

RStudio IDE :: CHEAT SHEET



Documents and Apps

Open Shiny, R Markdown, knitr, Sweave, LaTeX, .R files and more in Source Pane

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

Access markdown guide at **Help > Markdown Quick Reference**

Jump to chunk, Set knitr chunk options, Run this and all previous code chunks, Run this code chunk

```

17 {r pressure, echo=FALSE}
18 plot(pressure)
19
20
    
```

RStudio recognizes that files named **app.R**, **server.R**, **ui.R**, and **global.R** belong to a shiny app

Run app, Choose location to view app, Publish to shinyapps.io or server, Manage publish accounts

Write Code

Navigate tabs, Open in new window, Save, Find and replace, Compile as notebook, Run selected code

Multiple cursors/column selection with **Alt + mouse drag**

Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.

Syntax highlighting based on your file's extension

Tab completion to finish function names, file paths, arguments, and more.

Multi-language code snippets to quickly use common blocks of code.

Jump to function in file, Change file type

Working Directory, Maximize, minimize panes, Press **↑** to see command history, Drag pane boundaries

Environment pane: Load workspace, Save workspace, Delete all saved objects, Search inside environment, Choose environment to display from list of parent environments, Display objects as list or grid

Files pane: Create folder, Upload file, Delete file, Rename file, Change directory, Path to displayed directory, A File browser keyed to your working directory. Click on file or directory name to open.

R Support

Import data with wizard, History of past commands to run/copy, Display .RPres slideshows **File > New File > R Presentation**

Pro Features

Share Project with Collaborators, Active shared collaborators, Start **new R Session** in current project, Close R Session in project, **Select R Version**

PROJECT SYSTEM
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.

RStudio opens plots in a dedicated Plots pane

Navigate recent plots, Open in window, **Export plot**, Delete plot, Delete all plots

GUI Package manager lists every installed package

Install Packages, Update Packages, Create reproducible package library for your project

Click to load package with **library()**. Unclick to detach package with **detach()**

Package version installed, Delete from library

RStudio opens documentation in a dedicated Help pane

Home page of helpful links, Search within help file, Search for help file

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

Stop Shiny app, Publish to shinyapps.io, rpubs, RSCONNECT, ..., Refresh

View(<data>) opens spreadsheet like view of data set

Filter	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
All	All	All	All	All	All
1	5.1	3.5	1.4	0.2	setosa
2					
3	Filter rows by value or value range		Sort by values		Search for value
4					

Debug Mode

Open with **debug()**, **browse()**, or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

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

Step through code one line at a time

Step into and out of functions to run

Resume execution mode

Quit debug mode

Launch debugger from origin of error, Open traceback to examine the functions that R called before the error occurred

```

> foo()
Error in get_digit(num, x) :
Error!
    
```

Next, Continue, Stop

Version Control with Git or SVN

Turn on at **Tools > Project Options > Git/SVN**

Stage files, Show file diff, Commit staged files, Push/Pull to remote, View History

Added, Deleted, Modified, Renamed, Untracked

Open shell to type commands, current branch

Package Writing

File > New Project > New Directory > R Package

Turn project into package, Enable roxygen documentation with **Tools > Project Options > Build Tools**

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

Build & Reload, Check, More...

Load All, Clean and Rebuild, Test Package, Check Package, Build Source Package, Build Binary Package, Document, Configure Build Tools...





1 LAYOUT

	Windows/Linux	Mac
Move focus to Source Editor	Ctrl+1	Ctrl+1
Move focus to Console	Ctrl+2	Ctrl+2
Move focus to Help	Ctrl+3	Ctrl+3
Show History	Ctrl+4	Ctrl+4
Show Files	Ctrl+5	Ctrl+5
Show Plots	Ctrl+6	Ctrl+6
Show Packages	Ctrl+7	Ctrl+7
Show Environment	Ctrl+8	Ctrl+8
Show Git/SVN	Ctrl+9	Ctrl+9
Show Build	Ctrl+0	Ctrl+0

2 RUN CODE

	Windows/Linux	Mac
Search command history	Ctrl+↑	Cmd+↑
Navigate command history	↑/↓	↑/↓
Move cursor to start of line	Home	Cmd+←
Move cursor to end of line	End	Cmd+→
Change working directory	Ctrl+Shift+H	Ctrl+Shift+H
Interrupt current command	Esc	Esc
Clear console	Ctrl+L	Ctrl+L
Quit Session (desktop only)	Ctrl+Q	Cmd+Q
Restart R Session	Ctrl+Shift+F10	Cmd+Shift+F10
Run current line/selection	Ctrl+Enter	Cmd+Enter
Run current (retain cursor)	Alt+Enter	Option+Enter
Run from current to end	Ctrl+Alt+E	Cmd+Option+E
Run the current function	Ctrl+Alt+F	Cmd+Option+F
Source a file	Ctrl+Alt+G	Cmd+Option+G
Source the current file	Ctrl+Shift+S	Cmd+Shift+S
Source with echo	Ctrl+Shift+Enter	Cmd+Shift+Enter

3 NAVIGATE CODE

	Windows /Linux	Mac
Goto File/Function	Ctrl+.	Ctrl+.
Fold Selected	Alt+L	Cmd+Option+L
Unfold Selected	Shift+Alt+L	Cmd+Shift+Option+L
Fold All	Alt+O	Cmd+Option+O
Unfold All	Shift+Alt+O	Cmd+Shift+Option+O
Go to line	Shift+Alt+G	Cmd+Shift+Option+G
Jump to	Shift+Alt+J	Cmd+Shift+Option+J
Switch to tab	Ctrl+Shift+.	Ctrl+Shift+.
Previous tab	Ctrl+F11	Ctrl+F11
Next tab	Ctrl+F12	Ctrl+F12
First tab	Ctrl+Shift+F11	Ctrl+Shift+F11
Last tab	Ctrl+Shift+F12	Ctrl+Shift+F12
Navigate back	Ctrl+F9	Cmd+F9
Navigate forward	Ctrl+F10	Cmd+F10
Jump to Brace	Ctrl+P	Ctrl+P
Select within Braces	Ctrl+Shift+Alt+E	Ctrl+Shift+Option+E
Use Selection for Find	Ctrl+F3	Cmd+E
Find in Files	Ctrl+Shift+F	Cmd+Shift+F
Find Next	Win: F3, Linux: Ctrl+G	Cmd+G
Find Previous	W: Shift+F3, L:	Cmd+Shift+G
Jump to Word	Ctrl+↔	Option+↔
Jump to Start/End	Ctrl+↑/↓	Cmd+↑/↓
Toggle Outline	Ctrl+Shift+O	Cmd+Shift+O

4 WRITE CODE

	Windows /Linux	Mac
Attempt completion	Tab or Ctrl+Space	Tab or Cmd+Space
Navigate candidates	↑/↓	↑/↓
Accept candidate	Enter, Tab, or →	Enter, Tab, or →
Dismiss candidates	Esc	Esc
Undo	Ctrl+Z	Cmd+Z
Redo	Ctrl+Shift+Z	Cmd+Shift+Z
Cut	Ctrl+X	Cmd+X
Copy	Ctrl+C	Cmd+C
Paste	Ctrl+V	Cmd+V
Select All	Ctrl+A	Cmd+A
Delete Line	Ctrl+D	Cmd+D
Select	Shift+[Arrow]	Shift+[Arrow]
Select Word	Ctrl+Shift+↔	Option+Shift+↔
Select to Line Start	Alt+Shift+←	Cmd+Shift+←
Select to Line End	Alt+Shift+→	Cmd+Shift+→
Select Page Up/Down	Shift+PageUp/Down	Shift+PageUp/Down
Select to Start/End	Shift+Alt+↑/↓	Cmd+Shift+↑/↓
Delete Word Left	Ctrl+Backspace	Ctrl+Opt+Backspace
Delete Word Right		Option+Delete
Delete to Line End		Ctrl+K
Delete to Line Start		Option+Backspace
Indent	Tab (at start of line)	Tab (at start of line)
Outdent	Shift+Tab	Shift+Tab
Yank line up to cursor	Ctrl+U	Ctrl+U
Yank line after cursor	Ctrl+K	Ctrl+K
Insert yanked text	Ctrl+Y	Ctrl+Y
Insert <-	Alt+-	Option+-
Insert %>%	Ctrl+Shift+M	Cmd+Shift+M
Show help for function	F1	F1
Show source code	F2	F2
New document	Ctrl+Shift+N	Cmd+Shift+N
New document (Chrome)	Ctrl+Alt+Shift+N	Cmd+Shift+Opt+N
Open document	Ctrl+O	Cmd+O
Save document	Ctrl+S	Cmd+S
Close document	Ctrl+W	Cmd+W
Close document (Chrome)	Ctrl+Alt+W	Cmd+Option+W
Close all documents	Ctrl+Shift+W	Cmd+Shift+W
Extract function	Ctrl+Alt+X	Cmd+Option+X
Extract variable	Ctrl+Alt+V	Cmd+Option+V
Reindent lines	Ctrl+I	Cmd+I
(Un)Comment lines	Ctrl+Shift+C	Cmd+Shift+C
Reflow Comment	Ctrl+Shift+/	Cmd+Shift+/
Reformat Selection	Ctrl+Shift+A	Cmd+Shift+A
Select within braces	Ctrl+Shift+E	Ctrl+Shift+E
Show Diagnostics	Ctrl+Shift+Alt+P	Cmd+Shift+Opt+P
Transpose Letters		Ctrl+T
Move Lines Up/Down	Alt+↑/↓	Option+↑/↓
Copy Lines Up/Down	Shift+Alt+↑/↓	Cmd+Option+↑/↓
Add New Cursor Above	Ctrl+Alt+Up	Ctrl+Option+Up
Add New Cursor Below	Ctrl+Alt+Down	Ctrl+Option+Down
Move Active Cursor Up	Ctrl+Alt+Shift+Up	Ctrl+Option+Shift+Up
Move Active Cursor Down	Ctrl+Alt+Shift+Down	Ctrl+Opt+Shift+Down
Find and Replace	Ctrl+F	Cmd+F
Use Selection for Find	Ctrl+F3	Cmd+E
Replace and Find	Ctrl+Shift+J	Cmd+Shift+J

