.6
Zeta	2005	24.2	-36.1
Zeta	2005	24.7	-36.6

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)
```

"cell" contents			
name	yr	lat	long
Amy	1975	27.5	-79.0
Amy	1975	28.5	-79.0
Amy	1975	29.5	-79.0
Bob	1979	22.0	-96.0
Bob	1979	22.5	-95.3
Bob	1979	23.0	-94.6
Zeta	2005	23.9	-35.6
Zeta	2005	24.2	-36.1
Zeta	2005	24.7	-36.6

name	films
Luke	<chr [5]>
C-3PO	<chr [6]>
R2-D2	<chr[7]>

name	films
Luke	The Empire Strik...
Luke	Revenge of the S...
Luke	Return of the Jed...
C-3PO	The Empire Strik...
C-3PO	Attack of the Cl...
C-3PO	The Phantom M...
R2-D2	The Empire Strik...
R2-D2	Attack of the Cl...
R2-D2	The Phantom M...

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

```
starwars |>
  select(name, films) |>
  unnest_wider(films, names_sep = "_")
```

name	films
Luke	<chr [5]>
C-3PO	<chr [6]>
R2-D2	<chr[7]>

name	films_1	films_2	films_3
Luke	The Empire...	Revenge of...	Return of...
C-3PO	The Empire...	Attack of...	The Phantom...
R2-D2	The Empire...	Attack of...	The Phantom...

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)
```

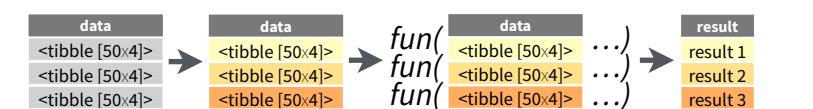
name	films
Luke	<chr [5]>
C-3PO	<chr [6]>
R2-D2	<chr[7]>

name	first_film	second_film	films
Luke	The Empire...	Revenge of...	<chr [3]>
C-3PO	The Empire...	Attack of...	<chr [4]>
R2-D2	The Empire...	Attack of...	<chr [5]>

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 transformation with dplyr :: CHEATSHEET



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



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



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

pipes

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

Summarize Cases

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



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

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

Group Cases

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

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

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

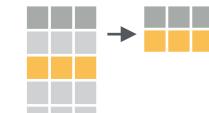
starwars |>
rowwise() |>
mutate(film_count = length(films))

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

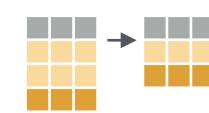
Manipulate Cases

EXTRACT CASES

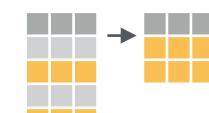
Row functions return a subset of rows as a new table.



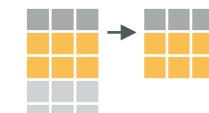
filter(.data, ..., .preserve = FALSE) Extract rows that meet logical criteria.
mtcars |> filter(mpg > 20)



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



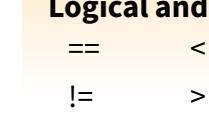
slice(.data, ..., .preserve = FALSE) Select rows by position.
mtcars |> slice(10:15)



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



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



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

Logical and boolean operators to use with filter()

<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>	

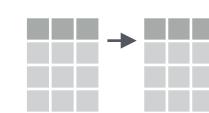
See [?base::Logic](#) and [?Comparison](#) for help.

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.
mtcars |> arrange(mpg)
mtcars |> arrange(desc(mpg))

ADD CASES



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

Manipulate Variables

EXTRACT VARIABLES

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



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



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



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

Use these helpers with select() and across()

e.g. mtcars |> select(mpg:cyl)

contains(match)

ends_with(match)

starts_with(match)

num_range(prefix, range)

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

! e.g., !gear

everything()

MANIPULATE MULTIPLE VARIABLES AT ONCE

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



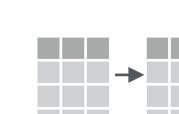
across(.cols, .funs, ..., .names = NULL) Summarize or mutate multiple columns in the same way.
df |> summarize(across(everything(), mean))



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

MAKE NEW VARIABLES

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



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



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



Vectorized Functions

TO USE WITH MUTATE ()

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

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 <= 1
dense_rank() - rank w ties = min, no gaps
min_rank() - rank with ties = min
ntile() - bins into n bins
percent_rank() - min_rank scaled to [0,1]
row_number() - rank with ties = "first"

MATH

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

MISCELLANEOUS

dplyr::case_when() - multi-case if_else()
starwars |>
 mutate(type = case_when(
 height > 200 | mass > 200 ~ "large",
 species == "Droid" ~ "robot",
 TRUE ~ "other"))

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

Summary Functions

TO USE WITH SUMMARIZE ()

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

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 TRUEs
sum() - # of TRUEs

ORDER

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

RANK

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

SPREAD

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

Row Names

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

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

A B C → A B tibble::column_to_rownames()
1 a t → 1 a Move col into row names.
2 b u → 2 b a |> column_to_rownames(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")
a t 1 3	Join matching values from y to x.

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

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

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

COLUMN MATCHING FOR JOINS

A B x C B y D	Use by = c("col1", "col2", ...) to specify one or more common columns to match on.
a t 1 t 3	left_join(x, y, by = "A")

A x B x C A y B y	Use a named vector, by = c("col1" = "col2") , to match on columns that have different names in each table.
a t 1 d w	left_join(x, y, by = c("C" = "D"))

A1 B1 C A2 B2	Use suffix to specify the suffix to give to unmatched columns that have the same name in both tables.
a t 1 d w	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).

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

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

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

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

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

A B C	nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, ...)
a t 1	<tibble [1x2]>
b u 2	<tibble [1x2]>
c v 3	<tibble [1x2]>

SET OPERATIONS

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



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



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



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

String manipulation with stringr :: CHEATSHEET



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

	str_detect(string, pattern, negate = FALSE) Detect the presence of a pattern match in a string. Also str_like() . str_detect(fruit, "a")
	str_starts(string, pattern, negate = FALSE) Detect the presence of a pattern match at the beginning of a string. Also str_ends() . str_starts(fruit, "a")
	str_which(string, pattern, negate = FALSE) Find the indexes of strings that contain a pattern match. str_which(fruit, "a")
	str_locate(string, pattern) Locate the positions of pattern matches in a string. Also str_locate_all() . str_locate(fruit, "a")
	str_count(string, pattern) Count the number of matches in a string. str_count(fruit, "a")

Subset Strings

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

Manage Lengths

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

Mutate Strings

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

Join and Split

	str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. str_c(letters, LETTERS)
	str_flatten(string, collapse = "") Combines into a single string, separated by collapse. str_flatten(fruit, ",")
	str_dup(string, times) Repeat strings times times. Also str_unique() to remove duplicates. str_dup(fruit, times = 2)
	str_split_fixed(string, pattern, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also str_split() to return a list of substrings and str_split_n() to return the nth substring. str_split_fixed(sentences, " ", n=3)
	str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. str_glue("Pi is {pi}")
	str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") 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

	str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)¹ Return the vector of indexes that sorts a character vector. fruit[str_order(fruit)]
	str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)¹ Sort a character vector. str_sort(fruit)