WHY RSTUDIO SERVER PRO?

RSP extends the 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
- edit the same project at the same time as others
- see what you and others are doing on your server
- switch easily from one version of R to a different version
- integrate with your authentication, authorization, and audit practices

Download a free 45 day evaluation at

www.rstudio.com/products/rstudio-server-pro/

5 DEBUG CODE

	Windows/Linux	Mac
Toggle Breakpoint	Shift+F9	Shift+F9
Execute Next Line	F10	F10
Step Into Function	Shift+F4	Shift+F4
Finish Function/Loop	Shift+F6	Shift+F6
Continue	Shift+F5	Shift+F5
Stop Debugging	Shift+F8	Shift+F8

6 VERSION CONTROL

	Windows/Linux	Mac
Show diff	Ctrl+Alt+D	Ctrl+Option+D
Commit changes	Ctrl+Alt+M	Ctrl+Option+M
Scroll diff view	Ctrl+↑/↓	Ctrl+↑/↓
Stage/Unstage (Git)	Spacebar	Spacebar
Stage/Unstage and move to next	Enter	Enter

7 MAKE PACKAGES

	Windows/Linux	Mac
Build and Reload	Ctrl+Shift+B	Cmd+Shift+B
Load All (devtools)	Ctrl+Shift+L	Cmd+Shift+L
Test Package (Desktop)	Ctrl+Shift+T	Cmd+Shift+T
Test Package (Web)	Ctrl+Alt+F7	Cmd+Opt+F7
Check Package	Ctrl+Shift+E	Cmd+Shift+E
Document Package	Ctrl+Shift+D	Cmd+Shift+D

8 DOCUMENTS AND APPS

	Windows/Linux	Mac
Preview HTML (Markdown, etc.)	Ctrl+Shift+K	Cmd+Shift+K
Knit Document (knitr)	Ctrl+Shift+K	Cmd+Shift+K
Compile Notebook	Ctrl+Shift+K	Cmd+Shift+K
Compile PDF (TeX and Sweave)	Ctrl+Shift+K	Cmd+Shift+K
Insert chunk (Sweave and Knitr)	Ctrl+Alt+I	Cmd+Option+I
Insert code section	Ctrl+Shift+R	Cmd+Shift+R
Re-run previous region	Ctrl+Shift+P	Cmd+Shift+P
Run current document	Ctrl+Alt+R	Cmd+Option+R
Run from start to current line	Ctrl+Alt+B	Cmd+Option+B
Run the current code section	Ctrl+Alt+T	Cmd+Option+T
Run previous Sweave/Rmd code	Ctrl+Alt+P	Cmd+Option+P
Run the current chunk	Ctrl+Alt+C	Cmd+Option+C
Run the next chunk	Ctrl+Alt+N	Cmd+Option+N
Sync Editor & PDF Preview	Ctrl+F8	Cmd+F8
Previous plot	Ctrl+Alt+F11	Cmd+Option+F11
Next plot	Ctrl+Alt+F12	Cmd+Option+F12
Show Keyboard Shortcuts	Alt+Shift+K	Option+Shift+K



R Markdown :: CHEAT SHEET



What is R Markdown?

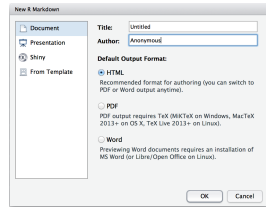


.Rmd files - An R Markdown (.Rmd) file is a record of your research. It contains the code that a scientist needs to reproduce your work along with the narration that a reader needs to understand your work.

Reproducible Research - At the click of a button, or the type of a command, you can rerun the code in an R Markdown file to reproduce your work and export the results as a finished report.

Dynamic Documents - You can choose to export the finished report in a variety of formats, including html, pdf, MS Word, or RTF documents; html or pdf based slides, Notebooks, and more.

Workflow



- 1 **Open a new .Rmd file** at File ► New File ► R Markdown. Use the wizard that opens to pre-populate the file with a template
- 2 **Write document** by editing template
- 3 **Knit document to create report**; use knit button or `render()` to knit
- 4 **Preview Output** in IDE window
- 5 **Publish** (optional) to web server
- 6 **Examine build log** in R Markdown console
- 7 **Use output file** that is saved along side .Rmd

render

Use `rmarkdown::render()` to render/knit at cmd line. Important args:

input - file to render	output_options - List of render options (as in YAML)	output_file	output_dir	params - list of params to use	envir - environment to evaluate code chunks in	encoding - of input file
-------------------------------	---	--------------------	-------------------	---------------------------------------	---	---------------------------------

Embed code with knitr syntax

INLINE CODE
Insert with ``r <code>``. Results appear as text without code.
Built with ``r getRversion()`` → Built with 3.2.3

CODE CHUNKS
One or more lines surrounded with ````${code}````. Place chunk options within curly braces, after `r`. Insert with ````${code}````

GLOBAL OPTIONS
Set with `knitr::opts_chunk$set()`, e.g.
````${code}````  
`knitr::opts_chunk$set(echo = TRUE)`

### IMPORTANT CHUNK OPTIONS

- cache** - cache results for future knits (default = FALSE)
- cache.path** - directory to save cached results in (default = "cache/")
- child** - file(s) to knit and then include (default = NULL)
- collapse** - collapse all output into single block (default = FALSE)
- comment** - prefix for each line of results (default = '###')

- dependson** - chunk dependencies for caching (default = NULL)
- echo** - Display code in output document (default = TRUE)
- engine** - code language used in chunk (default = 'R')
- error** - Display error messages in doc (TRUE) or stop render when errors occur (FALSE) (default = FALSE)
- eval** - Run code in chunk (default = TRUE)

- fig.align** - 'left', 'right', or 'center' (default = 'default')
- fig.cap** - figure caption as character string (default = NULL)
- fig.height, fig.width** - Dimensions of plots in inches
- highlight** - highlight source code (default = TRUE)
- include** - Include chunk in doc after running (default = TRUE)

- message** - display code messages in document (default = TRUE)
- results** (default = 'markup')  
'asis' - passthrough results  
'hide' - do not display results  
'hold' - put all results below all code
- tidy** - tidy code for display (default = FALSE)
- warning** - display code warnings in document (default = TRUE)

Options not listed above: `R.options`, `aniopts`, `autodep`, `background`, `cache.comments`, `cache.lazy`, `cache.rebuild`, `cache.vars`, `dev`, `dev.args`, `dpi`, `engine.opts`, `engine.path`, `fig.asp`, `fig.env`, `fig.ext`, `fig.keep`, `fig.lp`, `fig.path`, `fig.pos`, `fig.process`, `fig.retina`, `fig.scap`, `fig.show`, `fig.showtext`, `fig.subcap`, `interval`, `out.extra`, `out.height`, `out.width`, `prompt`, `purl`, `ref.label`, `render`, `size`, `split`, `tidy.opts`

## .rmd Structure

**YAML Header**  
Optional section of render (e.g. pandoc) options written as key:value pairs (YAML).  
At start of file  
Between lines of ---  
**Text**  
Narration formatted with markdown, mixed with:  
**Code Chunks**  
Chunks of embedded code. Each chunk:  
Begins with ````${code}````  
ends with ``````  
R Markdown will run the code and append the results to the doc.  
It will use the location of the .Rmd file as the **working directory**

## Parameters

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

1. **Add parameters** - Create and set parameters in the header as sub-values of params