Helpers

	str_conv(string, encoding) Override the encoding of a string. str_conv(fruit, "ISO-8859-1")
	str_view(string, pattern, match = NA) View HTML rendering of all regex matches. str_view(sentences, "[aeiou])")
	str_equal(x, y, locale = "en", ignore_case = FALSE, ...)¹ Determine if two strings are equivalent. str_equal(c("a", "b"), c("a", "c"))
	str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. str_wrap(sentences, 20)

¹ See 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 (type this)	regexp (to mean this)	matches (which matches this)	example
a (etc.)	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 (\\$ for non-whitespaces)	see("\s")
\d	\d	any digit (\D for non-digits)	see("\d")
\w	\w	any word character (\W for non-word chars)	see("\w")
\b	\b	word boundaries	see("\b")
[:digit:] ¹	[:digit:] ¹	digits	see("[:digit:]")
[:alpha:] ¹	[:alpha:] ¹	letters	see("[:alpha:]")
[:lower:] ¹	[:lower:] ¹	lowercase letters	see("[:lower:]")
[:upper:] ¹	[:upper:] ¹	uppercase letters	see("[:upper:]")
[:alnum:] ¹	[:alnum:] ¹	letters and numbers	see("[:alnum:]")
[:punct:] ¹	[:punct:] ¹	punctuation	see("[:punct:]")
[:graph:] ¹	[:graph:] ¹	letters, numbers, and punctuation	see("[:graph:]")
[:space:] ¹	[:space:] ¹	space characters (i.e. \s)	see("[:space:]")
[:blank:] ¹	[:blank:] ¹	space and tab (but not new line)	see("[:blank:]")
.	.	every character except a new line	see(".")

¹ Many base R functions require classes to be wrapped in a second set of [], e.g. [[:digit:]]

ALTERNATES

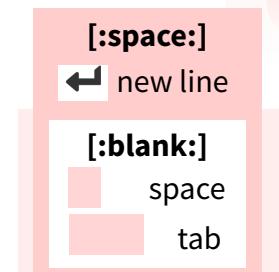
regexp	matches	example
ab d	or	alt("ab d")
[abe]	one of	alt("[abe]")
[^abe]	anything but	alt("[^abe]")
[a-c]	range	alt("[a-c]")

ANCHORS

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

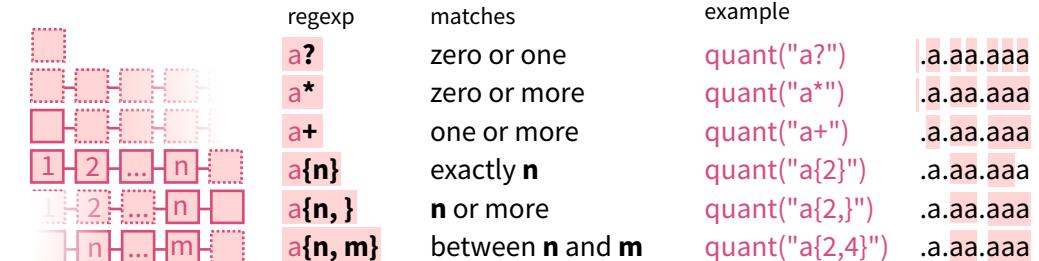
LOOK AROUNDS

regexp	matches	example
a(?=c)	followed by	look("a(?=c)")
a(?!c)	not followed by	look("a(?!c)")
(?<=b)a	preceded by	look("(?<=b)a")
(?<!b)a	not preceded by	look("(?<!b)a")



[:lower:]	[:upper:]
a	A
b	B
c	C
d	D
e	E
f	F
g	G
h	H
i	I
j	J
k	K
l	L
m	M
n	N
o	O
p	P
q	Q
r	R
s	S
t	T
u	U
v	V
w	W
x	X
y	Y
z	Z

QUANTIFIERS



GROUPS

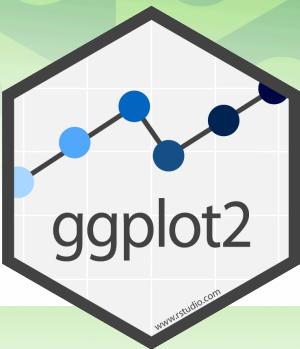
Use parentheses to set precedent (order of evaluation) and create groups

regexp	matches	example
(ab d)e	sets precedence	alt("(ab d)e")

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string	regexp	matches	example
(type this)	(to mean this)	(which matches this)	(the result is the same as ref("abba"))

Data visualization with ggplot2 :: CHEATSHEET



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: **GEOM_FUNCTION**, **mapping**, **stat**, **position**
Not required, sensible defaults supplied: **COORDINATE_FUNCTION**, **FACET_FUNCTION**, **SCALE_FUNCTION**, **THEME_FUNCTION**

`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")

size - integer (line width in mm)

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

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
⊗	□	○	△	◇	○	○	●	●	●	●	●	△△



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(f1))
```
- 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

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 <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))
```

- 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))
```

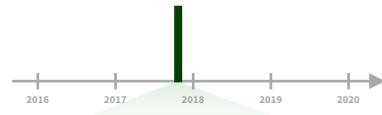
- k + geom_map(aes(map_id = state), map = map)**
+ **expand_limits(x = map\$long, y = map\$lat)**
map_id, alpha, color, fill, linetype, size

- 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

Dates and times with lubridate :: CHEATSHEET



Date-times



2017-11-28 12:00:00

2017-11-28 12:00:00

A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

```
dt <- as_datetime(1511870400)  
## "2017-11-28 12:00:00 UTC"
```

PARSE DATE-TIMES (Convert strings or numbers to date-times)

1. Identify the order of the year (**y**), month (**m**), day (**d**), hour (**h**), minute (**m**) and second (**s**) elements in your data.
2. Use the function below whose name replicates the order. Each accepts a tz argument to set the time zone, e.g. ymd(x, tz = "UTC").

2017-11-28T14:02:00

ymd_hms(), ymd_hm(), ymd_h().
ymd_hms("2017-11-28T14:02:00")

2017-22-12 10:00:00

ydm_hms(), ydm_hm(), ydm_h().
ydm_hms("2017-22-12 10:00:00")

11/28/2017 1:02:03

mdy_hms(), mdy_hm(), mdy_h().
mdy_hms("11/28/2017 1:02:03")

1 Jan 2017 23:59:59

dmy_hms(), dmy_hm(), dmy_h().
dmy_hms("1 Jan 2017 23:59:59")

20170131

ymd(), ydm(). ymd(20170131)

July 4th, 2000

mdy(), myd(). mdy("July 4th, 2000")

4th of July '99

dmy(), dym(). dmy("4th of July '99")

2001: Q3

yq() Q for quarter. yq("2001: Q3")

07-2020

my(), ym(). my("07-2020")

2:01

hms::hms() Also lubridate::hms(), hm() and ms(), which return periods.* hms::hms(seconds = 0, minutes = 1, hours = 2)

2017.5

date_decimal(decimal, tz = "UTC")
date_decimal(2017.5)



now(zone = "") Current time in tz
(defaults to system tz). now()



today(zone = "") Current date in a tz
(defaults to system tz). today()

fast.strptime() Faster strftime.
fast.strptime("9/1/01", "%y/%m/%d")

parse_date_time() Easier strftime.
parse_date_time("09-01-01", "ymd")

2017-11-28

A **date** is a day stored as the number of days since 1970-01-01

```
d <- as_date(17498)  
## "2017-11-28"
```

GET AND SET COMPONENTS

Use an accessor function to get a component. Assign into an accessor function to change a component in place.

2018-01-31 11:59:59

2018-01-31 11:59:59

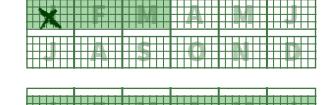
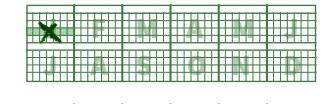
2018-01-31 11:59:59

2018-01-31 11:59:59

2018-01-31 11:59:59

2018-01-31 11:59:59

2018-01-31 11:59:59 UTC



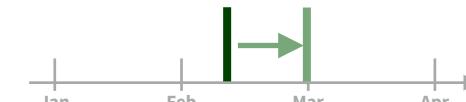
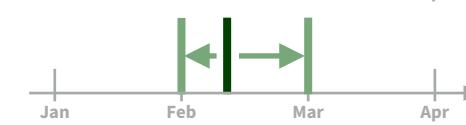
12:00:00

An hms is a **time** stored as the number of seconds since 00:00:00

```
t <- hms::as_hms(85)  
## 00:01:25
```

```
d ## "2017-11-28"  
day(d) ## 28  
day(d) <- 1  
d ## "2017-11-01"
```

Round Date-times



Valid units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear and year.

rollback(dates, roll_to_first = FALSE, preserve_hms = TRUE) Roll back to last day of previous month. Also rollforward(). rollback(dt)

Stamp Date-times

stamp() Derive a template from an example string and return a new function that will apply the template to date-times. Also stamp_date() and stamp_time().

1. Derive a template, create a function
`sf <- stamp("Created Sunday, Jan 17, 1999 3:34")`
2. Apply the template to dates
`sf(ymd("2010-04-05"))
[1] "Created Monday, Apr 05, 2010 00:00"`

Tip: use a date with day > 12

Time Zones

R recognizes ~600 time zones. Each encodes the time zone, Daylight Savings Time, and historical calendar variations for an area. R assigns one time zone per vector.

Use the **UTC** time zone to avoid Daylight Savings.

OlsonNames() Returns a list of valid time zone names. OlsonNames()

Sys.timezone() Gets current time zone.



with_tz(time, tzzone = "") Get the same date-time in a new time zone (a new clock time). Also **local_time(dt, tz, units)**. **with_tz(dt, "US/Pacific")**



force_tz(time, tzzone = "") Get the same clock time in a new time zone (a new date-time). Also **force_tzs()**. **force_tz(dt, "US/Pacific")**



Math with Date-times

Math with date-times relies on the **timeline**, which behaves inconsistently. Consider how the timeline behaves during:

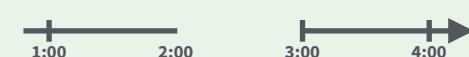
A normal day

```
nor <- ymd_hms("2018-01-01 01:30:00",tz="US/Eastern")
```



The start of daylight savings (spring forward)

```
gap <- ymd_hms("2018-03-11 01:30:00",tz="US/Eastern")
```



The end of daylight savings (fall back)

```
lap <- ymd_hms("2018-11-04 00:30:00",tz="US/Eastern")
```



Leap years and leap seconds

```
leap <- ymd("2019-03-01")
```

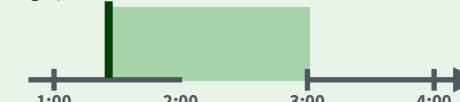


Periods track changes in clock times, which ignore time line irregularities.

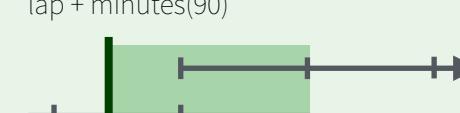
nor + minutes(90)



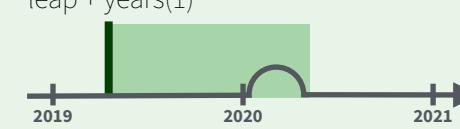
gap + minutes(90)



lap + minutes(90)

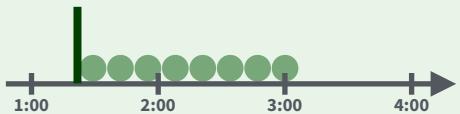


leap + years(1)



Durations track the passage of physical time, which deviates from clock time when irregularities occur.

nor + dminutes(90)



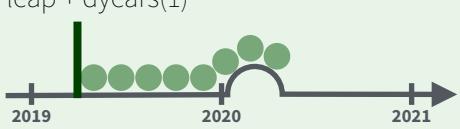
gap + dminutes(90)



lap + dminutes(90)

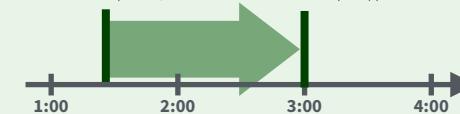


leap + dyears(1)



Intervals represent specific intervals of the timeline, bounded by start and end date-times.

interval(nor, nor + minutes(90))



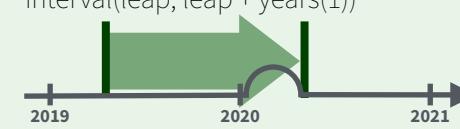
interval(gap, gap + minutes(90))



interval(lap, lap + minutes(90))



interval(leap, leap + years(1))



Not all years are 365 days due to **leap days**.

Not all minutes are 60 seconds due to **leap seconds**.

It is possible to create an imaginary date by adding **months**, e.g. February 31st

```
jan31 <- ymd(20180131)
jan31 + months(1)
## "NA"
```

%m+% and %m-% will roll imaginary dates to the last day of the previous month.

```
jan31 %m+% months(1)
## "2018-02-28"
```

add_with_rollback(e1, e2, roll_to_first = TRUE) will roll imaginary dates to the first day of the new month.

```
add_with_rollback(jan31, months(1),
roll_to_first = TRUE)
## "2018-03-01"
```

PERIODS

Add or subtract periods to model events that happen at specific clock times, like the NYSE opening bell.

Make a period with the name of a time unit **pluralized**, e.g.