```

params:
 n: 100
 d: !r Sys.Date()

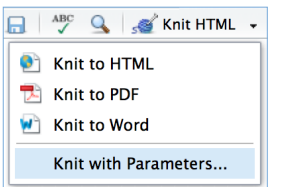
```

2. **Call parameters** - Call parameter values in code as `params$<name>`

```
Today's date
is !r params$d`
```

3. **Set parameters** - Set values with Knit with parameters or the params argument of render():

```
render("doc.Rmd", params = list(n = 1,
d = as.Date("2015-01-01")))
```



## Interactive Documents

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

Embed a complete app into your document with `shiny::shinyAppDir()`

NOTE: Your report will be rendered as a Shiny app, which means you must choose an html output format, like `html_document`, and serve it with an active R Session.





# Pandoc's Markdown

Write with syntax on the left to create effect on right (after render)

Plain text  
End a line with two spaces to start a new paragraph.  
**italics** and **bold**  
`verbatim code`  
sub/superscript<sup>2</sup>~  
~strikethrough~  
escaped: \\* \\_ \\  
endash: --, emdash: ---  
equation: \$A = \pi \* r^2\$  
equation block:

Plain text  
End a line with two spaces to start a new paragraph.  
*italics* and **bold**  
`verbatim code`  
sub/superscript<sup>2</sup>  
~strikethrough~  
escaped: \* \_ \  
endash: --, emdash: ---  
equation:  $A = \pi * r^2$   
equation block:

\$\$E = mc^2\$\$  
> block quote  
# Header1 {#anchor}  
## Header 2 {#css\_id}  
### Header 3 {#css\_class}  
#### Header 4  
##### Header 5  
##### Header 6  
<!--Text comment-->  
<code>Text ignored in HTML</code>  
<em>HTML ignored in pdfs</em>  
<http://www.rstudio.com>  
[[link](www.rstudio.com)]  
Jump to [Header 1](#anchor)  
image:

$$E = mc^2$$
  
block quote  


## Header1

### Header 2

#### Header 3

##### Header 4

###### Header 5

HTML ignored in pdfs  
<http://www.rstudio.com>  
link  
Jump to [Header 1](#)  
image:  
  
Caption

\* unordered list  
+ sub-item 1  
+ sub-item 2  
- sub-sub-item 1  
\* item 2  
Continued (indent 4 spaces)  
1. ordered list  
2. item 2  
i) sub-item 1  
A. sub-sub-item 1  
(@) A list whose numbering continues after  
(@) an interruption  
Term 1  
Definition 1

• unordered list  
◦ sub-item 1  
◦ sub-item 2  
▪ sub-sub-item 1  
• item 2  
Continued (indent 4 spaces)  
1. ordered list  
2. item 2  
i. sub-item 1  
A. sub-sub-item 1  
1. A list whose numbering continues after  
2. an interruption  
Term 1  
Definition 1

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1
- slide bullet 1  
- slide bullet 2  
(-> to have bullets appear on click)  
horizontal rule/slide break:  
\*\*\*  
A footnote [^1]  
[^1]: Here is the footnote.

Right Left Default Center  
12 12 12 12  
123 123 123 123  
1 1 1 1  
• slide bullet 1  
• slide bullet 2  
(-> to have bullets appear on click)  
horizontal rule/slide break:  
A footnote <sup>1</sup>  
1. Here is the footnote. ↩

# Set render options with YAML

1. runs the R code, embeds results and text into .md file with knitr
2. then converts the .md file into the finished format with pandoc



Set a document's default output format in the YAML header:

```

output: html_document

Body
```

output value	creates
html_document	html
pdf_document	pdf (requires Tex)
word_document	Microsoft Word (.docx)
odt_document	OpenDocument Text
rtf_document	Rich Text Format
md_document	Markdown
github_document	Github compatible markdown
ioslides_presentation	ioslides HTML slides
slidy_presentation	slidy HTML slides
beamer_presentation	Beamer pdf slides (requires Tex)

Customize output with sub-options (listed to the right):

```

output: html_document:
 code_folding: hide
 toc_float: TRUE

Body
```

### html tabsets

Use tabset css class to place sub-headers into tabs

```
Tabset {tabset .tabset-fade .tabset-pills}
Tab 1
text 1
Tab 2
text 2
End tabset
```

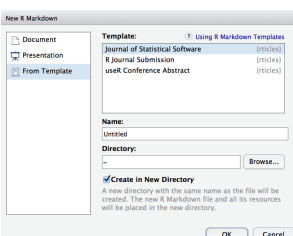
## Create a Reusable Template

1. Create a new package with an inst/rmarkdown/templates directory
2. In the directory, Place a folder that contains: **template.yaml** (see below) **skeleton.Rmd** (contents of the template) any supporting files
3. Install the package
4. Access **template** in wizard at File ▶ New File ▶ R Markdown template.yaml

```

name: My Template

```



sub-option	description	html	pdf	word	odt	rtf	md	github	ioslides	slidy	beamer
citation_package	The LaTeX package to process citations, natbib, biblatex or none		X				X				X
code_folding	Let readers to toggle the display of R code, "none", "hide", or "show"	X									
colortheme	Beamer color theme to use										X
css	CSS file to use to style document	X							X	X	
dev	Graphics device to use for figure output (e.g. "png")	X	X				X	X	X	X	X
duration	Add a countdown timer (in minutes) to footer of slides										X
fig_caption	Should figures be rendered with captions?	X	X	X	X				X	X	X
fig_height, fig_width	Default figure height and width (in inches) for document	X	X	X	X	X	X	X	X	X	X
highlight	Syntax highlighting: "tango", "pygments", "kate", "zenburn", "textmate"	X	X	X							X
includes	File of content to place in document (in_header, before_body, after_body)	X	X		X		X	X	X	X	X
incremental	Should bullets appear one at a time (on presenter mouse clicks)?									X	X
keep_md	Save a copy of .md file that contains knitr output	X		X	X	X				X	X
keep_tex	Save a copy of .tex file that contains knitr output	X									X
latex_engine	Engine to render latex, "pdflatex", "xelatex", or "lualatex"		X								X
lib_dir	Directory of dependency files to use (Bootstrap, MathJax, etc.)	X								X	X
mathjax	Set to local or a URL to use a local/URL version of MathJax to render equations	X								X	X
md_extensions	Markdown extensions to add to default definition or R Markdown	X	X	X	X	X	X	X	X	X	X
number_sections	Add section numbering to headers	X	X								
pandoc_args	Additional arguments to pass to Pandoc	X	X	X	X	X	X	X	X	X	X
preserve_yaml	Preserve YAML front matter in final document?							X			
reference_docx	docx file whose styles should be copied when producing docx output			X							
self_contained	Embed dependencies into the doc	X								X	X
slide_level	The lowest heading level that defines individual slides										X
smaller	Use the smaller font size in the presentation?										X
smart	Convert straight quotes to curly, dashes to em-dashes, ... to ellipses, etc.	X								X	X
template	Pandoc template to use when rendering file quarterly_report.html).	X	X		X						X
theme	Bootswatch or Beamer theme to use for page	X									X
toc	Add a table of contents at start of document	X	X	X		X	X	X			X
toc_depth	The lowest level of headings to add to table of contents	X	X	X		X	X	X			
toc_float	Float the table of contents to the left of the main content	X									

## Table Suggestions

Several functions format R data into tables

eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62

eruptions	waiting	
1	3.600	79
2	1.800	54.00
3	3.333	74.00
4	2.283	62.00

Table with xtable

```
data <- faithful[1:4,]
knitr::kable(data, caption = "Table with kable")
knitr::xtable(data, caption = "Table with xtable",
 type = "html", html.table.attributes = "border=0")
stargazer::stargazer(data, type = "html", title = "Table
with stargazer")
```

Learn more in the stargazer, xtable, and knitr packages.

## Citations and Bibliographies

Create citations with .bib, .bibtex, .copac, .enl, .json, .medline, .mods, .ris, .wos, and .xml files

1. Set bibliography file and CSL 1.0 Style file (optional) in the YAML header
2. Use citation keys in text

```

bibliography: refs.bib
cs1: style.csl

```

Smith cited [@smith04].  
Smith cited without author [-@smith04].  
@smith04 cited in line.

3. Render. Bibliography will be added to end of document

Smith cited (Joe Smith 2004).  
Smith cited without author (2004).  
Joe Smith (2004) cited in line.



# 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 Packages

##### 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

#### Creating Vectors

c(2, 4, 6)	2 4 6	Join elements into a vector
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
rep(1:2, each=3)	1 1 1 2 2 2	Repeat elements of a vector

#### Vector Functions

##### sort(x)

Return x sorted.

##### table(x)

See counts of values.

##### rev(x)

Return x reversed.

##### unique(x)

See unique values.

#### Selecting Vector Elements

##### By Position

x[4]	The fourth element.
x[-4]	All but the fourth.
x[2:4]	Elements two to four.
x[-(2:4)]	All elements except two to four.
x[c(1, 5)]	Elements one and five.

##### By Value

x[x == 10]	Elements which are equal to 10.
x[x < 0]	All elements less than zero.
x[x %in% c(1, 2, 5)]	Elements in the set 1, 2, 5.

##### Named Vectors

x['apple']	Element with name 'apple'.
------------	----------------------------

### Programming

#### For Loop

```
for (variable in sequence){
 Do something
}
```

##### Example

```
for (i in 1:4){
 j <- i + 10
 print(j)
}
```

#### While Loop

```
while (condition){
 Do something
}
```

##### Example

```
while (i < 5){
 print(i)
 i <- i + 1
}
```

#### If Statements

```
if (condition){
 Do something
} else {
 Do something different
}
```

##### Example

```
if (i > 3){
 print('Yes')
} else {
 print('No')
}
```

#### Functions

```
function_name <- function(var){
 Do something
 return(new_variable)
}
```

##### Example

```
square <- function(x){
 squared <- x*x
 return(squared)
}
```

### Reading and Writing Data

Also see the **readr** package.

Input	Output	Description
df <- read.table('file.txt')	write.table(df, 'file.txt')	Read and write a delimited text file.
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.
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	is.na(a)	Is missing
a != b	Not equal	a < b	Less than	a <= b	Less than or equal to	is.null(a)	Is null

## Types

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

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

## Maths Functions

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

## Variable Assignment

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




## The Environment

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

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

## Matrices

```
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: $m * x = n$

## Lists

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

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




Also see the **dplyr** package.

## 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

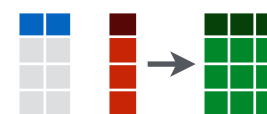
<code>View(df)</code>	See the full data frame.
<code>head(df)</code>	See the first 6 rows.

`nrow(df)`  
Number of rows.

`ncol(df)`  
Number of columns.

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

`cbind` - Bind columns.



`rbind` - Bind rows.



## Strings

Also see the **stringr** package.

<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>	Turn a vector into a factor. Can set the levels of the factor and the order.	<code>cut(x, breaks = 4)</code>	Turn a numeric vector into a factor by 'cutting' into sections.
------------------------	------------------------------------------------------------------------------	---------------------------------	-----------------------------------------------------------------

## Statistics

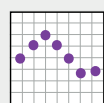
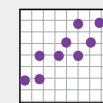
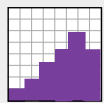
<code>lm(y ~ x, data=df)</code> Linear model.	<code>t.test(x, y)</code> Perform a t-test for difference between means.	<code>prop.test</code> Test for a difference between proportions.
<code>glm(y ~ x, data=df)</code> Generalised linear model.	<code>pairwise.t.test</code> Perform a t-test for paired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

## 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>punif</code>	<code>qunif</code>

## Plotting

Also see the **ggplot2** package.

 <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** package.

# Data Import :: CHEAT SHEET



R's **tidyverse** is built around **tidy data** stored in **tibbles**, which are enhanced data frames.



The front side of this sheet shows how to read text files into R with **readr**.



The reverse side shows how to create tibbles with **tibble** and to layout tidy data with **tidyr**.

## OTHER TYPES OF DATA

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

- **haven** - SPSS, Stata, and SAS files
- **readxl** - excel files (.xls and .xlsx)
- **DBI** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)

## Save Data

Save **x**, an R object, to **path**, a file path, as:

### Comma delimited file

**write\_csv(x, path, na = "NA", append = FALSE, col\_names = !append)**

### File with arbitrary delimiter

**write\_delim(x, path, delim = " ", na = "NA", append = FALSE, col\_names = !append)**

### CSV for excel

**write\_excel\_csv(x, path, na = "NA", append = FALSE, col\_names = !append)**

### String to file

**write\_file(x, path, append = FALSE)**

### String vector to file, one element per line

**write\_lines(x, path, na = "NA", append = FALSE)**

### Object to RDS file

**write\_rds(x, path, compress = c("none", "gz", "bz2", "xz"), ...)**

### Tab delimited files

**write\_tsv(x, path, na = "NA", append = FALSE, col\_names = !append)**

## Read Tabular Data - These functions share the common arguments:

```
read_*(file, col_names = TRUE, col_types = NULL, locale = default_locale(), na = c("", "NA"), quoted_na = TRUE, comment = "", trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000, n_max), progress = interactive())
```

```
a,b,c
1,2,3
4,5,NA
```

A	B	C
1	2	3
4	5	NA

### Comma Delimited Files

**read\_csv("file.csv")**

To make file.csv run:

`write_file(x = "a,b,c\n1,2,3\n4,5,NA", path = "file.csv")`

```
a;b;c
1;2;3
4;5;NA
```

A	B	C
1	2	3
4	5	NA

### Semi-colon Delimited Files

**read\_csv2("file2.csv")**

`write_file(x = "a;b;c\n1;2;3\n4;5;NA", path = "file2.csv")`

```
a|b|c
1|2|3
4|5|NA
```

A	B	C
1	2	3
4	5	NA

### Files with Any Delimiter

**read\_delim("file.txt", delim = "|")**

`write_file(x = "a|b|c\n1|2|3\n4|5|NA", path = "file.txt")`

```
a b c
1 2 3
4 5 NA
```

A	B	C
1	2	3
4	5	NA

### Fixed Width Files

**read\_fwf("file.fwf", col\_positions = c(1, 3, 5))**

`write_file(x = "a b c\n1 2 3\n4 5 NA", path = "file.fwf")`

### Tab Delimited Files

**read\_tsv("file.tsv")** Also **read\_table()**.

`write_file(x = "a\tb\tc\n1\t2\t3\n4\t5\tNA", path = "file.tsv")`

## USEFUL ARGUMENTS

```
a,b,c
1,2,3
4,5,NA
```

### Example file

`write_file("a,b,c\n1,2,3\n4,5,NA","file.csv")`  
`f <- "file.csv"`

1	2	3
4	5	NA

### Skip lines

`read_csv(f, skip = 1)`

A	B	C
1	2	3
4	5	NA

### No header

`read_csv(f, col_names = FALSE)`

A	B	C
1	2	3

### Read in a subset

`read_csv(f, n_max = 1)`

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

### Provide header

`read_csv(f, col_names = c("x", "y", "z"))`

A	B	C
NA	2	3
4	5	NA

### Missing Values

`read_csv(f, na = c("1", "!"))`

## Read Non-Tabular Data

### Read a file into a single string

`read_file(file, locale = default_locale())`

### Read each line into its own string

`read_lines(file, skip = 0, n_max = -1L, na = character(), locale = default_locale(), progress = interactive())`

### Read Apache style log files

`read_log(file, col_names = FALSE, col_types = NULL, skip = 0, n_max = -1, progress = interactive())`

### Read a file into a raw vector

`read_file_raw(file)`

### Read each line into a raw vector

`read_lines_raw(file, skip = 0, n_max = -1L, progress = interactive())`

## Data types

readr functions guess the types of each column and convert types when appropriate (but will NOT convert strings to factors automatically).