```
p <- months(3) + days(12)
```

p

"3m 12d 0H 0M 0S"

Number of months Number of days etc.

years(x = 1) x years.

months(x) x months.

weeks(x = 1) x weeks.

days(x = 1) x days.

hours(x = 1) x hours.

minutes(x = 1) x minutes.

seconds(x = 1) x seconds.

milliseconds(x = 1) x milliseconds.

microseconds(x = 1) x microseconds.

nanoseconds(x = 1) x nanoseconds.

picoseconds(x = 1) x picoseconds.

period(num = NULL, units = "second", ...)

An automation friendly period constructor.
period(5, unit = "years")

as.period(x, unit) Coerce a timespan to a period, optionally in the specified units. Also **is.period()**. as.period(p)

period_to_seconds(x) Convert a period to the "standard" number of seconds implied by the period. Also **seconds_to_period()**. period_to_seconds(p)

DURATIONS

Add or subtract durations to model physical processes, like battery life. Durations are stored as seconds, the only time unit with a consistent length.

Diftimes are a class of durations found in base R.

Make a duration with the name of a period prefixed with a **d**, e.g.

```
dd <- ddays(14)
```

dd

"1209600s (~2 weeks)"

Exact length in seconds Equivalent in common units

dyears(x = 1) 31536000x seconds.

dmonths(x = 1) 2629800x seconds.

dweeks(x = 1) 604800x seconds.

ddays(x = 1) 86400x seconds.

dhours(x = 1) 3600x seconds.

dminutes(x = 1) 60x seconds.

dseconds(x = 1) x seconds.

dmilliseconds(x = 1) x × 10⁻³ seconds.

dmicroseconds(x = 1) x × 10⁻⁶ seconds.

dnanoseconds(x = 1) x × 10⁻⁹ seconds.

dpicoseconds(x = 1) x × 10⁻¹² seconds.

duration(num = NULL, units = "second", ...)

An automation friendly duration constructor. duration(5, unit = "years")

as.duration(x, ...) Coerce a timespan to a duration. Also **is.duration()**, **is.difftime()**. as.duration(i)

make_difftime(x) Make difftime with the specified number of units. make_difftime(99999)

INTERVALS

Divide an interval by a duration to determine its physical length, divide an interval by a period to determine its implied length in clock time.

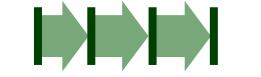
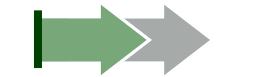
Make an interval with **interval()** or %--%, e.g.

```
i <- interval(ymd("2017-01-01"), d)
```

```
## 2017-01-01 UTC--2017-11-28 UTC
```

```
j <- d %--% ymd("2017-12-31")
```

```
## 2017-11-28 UTC--2017-12-31 UTC
```



Start Date End Date

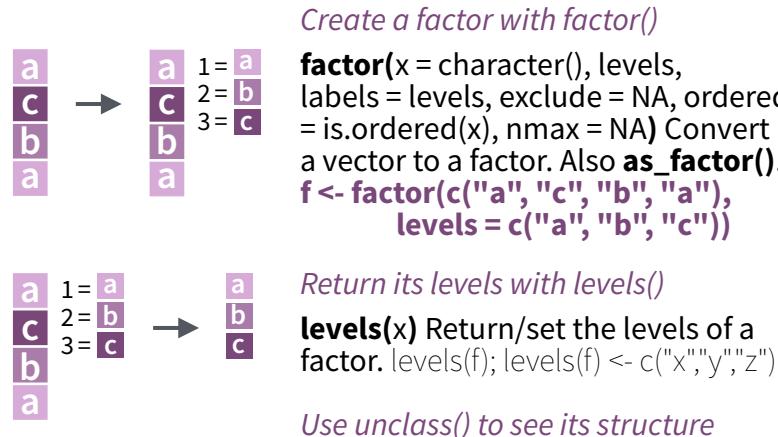


Factors withforcats :: CHEATSHEET

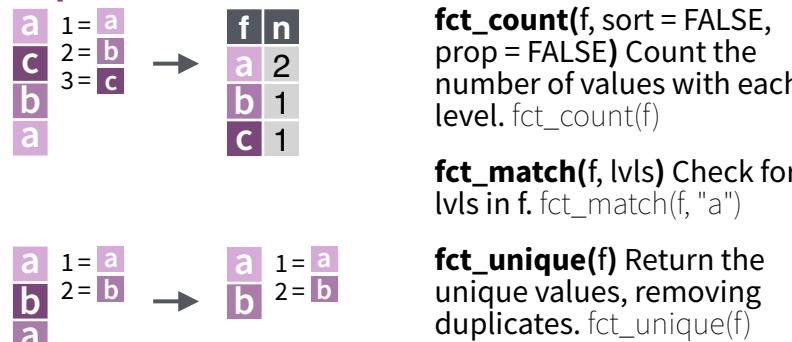
The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors

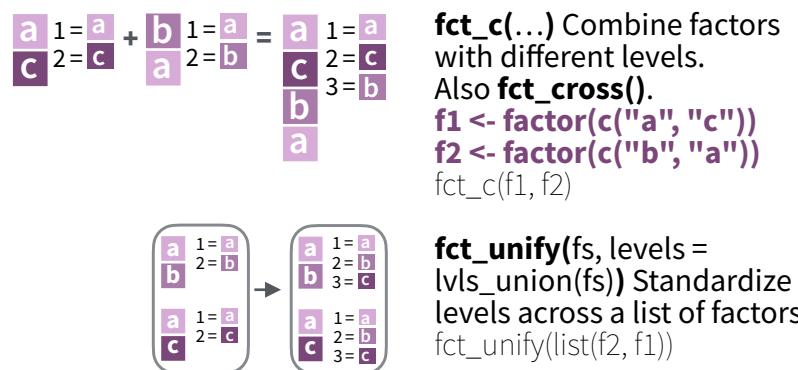
R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the levels associated with them.



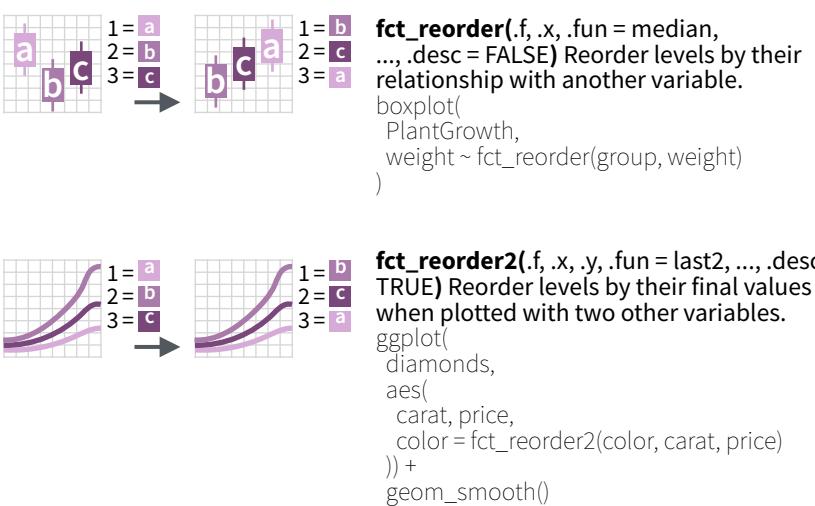
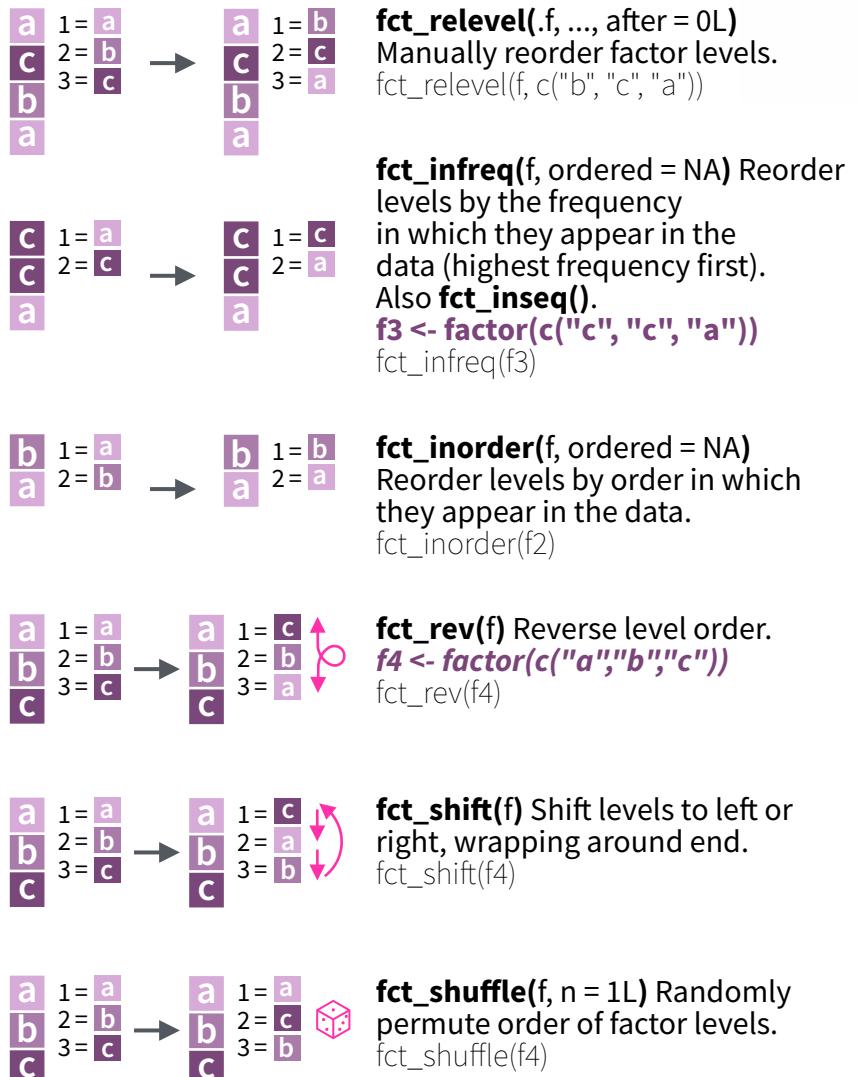
Inspect Factors



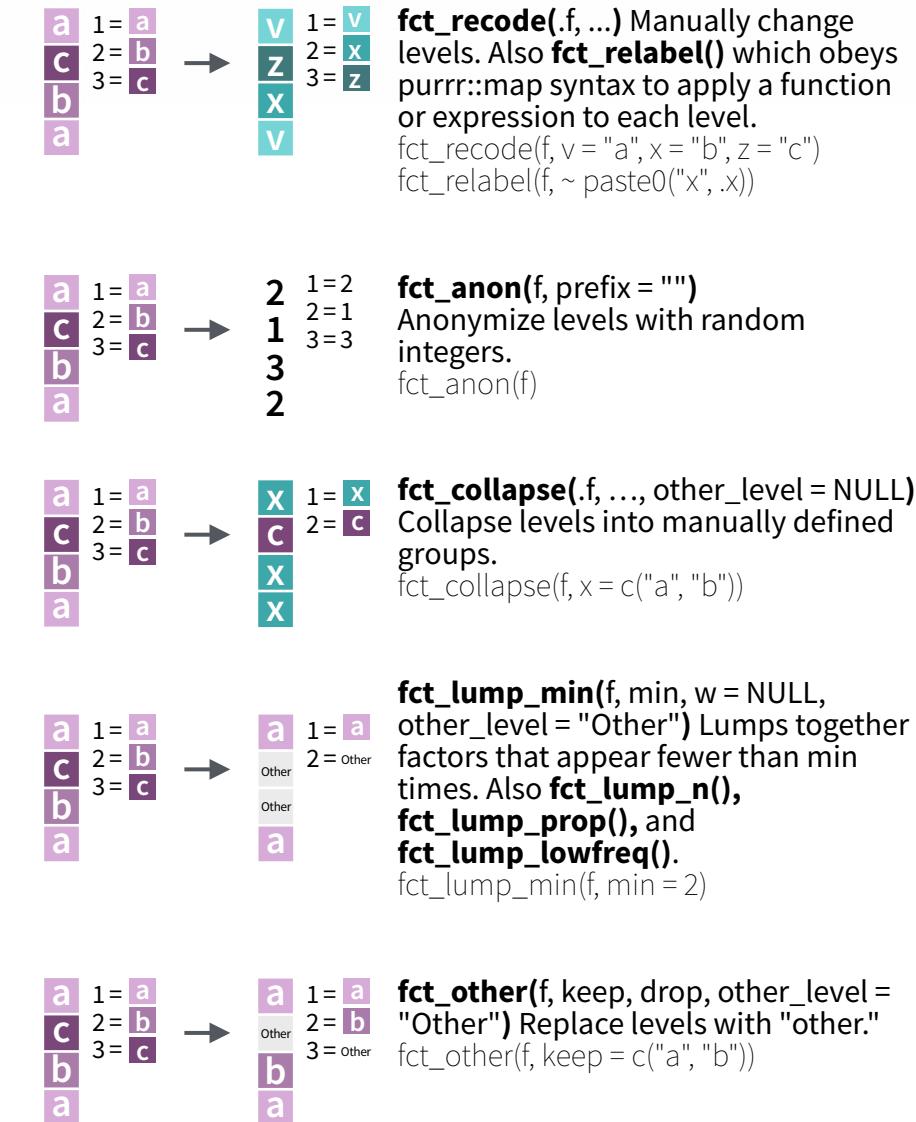
Combine Factors



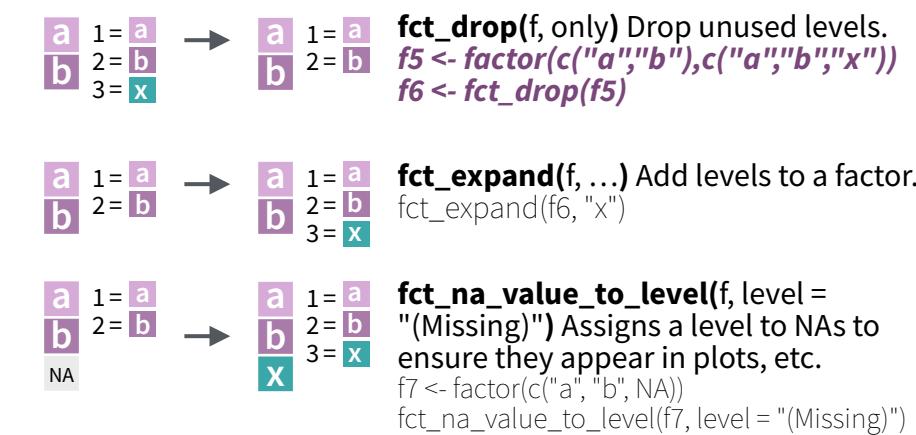
Change the order of levels



Change the value of levels



Add or drop levels



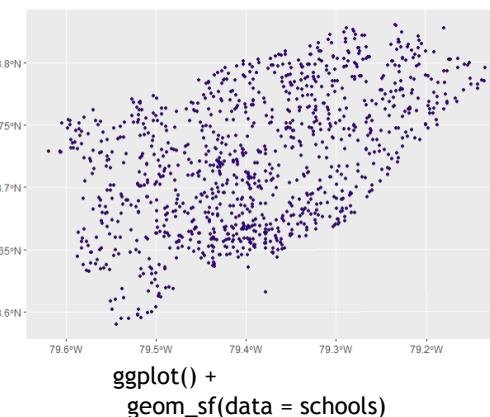
Spatial manipulation with sf: : CHEAT SHEET



The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.

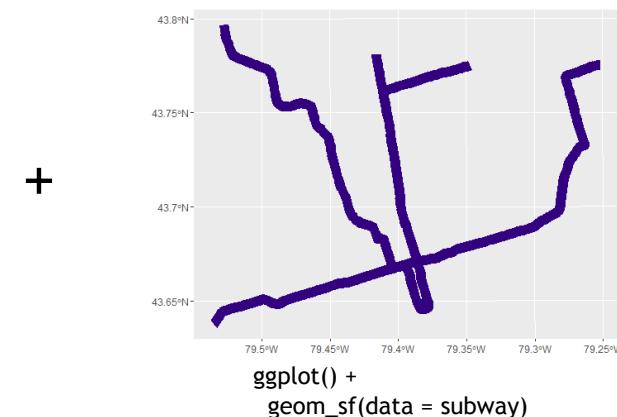
Geometric confirmation

- st_contains(x, y, ...) Identifies if y is within x (i.e. point within polygon)
- st_covered_by(x, y, ...) Identifies if x is completely within y (i.e. polygon completely within polygon)
- st_covers(x, y, ...) Identifies if any point from x is outside of y (i.e. polygon outside polygon)
- st_crosses(x, y, ...) Identifies if any geometry of x have commonalities with y
- st_disjoint(x, y, ...) Identifies when geometries from x do not share space with y
- st_equals(x, y, ...) Identifies if x and y share the same geometry
- st_intersects(x, y, ...) Identifies if x and y geometry share any space
- st_overlaps(x, y, ...) Identifies if geometries of x and y share space, are of the same dimension, but are not completely contained by each other
- st_touches(x, y, ...) Identifies if geometries of x and y share a common point but their interiors do not intersect
- st_within(x, y, ...) Identifies if x is in a specified distance to y



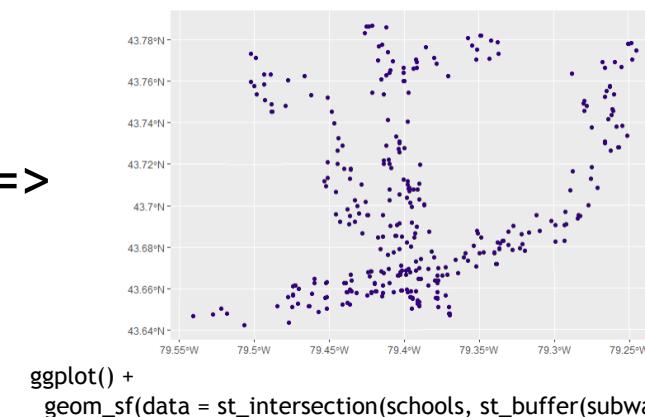
Geometric operations

- st_boundary(x) Creates a polygon that encompasses the full extent of the geometry
- st_buffer(x, dist, nQuadSegs) Creates a polygon covering all points of the geometry within a given distance
- st_centroid(x, ..., of_largest_polygon) Creates a point at the geometric centre of the geometry
- st_convex_hull(x) Creates geometry that represents the minimum convex geometry of x
- st_line_merge(x) Creates linestring geometry from sewing multi linestring geometry together
- st_node(x) Creates nodes on overlapping geometry where nodes do not exist
- st_point_on_surface(x) Creates a point that is guaranteed to fall on the surface of the geometry
- st_polygonize(x) Creates polygon geometry from linestring geometry
- st_segmentize(x, dfMaxLength, ...) Creates linestring geometry from x based on a specified length
- st_simplify(x, preserveTopology, dTolerance) Creates a simplified version of the geometry based on a specified tolerance