A message shows the type of each column in the result.

```
Parsed with column specification:
cols(
age = col_integer(),
sex = col_character(),
earn = col_double()
)
```

age is an integer

sex is a character

earn is a double (numeric)

1. Use **problems()** to diagnose problems

`x <- read_csv("file.csv"); problems(x)`

2. Use a **col\_** function to guide parsing

- **col\_guess()** - the default
- **col\_character()**
- **col\_double()**, **col\_euro\_double()**
- **col\_datetime(format = "")** Also **col\_date(format = "")**, **col\_time(format = "")**
- **col\_factor(levels, ordered = FALSE)**
- **col\_integer()**
- **col\_logical()**
- **col\_number()**, **col\_numeric()**
- **col\_skip()**

```
x <- read_csv("file.csv", col_types = cols(
 A = col_double(),
 B = col_logical(),
 C = col_factor()))
```

3. Else, read in as character vectors then parse with a **parse\_** function.

- **parse\_guess()**
- **parse\_character()**
- **parse\_datetime()** Also **parse\_date()** and **parse\_time()**
- **parse\_double()**
- **parse\_factor()**
- **parse\_integer()**
- **parse\_logical()**
- **parse\_number()**

`x$A <- parse_number(x$A)`



# Tibbles - an enhanced data frame



The **tibble** package provides a new S3 class for storing tabular data, the tibble. Tibbles inherit the data frame class, but improve three behaviors:

- **Subsetting** - `[` always returns a new tibble, `[[` and `$` always return a vector.
- **No partial matching** - You must use full column names when subsetting
- **Display** - When you print a tibble, R provides a concise view of the data that fits on one screen

A large table to display

```
A tibble: 234 × 6
 manufacturer <chr> model <chr> displ <dbl>
1 audi a4 1.8
2 audi a4 2.0
3 audi a4 2.0
4 audi a4 2.0
5 audi a4 3.1
6 audi a4 quattro 1.8
7 audi a4 quattro 1.8
8 ... with 224 more rows, and 3
 more variables: year <int>,
 cyl <int>, trans <chr>
```

tibble display

```
156 1999 6 auto(l4)
157 1999 6 auto(l4)
158 2008 6 auto(l4)
159 2008 8 auto(s4)
160 1999 4 manual(m5)
161 1999 4 auto(l4)
162 2008 4 manual(m5)
163 2008 4 manual(m5)
164 2008 4 auto(l4)
165 2008 4 auto(l4)
166 1999 4 auto(l4)
[reached getOption("max.print")
 -- omitted 68 rows]
```

data frame display

- Control the default appearance with options:
  - `options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf)`
- View full data set with **View()** or **glimpse()**
- Revert to data frame with **as.data.frame()**

## CONSTRUCT A TIBBLE IN TWO WAYS

**tibble(...)**  
Construct by columns.  
`tibble(x = 1:3, y = c("a", "b", "c"))`

**tribble(...)**  
Construct by rows.  
`tribble(~x, ~y, 1, "a", 2, "b", 3, "c")`

```
A tibble: 3 × 2
 x y
 <int> <dbl>
1 1 2
2 2 3
3 3 c
```

Both make this tibble

- **as\_tibble(x, ...)** Convert data frame to tibble.
- **enframe(x, name = "name", value = "value")** Convert named vector to a tibble
- **is\_tibble(x)** Test whether x is a tibble.

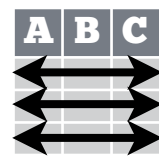
# Tidy Data with Tidyr

**Tidy data** is a way to organize tabular data. It provides a consistent data structure across packages.

A table is tidy if:

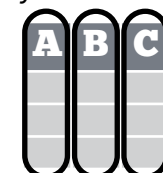


Each **variable** is in its own **column**

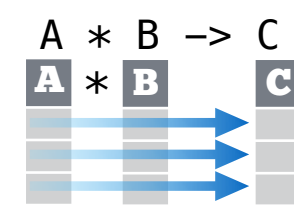


Each **observation**, or **case**, is in its own **row**

Tidy data:



Makes variables easy to access as vectors



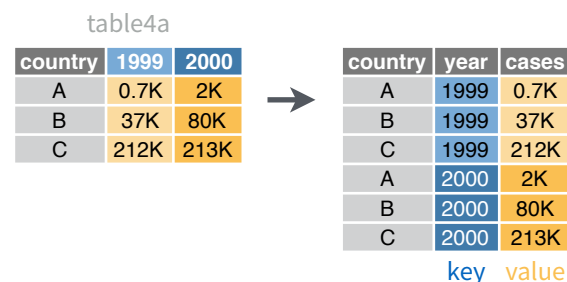
Preserves cases during vectorized operations

## Reshape Data - change the layout of values in a table

Use **gather()** and **spread()** to reorganize the values of a table into a new layout.

**gather(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor\_key = FALSE)**

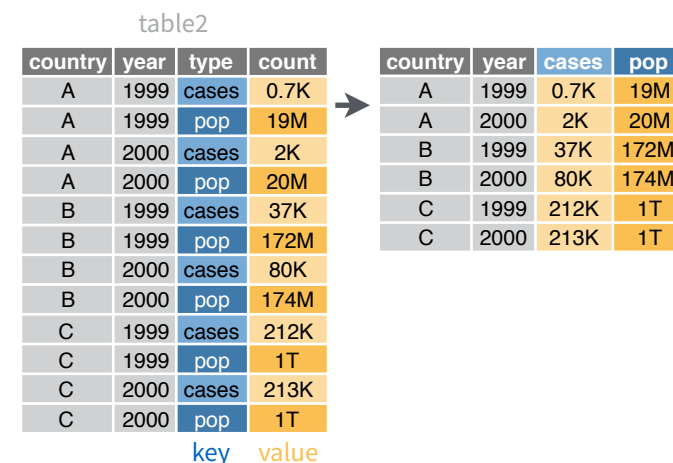
Gather moves column names into a **key** column, gathering the column values into a single **value** column.



`gather(table4a, `1999`, `2000`, key = "year", value = "cases")`

**spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL)**

Spread moves the unique values of a **key** column into the column names, spreading the values of a **value** column across the new columns.

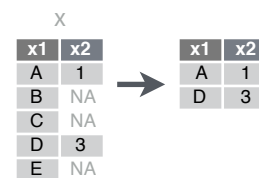


`spread(table2, type, count)`

## Handle Missing Values

**drop\_na(data, ...)**

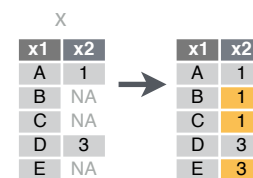
Drop rows containing NA's in ... columns.



`drop_na(x, x2)`

**fill(data, ..., .direction = c("down", "up"))**

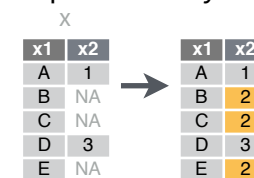
Fill in NA's in ... columns with most recent non-NA values.



`fill(x, x2)`

**replace\_na(data, replace = list(), ...)**

Replace NA's by column.



`replace_na(x, list(x2 = 2))`

## Expand Tables - quickly create tables with combinations of values

**complete(data, ..., fill = list())**

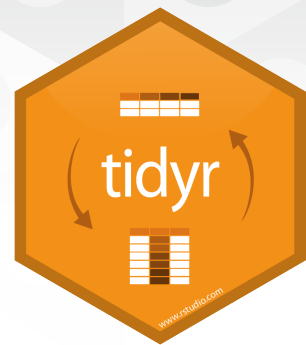
Adds to the data missing combinations of the values of the variables listed in ...  
`complete(mtcars, cyl, gear, carb)`

**expand(data, ...)**

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

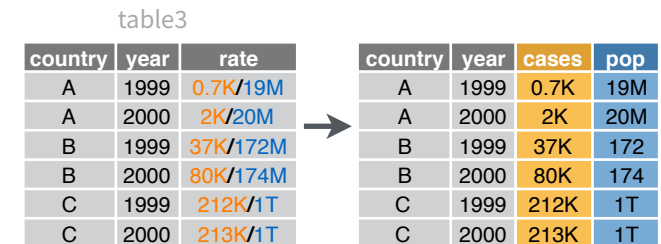
# Split Cells

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



**separate(data, col, into, sep = "[^:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ...)**

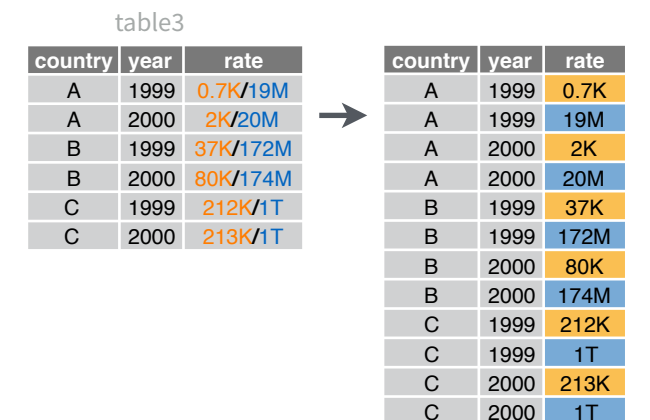
Separate each cell in a column to make several columns.



`separate(table3, rate, into = c("cases", "pop"))`

**separate\_rows(data, ..., sep = "[^:alnum:]]+", convert = FALSE)**

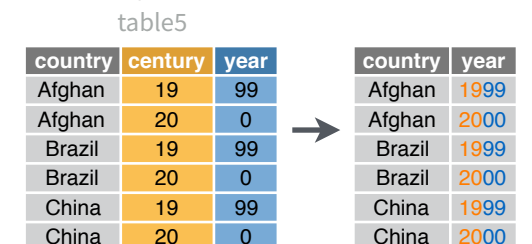
Separate each cell in a column to make several rows. Also **separate\_rows\_()**.



`separate_rows(table3, rate)`

**unite(data, col, ..., sep = "\_", remove = TRUE)**

Collapse cells across several columns to make a single column.



`unite(table5, century, year, col = "year", sep = "")`

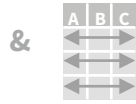
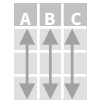




# Data Transformation with dplyr : : CHEAT SHEET



**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**



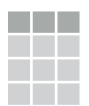
**pipes**

$x \%>\% f(y)$  becomes  $f(x, y)$

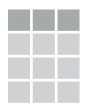
## Summarise Cases

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

**summary function**



**summarise(.data, ...)**  
Compute table of summaries. Also **summarise\_()**.  
*summarise(mtcars, avg = mean(mpg))*



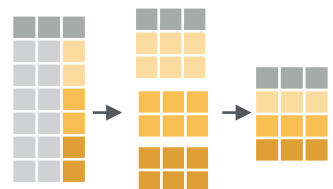
**count(x, ..., wt = NULL, sort = FALSE)**  
Count number of rows in each group defined by the variables in ... Also **tally()**.  
*count(iris, Species)*

### VARIATIONS

**summarise\_all()** - Apply funs to every column.  
**summarise\_at()** - Apply funs to specific columns.  
**summarise\_if()** - Apply funs to all cols of one type.

## Group Cases

Use **group\_by()** to create a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.



*mtcars %>%  
group\_by(cyl) %>%  
summarise(avg = mean(mpg))*

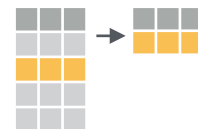
**group\_by(.data, ..., add = FALSE)**  
Returns copy of table grouped by ...  
*g\_iris <- group\_by(iris, Species)*

**ungroup(x, ...)**  
Returns ungrouped copy of table.  
*ungroup(g\_iris)*

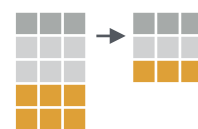
## Manipulate Cases

### EXTRACT CASES

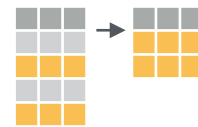
Row functions return a subset of rows as a new table. Use a variant that ends in **\_** for non-standard evaluation friendly code.



**filter(.data, ...)** Extract rows that meet logical criteria. Also **filter\_()**. *filter(iris, Sepal.Length > 7)*



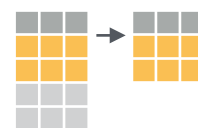
**distinct(.data, ..., .keep\_all = FALSE)** Remove rows with duplicate values. Also **distinct\_()**.  
*distinct(iris, Species)*



**sample\_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame())** Randomly select fraction of rows.  
*sample\_frac(iris, 0.5, replace = TRUE)*



**sample\_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame())** Randomly select size rows. *sample\_n(iris, 10, replace = TRUE)*



**slice(.data, ...)** Select rows by position. Also **slice\_()**. *slice(iris, 10:15)*

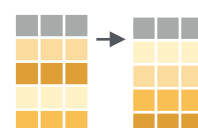
**top\_n(x, n, wt)** Select and order top n entries (by group if grouped data). *top\_n(iris, 5, Sepal.Width)*

### Logical and boolean operators to use with filter()

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

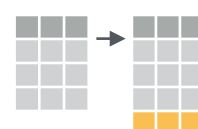
See **?base::logic** and **?Comparison** for help.

### ARRANGE CASES



**arrange(.data, ...)**  
Order rows by values of a column (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(faithful, eruptions = 1, waiting = 1)*

Column functions return a set of columns as a new table. Use a variant that ends in **\_** for non-standard evaluation friendly code.



**select(.data, ...)**  
Extract columns by name. Also **select\_if()**  
*select(iris, Sepal.Length, Species)*

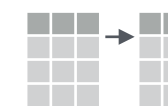
Use these helpers with **select()**, e.g. *select(iris, starts\_with("Sepal"))*

<b>contains(match)</b>	<b>num_range(prefix, range)</b>	;, e.g. <i>mpg:cyl</i>
<b>ends_with(match)</b>	<b>one_of(...)</b>	-, e.g. <i>-Species</i>
<b>matches(match)</b>	<b>starts_with(match)</b>	

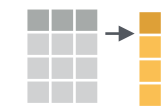
### MAKE NEW VARIABLES

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

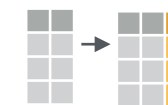
**vectorized function**



**mutate(.data, ...)**  
Compute new column(s).  
*mutate(mtcars, gpm = 1/mpg)*



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



**mutate\_all(.tbl, .funs, ...)** Apply funs to every column. Use with **funs()**.  
*mutate\_all(faithful, funs(log(.), log2(.)))*



**mutate\_at(.tbl, .cols, .funs, ...)** Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for **select()**.  
*mutate\_at(iris, vars(-Species), funs(log(.)))*

**mutate\_if(.tbl, .predicate, .funs, ...)**  
Apply funs to all columns of one type. Use with **funs()**.  
*mutate\_if(iris, is.numeric, funs(log(.)))*



**add\_column(.data, ..., .before = NULL, .after = NULL)** Add new column(s).  
*add\_column(mtcars, new = 1:32)*



**rename(.data, ...)** Rename columns.  
*rename(iris, Length = Sepal.Length)*



# 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.



## OFFSETS

`dplyr::lag()` - Offset elements by 1  
`dplyr::lead()` - Offset elements by -1

## CUMULATIVE AGGREGATES

`dplyr::cumall()` - Cumulative all()  
`dplyr::cumany()` - Cumulative any()  
**cummax()** - Cumulative max()  
`dplyr::cummean()` - Cumulative mean()  
**cummin()** - Cumulative min()  
**cumprod()** - Cumulative prod()  
**cumsum()** - Cumulative sum()

## RANKINGS

`dplyr::cume_dist()` - Proportion of all values <=   
`dplyr::dense_rank()` - rank with 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

## MISC

`dplyr::between()` - `x >= left & x <= right`  
`dplyr::case_when()` - multi-case if\_else()  
`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()  
`dplyr::recode()` - Vectorized switch()  
`dplyr::recode_factor()` - Vectorized switch() for factors

# 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.



## COUNTS

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

## LOCATION

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

## LOGICALS

**mean()** - Proportion of TRUE's  
**sum()** - # of TRUE's

## POSITION/ORDER

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

## RANK

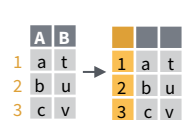
`quantile()` - nth quantile  
`min()` - minimum value  
`max()` - maximum value

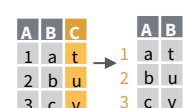
## SPREAD

`IQR()` - Inter-Quartile Range  
`mad()` - mean 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.

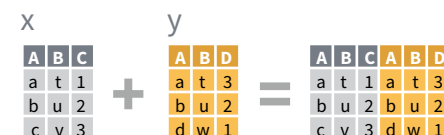
 **rownames\_to\_column()**  
Move row names into col.  
`a <- rownames_to_column(iris, var = "C")`

 **column\_to\_rownames()**  
Move col in row names.  
`column_to_rownames(a, var = "C")`

Also has `rownames()`, `remove_rownames()`

# Combine Tables

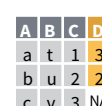
## COMBINE VARIABLES

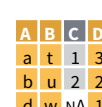



Use **bind\_cols()** to paste tables beside each other as they are.


**bind\_cols(...)** Returns tables placed side by side as a single table.  
BE SURE THAT ROWS ALIGN.


Use a "Mutating Join" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.

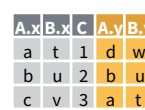
 **left\_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)**  
Join matching values from y to x.