Geometry creation

- st_triangulate(x, dTolerance, bOnlyEdges) Creates polygon geometry as triangles from point geometry
- st_voronoi(x, envelope, dTolerance, bOnlyEdges) Creates polygon geometry covering the envelope of x, with x at the centre of the geometry
- st_point(x, c(numeric vector), dim = "XYZ") Creating point geometry from numeric values
- st_multipoint(x = matrix(numeric values in rows), dim = "XYZ") Creating multi point geometry from numeric values
- st_linestring(x = matrix(numeric values in rows), dim = "XYZ") Creating linestring geometry from numeric values
- st_multilinestring(x = list(numeric matrices in rows), dim = "XYZ") Creating multi linestring geometry from numeric values
- st_polygon(x = list(numeric matrices in rows), dim = "XYZ") Creating polygon geometry from numeric values
- st_multipolygon(x = list(numeric matrices in rows), dim = "XYZ") Creating multi polygon geometry from numeric values



Spatial manipulation with sf: : CHEAT SHEET



The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.

Geometry operations

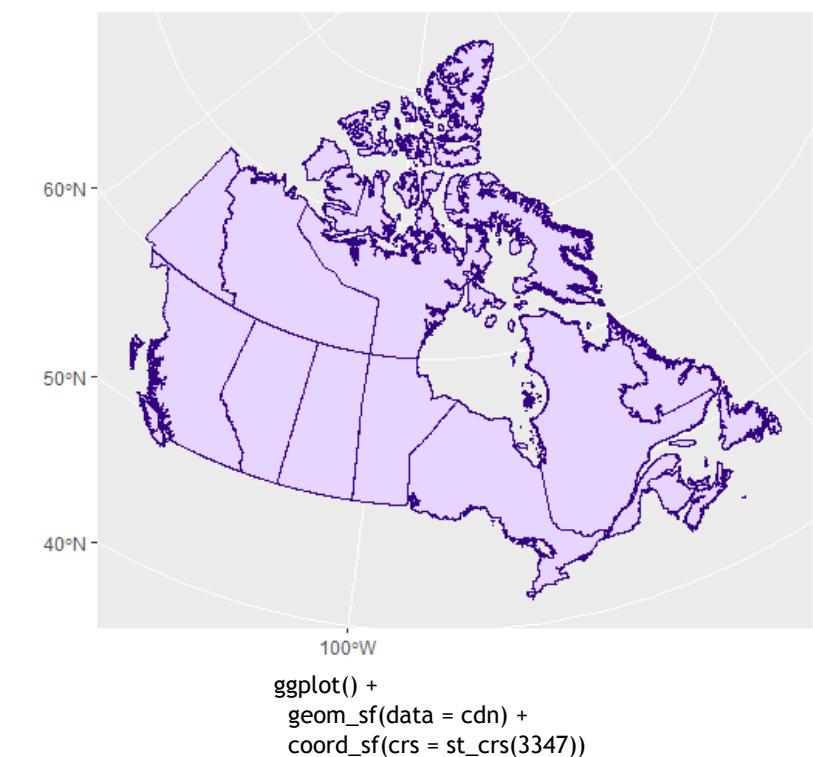
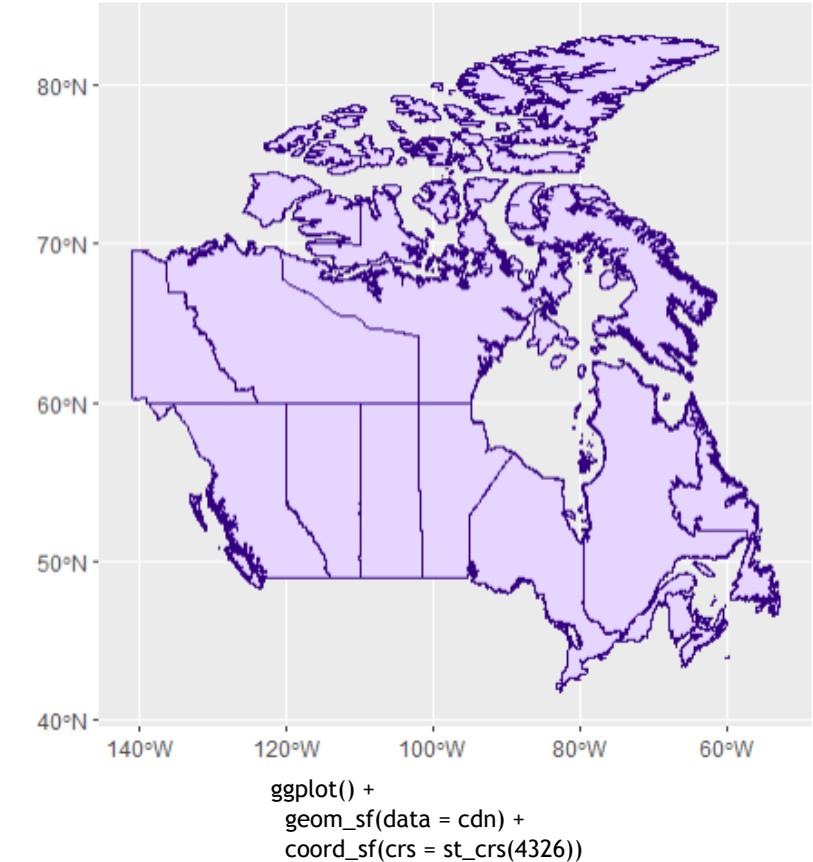
- ↳ `st_contains(x, y, ...)` Identifies if y is within x (i.e. point within polygon)
- ↳ `st_crop(x, y, ..., xmin, ymin, xmax, ymax)` Creates geometry of x that intersects a specified rectangle
- ↳ `st_difference(x, y)` Creates geometry from x that does not intersect with y
- ↳ `st_intersection(x, y)` Creates geometry of the shared portion of x and y
- ↳ `st_sym_difference(x, y)` Creates geometry representing portions of x and y that do not intersect
- ↳ `st_snap(x, y, tolerance)` Snap nodes from geometry x to geometry y
- ↳ `st_union(x, y, ..., by_feature)` Creates multiple geometries into a single geometry, consisting of all geometry elements

Geometric measurement

- `st_area(x)` Calculate the surface area of a polygon geometry based on the current coordinate reference system
- `st_distance(x, y, ..., dist_fun, by_element, which)` Calculates the 2D distance between x and y based on the current coordinate system
- `st_length(x)` Calculates the 2D length of a geometry based on the current coordinate system

Misc operations

- ↳ `st_as_sf(x, ...)` Create a sf object from a non-geospatial tabular data frame
- ↳ `st_cast(x, to, ...)` Change x geometry to a different geometry type
- ↳ `st_coordinates(x, ...)` Creates a matrix of coordinate values from x
- ↳ `st_crs(x, ...)` Identifies the coordinate reference system of x
- ↳ `st_join(x, y, join, FUN, suffix, ...)` Performs a spatial left or inner join between x and y
- ↳ `st_make_grid(x, cellsize, offset, n, crs, what)` Creates rectangular grid geometry over the bounding box of x
- ↳ `st_nearest_feature(x, y)` Creates an index of the closest feature between x and y
- ↳ `st_nearest_points(x, y, ...)` Returns the closest point between x and y
- ↳ `st_read(dsn, layer, ...)` Read file or database vector dataset as a sf object
- ↳ `st_transform(x, crs, ...)` Convert coordinates of x to a different coordinate reference system



brown4	darkorange4	gray	gray57	hotpink3	lightsalmon4	navajowhite1	plum3	slategray3	antiquewhite
brown3	darkorange3	goldenrod4	gray56	hotpink2	lightsalmon3	navajowhite	plum2	slategray2	aliceblue
brown2	darkorange2	goldenrod3	gray55	hotpink1	lightsalmon2	moccasin	plum1	slategray1	white
brown1	darkorange1	goldenrod2	gray54	hotpink	lightsalmon1	mistyrose4	plum	slategray	yellowgreen
brown	darkorange	goldenrod1	gray53	honeydew4	lightsalmon	mistyrose3	pink4	slateblue4	yellow4
blueviolet	darkolivegreen4	goldenrod	gray52	honeydew3	lightpink4	mistyrose2	pink3	slateblue3	yellow3
blue4	darkolivegreen3	gold4	gray51	honeydew2	lightpink3	mistyrose1	pink2	slateblue2	yellow2
blue3	darkolivegreen2	gold3	gray50	honeydew1	lightpink2	mistyrose	pink1	slateblue1	yellow1
blue2	darkolivegreen1	gold2	gray49	honeydew	lightpink1	mintcream	pink	slateblue	yellow
blue1	darkolivegreen	gold1	gray48	greenyellow	lightpink	midnightblue	peru	skyblue4	whitesmoke
blue	darkmagenta	gold	gray47	green4	lightgrey	mediumvioletred	peachpuff4	skyblue3	wheat4
blanchedalmond	darkkhaki	ghostwhite	gray46	green3	lightgreen	mediumturquoise	peachpuff3	skyblue2	wheat3
black	darkgrey	gainsboro	gray45	green2	lightgray	mediumspringgreen	peachpuff2	skyblue1	wheat2
bisque4	darkgreen	forestgreen	gray44	green1	lightgoldenrodyellow	mediumslateblue	peachpuff1	skyblue	wheat1
bisque3	darkgray	floralwhite	gray43	green	lightgoldenrod4	mediumseagreen	peachpuff	sienna4	wheat
bisque2	darkgoldenrod4	firebrick4	gray42	gray100	lightgoldenrod3	mediumpurple4	papayawhip	sienna3	violetred4
bisque1	darkgoldenrod3	firebrick3	gray41	gray99	lightgoldenrod2	mediumpurple3	palevioletred4	sienna2	violetred3
bisque	darkgoldenrod2	firebrick2	gray40	gray98	lightgoldenrod1	mediumpurple2	palevioletred3	sienna1	violetred2
beige	darkgoldenrod1	firebrick1	gray39	gray97	lightgoldenrod	mediumpurple1	palevioletred2	sienna	violetred1
azure4	darkgoldenrod	firebrick	gray38	gray96	lightcyan4	mediumpurple	palevioletred1	seashell4	violetred
azure3	darkcyan	dodgerblue4	gray37	gray95	lightcyan3	mediumorchid4	palevioletred	seashell3	violet
azure2	darkblue	dodgerblue3	gray36	gray94	lightcyan2	mediumorchid3	paleturquoise4	seashell2	turquoise4
azure1	cyan4	dodgerblue2	gray35	gray93	lightcyan1	mediumorchid2	paleturquoise3	seashell1	turquoise3
azure	cyan3	dodgerblue1	gray34	gray92	lightcyan	mediumorchid1	paleturquoise2	seashell	turquoise2
aquamarine4	cyan2	dodgerblue	gray33	gray91	lightcoral	mediumorchid	paleturquoise1	seagreen4	turquoise1
aquamarine3	cyan1	dimgray	gray32	gray90	lightblue4	mediumblue	paleturquoise	seagreen3	turquoise
aquamarine2	cyan	dimgray	gray31	gray89	lightblue3	mediumaquamarine	palegreen4	seagreen2	tomato4
aquamarine1	cornsilk4	deepskyblue4	gray30	gray88	lightblue2	maroon4	palegreen3	seagreen1	tomato3