 **right\_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...)**  
Join matching values from x to y.

 **inner\_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),...)**  
Join data. Retain only rows with matches.

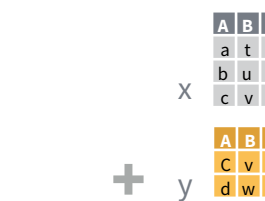
 **full\_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),...)**  
Join data. Retain all values, all rows.

 Use **by = c("col1", "col2")** to specify the column(s) to match on.  
`left_join(x, y, by = "A")`


 Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.  
`left_join(x, y, by = c("C" = "D"))`


 Use **suffix** to specify suffix to give to duplicate column names.  
`left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))`


## COMBINE CASES




Use **bind\_rows()** to paste tables below each other as they are.

 **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)

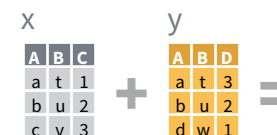
 **intersect(x, y, ...)**  
Rows that appear in both x and z.

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


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


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

## EXTRACT ROWS



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

 **semi\_join(x, y, by = NULL, ...)**  
Return rows of x that have a match in y. USEFUL TO SEE WHAT WILL BE JOINED.

 **anti\_join(x, y, by = NULL, ...)**  
Return rows of x that do not have a match in y. USEFUL TO SEE WHAT WILL NOT BE JOINED.

# Data Wrangling with dplyr and tidyr

## Cheat Sheet



### Syntax - Helpful conventions for wrangling

#### dplyr::tbl\_df(iris)

Converts data to tbl class. tbl's are easier to examine than data frames. R displays only the data that fits onscreen:

```
Source: local data frame [150 x 5]
 Sepal.Length Sepal.Width Petal.Length
1 5.1 3.5 1.4
2 4.9 3.0 1.4
3 4.7 3.2 1.3
4 4.6 3.1 1.5
5 5.0 3.6 1.4
..
Variables not shown: Petal.Width (dbl),
Species (fctr)
```

#### dplyr::glimpse(iris)

Information dense summary of tbl data.

#### utils::View(iris)

View data set in spreadsheet-like display (note capital V).

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa

#### dplyr::%>%

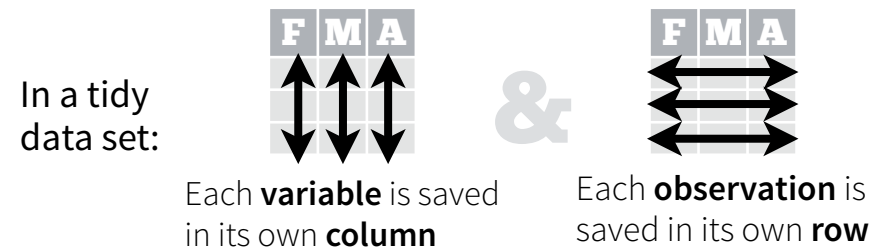
Passes object on left hand side as first argument (or . argument) of function on righthand side.

$x \%>\% f(y)$  is the same as  $f(x, y)$   
 $y \%>\% f(x, ., z)$  is the same as  $f(x, y, z)$

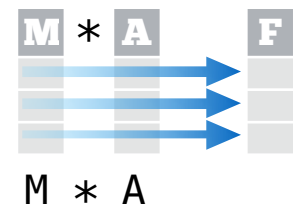
"Piping" with %>% makes code more readable, e.g.

```
iris %>%
 group_by(Species) %>%
 summarise(avg = mean(Sepal.Width)) %>%
 arrange(avg)
```

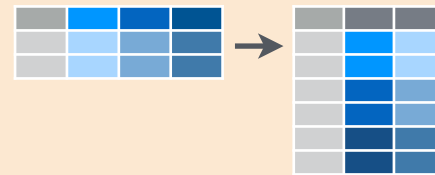
## Tidy Data - A foundation for wrangling in R



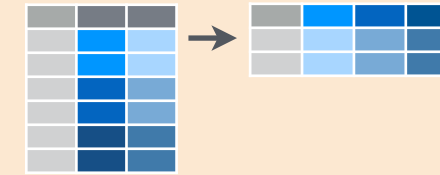
Tidy data complements R's **vectorized operations**. R will automatically preserve observations as you manipulate variables. No other format works as intuitively with R.



### Reshaping Data - Change the layout of a data set



**tidyr::gather(cases, "year", "n", 2:4)**  
Gather columns into rows.



**tidyr::spread(pollution, size, amount)**  
Spread rows into columns.



**tidyr::separate(storms, date, c("y", "m", "d"))**  
Separate one column into several.



**tidyr::unite(data, col, ..., sep)**  
Unite several columns into one.

**dplyr::data\_frame(a = 1:3, b = 4:6)**  
Combine vectors into data frame (optimized).

**dplyr::arrange(mtcars, mpg)**  
Order rows by values of a column (low to high).

**dplyr::arrange(mtcars, desc(mpg))**  
Order rows by values of a column (high to low).

**dplyr::rename(tb, y = year)**  
Rename the columns of a data frame.

### Subset Observations (Rows)



**dplyr::filter(iris, Sepal.Length > 7)**  
Extract rows that meet logical criteria.

**dplyr::distinct(iris)**  
Remove duplicate rows.

**dplyr::sample\_frac(iris, 0.5, replace = TRUE)**  
Randomly select fraction of rows.

**dplyr::sample\_n(iris, 10, replace = TRUE)**  
Randomly select n rows.

**dplyr::slice(iris, 10:15)**  
Select rows by position.

**dplyr::top\_n(storms, 2, date)**  
Select and order top n entries (by group if grouped data).

### Subset Variables (Columns)



**dplyr::select(iris, Sepal.Width, Petal.Length, Species)**  
Select columns by name or helper function.

#### Helper functions for select - ?select

- select(iris, contains("."))**  
Select columns whose name contains a character string.
- select(iris, ends\_with("Length"))**  
Select columns whose name ends with a character string.
- select(iris, everything())**  
Select every column.
- select(iris, matches(".t."))**  
Select columns whose name matches a regular expression.
- select(iris, num\_range("x", 1:5))**  
Select columns named x1, x2, x3, x4, x5.
- select(iris, one\_of(c("Species", "Genus")))**  
Select columns whose names are in a group of names.
- select(iris, starts\_with("Sepal"))**  
Select columns whose name starts with a character string.
- select(iris, Sepal.Length:Petal.Width)**  
Select all columns between Sepal.Length and Petal.Width (inclusive).
- select(iris, -Species)**  
Select all columns except Species.

#### Logic in R - ?Comparison, ?base::Logic

<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&,  , !, xor, any, all	Boolean operators

## Summarise Data



`dplyr::summarise(iris, avg = mean(Sepal.Length))`

Summarise data into single row of values.

`dplyr::summarise_each(iris, funs(mean))`

Apply summary function to each column.

`dplyr::count(iris, Species, wt = Sepal.Length)`

Count number of rows with each unique value of variable (with or without weights).



Summarise uses **summary functions**, functions that take a vector of values and return a single value, such as:

- |                                                                     |                                                    |
|---------------------------------------------------------------------|----------------------------------------------------|
| <code>dplyr::first</code><br>First value of a vector.               | <code>min</code><br>Minimum value in a vector.     |
| <code>dplyr::last</code><br>Last value of a vector.                 | <code>max</code><br>Maximum value in a vector.     |
| <code>dplyr::nth</code><br>Nth value of a vector.                   | <code>mean</code><br>Mean value of a vector.       |
| <code>dplyr::n</code><br># of values in a vector.                   | <code>median</code><br>Median value of a vector.   |
| <code>dplyr::n_distinct</code><br># of distinct values in a vector. | <code>var</code><br>Variance of a vector.          |
| <code>IQR</code><br>IQR of a vector.                                | <code>sd</code><br>Standard deviation of a vector. |

## Group Data

`dplyr::group_by(iris, Species)`

Group data into rows with the same value of Species.

`dplyr::ungroup(iris)`

Remove grouping information from data frame.

`iris %>% group_by(Species) %>% summarise(...)`

Compute separate summary row for each group.



## Make New Variables



`dplyr::mutate(iris, sepal = Sepal.Length + Sepal.Width)`

Compute and append one or more new columns.

`dplyr::mutate_each(iris, funs(min_rank))`

Apply window function to each column.

`dplyr::transmute(iris, sepal = Sepal.Length + Sepal.Width)`

Compute one or more new columns. Drop original columns.