aquamarine	cornsilk3	deepskyblue3	gray29	gray87	lightblue1	maroon3	palegreen2	seagreen	tomato2
antiquewhite4	cornsilk2	deepskyblue2	gray28	gray86	lightblue	maroon2	palegreen1	sandybrown	tomato1
antiquewhite3	cornsilk1	deepskyblue1	gray27	gray85	lemonchiffon4	maroon1	palegreen	salmon4	tomato
antiquewhite2	cornsilk	deepskyblue	gray26	gray84	lemonchiffon3	maroon	palegoldenrod	salmon3	thistle4
antiquewhite1	cornflowerblue	deeppink4	gray25	gray83	lemonchiffon2	magenta4	orchid4	salmon2	thistle3
antiquewhite	coral4	deeppink3	gray24	gray82	lemonchiffon1	magenta3	orchid3	salmon1	thistle2
aliceblue	coral3	deeppink2	gray23	gray81	lemonchiffon	magenta2	orchid2	salmon	thistle1
white	coral2	deeppink1	gray22	gray80	lawngreen	magenta1	orchid1	saddlebrown	thistle
bisque3	coral1	deeppink	gray21	gray79	lavenderblush4	magenta	orchid	royalblue4	tan4
bisque2	coral	darkviolet	gray20	gray78	lavenderblush3	linen	orangered4	royalblue3	tan3
bisque1	chocolate4	darkturquoise	gray19	gray77	lavenderblush2	limegreen	orangered3	royalblue2	tan2
bisque	chocolate3	darkslategray	gray18	gray76	lavenderblush1	lightyellow4	orangered2	royalblue1	tan1
beige	chocolate2	darkslategray4	gray17	gray75	lavenderblush	lightyellow3	orangered1	royalblue	tan
azure4	chocolate1	darkslategray3	gray16	gray74	lavender	lightyellow2	orangered	rosybrown4	steelblue4
azure3	chocolate	darkslategray2	gray15	gray73	khaki4	lightyellow1	orange4	rosybrown3	steelblue3
azure2	chartreuse4	darkslategray1	gray14	gray72	khaki3	lightyellow	orange3	rosybrown2	steelblue2
azure1	chartreuse3	darkslategray	gray13	gray71	khaki2	lightsteelblue4	orange2	rosybrown1	steelblue1
azure	chartreuse2	darkslateblue	gray12	gray70	khaki1	lightsteelblue3	orange1	rosybrown	steelblue
aquamarine4	chartreuse1	darkseagreen4	gray11	gray69	khaki	lightsteelblue2	orange	red4	springgreen4
aquamarine3	chartreuse	darkseagreen3	gray10	gray68	ivory4	lightsteelblue1	olivedrab4	red3	springgreen3
aquamarine2	cadetblue4	darkseagreen2	gray9	gray67	ivory3	lightsteelblue	olivedrab3	red2	springgreen2
aquamarine1	cadetblue3	darkseagreen1	gray8	gray66	ivory2	lightslategrey	olivedrab2	red1	springgreen1
aquamarine	cadetblue2	darkseagreen	gray7	gray65	ivory1	lightslategray	olivedrab1	red	springgreen
antiquewhite4	cadetblue1	darksalmon	gray6	gray64	ivory	lightslateblue	olivedrab	purple4	snow4
antiquewhite3	cadetblue	darkred	gray5	gray63	indianred4	lightskyblue4	oldlace	purple3	snow3
antiquewhite2	burlywood4	darkorchid4	gray4	gray62	indianred3	lightskyblue3	navyblue	purple2	snow2
antiquewhite1	burlywood3	darkorchid3	gray3	gray61	indianred2	lightskyblue2	navy	purple1	snow1
antiquewhite	burlywood2	darkorchid2	gray2	gray60	indianred1	lightskyblue1	navajowhite4	purple	snow
aliceblue	burlywood1	darkorchid1	gray1	gray59	indianred	lightskyblue	navajowhite3	powderblue	slategrey
white	burlywood	darkorchid	gray0	gray58	hotpink4	lightseagreen	navajowhite2	plum4	slategray4