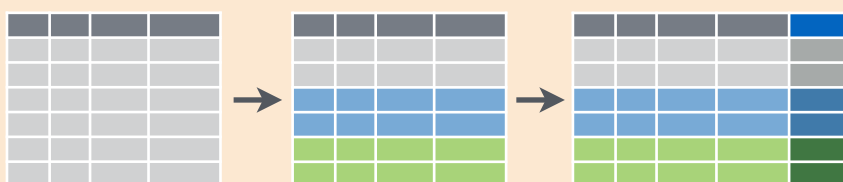


Mutate uses **window functions**, functions that take a vector of values and return another vector of values, such as:

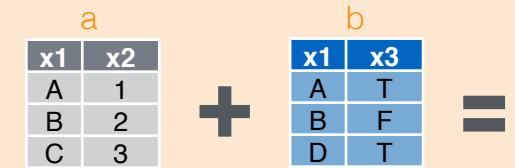
- |                                                                   |                                                       |
|-------------------------------------------------------------------|-------------------------------------------------------|
| <code>dplyr::lead</code><br>Copy with values shifted by 1.        | <code>dplyr::cumall</code><br>Cumulative <b>all</b>   |
| <code>dplyr::lag</code><br>Copy with values lagged by 1.          | <code>dplyr::cumany</code><br>Cumulative <b>any</b>   |
| <code>dplyr::dense_rank</code><br>Ranks with no gaps.             | <code>dplyr::cummean</code><br>Cumulative <b>mean</b> |
| <code>dplyr::min_rank</code><br>Ranks. Ties get min rank.         | <code>cumsum</code><br>Cumulative <b>sum</b>          |
| <code>dplyr::percent_rank</code><br>Ranks rescaled to [0, 1].     | <code>cummax</code><br>Cumulative <b>max</b>          |
| <code>dplyr::row_number</code><br>Ranks. Ties got to first value. | <code>cummin</code><br>Cumulative <b>min</b>          |
| <code>dplyr::ntile</code><br>Bin vector into n buckets.           | <code>cumprod</code><br>Cumulative <b>prod</b>        |
| <code>dplyr::between</code><br>Are values between a and b?        | <code>pmax</code><br>Element-wise <b>max</b>          |
| <code>dplyr::cume_dist</code><br>Cumulative distribution.         | <code>pmin</code><br>Element-wise <b>min</b>          |

`iris %>% group_by(Species) %>% mutate(...)`

Compute new variables by group.



## Combine Data Sets



Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

x1	x3	x2
A	T	1
B	F	2
D	T	NA

x1	x2	x3
A	1	T
B	2	F

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

`dplyr::left_join(a, b, by = "x1")`

Join matching rows from b to a.

`dplyr::right_join(a, b, by = "x1")`

Join matching rows from a to b.

`dplyr::inner_join(a, b, by = "x1")`

Join data. Retain only rows in both sets.

`dplyr::full_join(a, b, by = "x1")`

Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

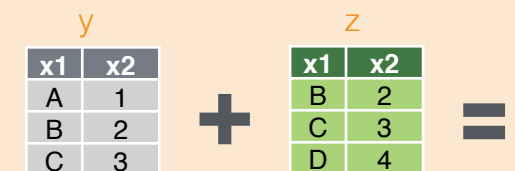
x1	x2
C	3

`dplyr::semi_join(a, b, by = "x1")`

All rows in a that have a match in b.

`dplyr::anti_join(a, b, by = "x1")`

All rows in a that do not have a match in b.



Set Operations

x1	x2
B	2
C	3

x1	x2
A	1
B	2
C	3
D	4

x1	x2
A	1

`dplyr::intersect(y, z)`

Rows that appear in both y and z.

`dplyr::union(y, z)`

Rows that appear in either or both y and z.

`dplyr::setdiff(y, z)`

Rows that appear in y but not z.

Binding

x1	x2
A	1
B	2
C	3
D	4

x1	x2	x1	x2
A	1	B	2
B	2	C	3
C	3	D	4

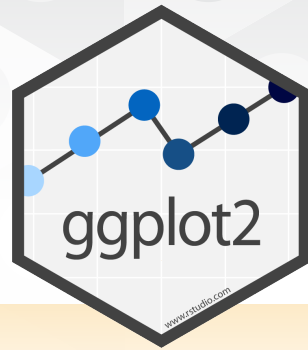
`dplyr::bind_rows(y, z)`

Append z to y as new rows.

`dplyr::bind_cols(y, z)`

Append z to y as new columns. Caution: matches rows by position.

# Data Visualization with ggplot2 : : CHEAT SHEET

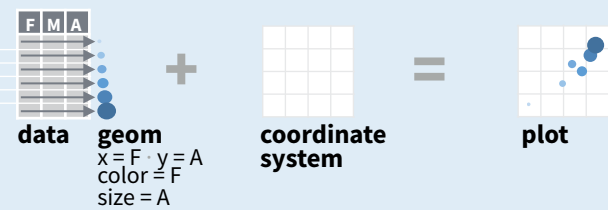


## Basics

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* (for geom function)  
*Not required, sensible defaults supplied* (for other functions)

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

qplot(x = cty, y = hwy, data = mpg, geom = "point") Creates a complete plot with given data, geom, and mappings. Supplies many useful defaults.

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.

## 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()** (Useful for expanding limits)
- b + geom\_curve**(aes(yend = lat + 1, xend=long+1, curvature=z)) - x, xend, y, yend, alpha, angle, color, curvature, linetype, size
- a + geom\_path**(lineend="butt", linejoin="round", linemitre=1) x, y, alpha, color, group, linetype, size
- a + geom\_polygon**(aes(group = group)) x, y, alpha, color, fill, group, 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

#### continuous x , continuous y

- ```
e <- ggplot(mpg, aes(cty, hwy))
```
- e + geom_label**(aes(label = cty), nudge_x = 1, nudge_y = 1, check_overlap = TRUE) x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust
 - e + geom_jitter**(height = 2, width = 2) x, y, alpha, color, fill, shape, size
 - 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, check_overlap = TRUE) x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

discrete x , continuous y

- ```
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

#### discrete x , discrete y

- ```
g <- ggplot(diamonds, aes(cut, color))
```
- g + geom_count**() x, y, alpha, color, fill, shape, size, stroke

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, colour, group, linetype, size, weight
  - 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_density2d**() x, y, alpha, colour, group, linetype, size
 - h + geom_hex**() x, y, alpha, colour, 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



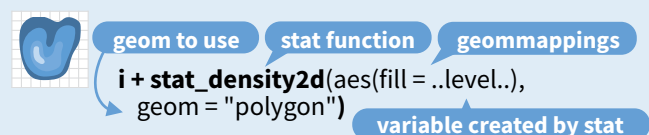


# Stats

An alternative way to build a layer



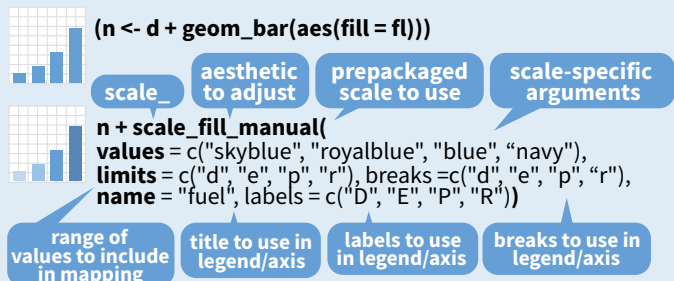
Visualize a stat by changing the default stat of a geom function, `geom_bar(stat="count")` or by using a stat function, `stat_count(geom="bar")`, which calls a default geom to make a layer (equivalent to a geom function). Use `..name..` syntax to map stat variables to aesthetics.



- c + stat\_bin(binwidth = 1, origin = 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() + stat\_function(aes(x = -3:3), n = 99, fun = dnorm, args = list(sd=0.5)) x | ..x.., ..y..
e + stat\_identity(na.rm = TRUE)
ggplot() + stat\_qq(aes(sample=1:100), dist = qt, dparam=list(df=5)) sample, x, y | ..sample.., ..theoretical..
e + stat\_sum() x, y, size | ..n.., ..prop..
e + stat\_summary(fun.data = "mean\_cl\_boot")
h + stat\_summary\_bin(fun.y = "mean", geom = "bar")
e + stat\_unique()

# Scales

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\_\*\_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 x values as date times. Use same arguments as scale\_x\_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 direction of x axis
scale\_x\_sqrt() - Plot x on square root scale

## COLOR AND FILL SCALES (DISCRETE)

- n <- d + geom\_bar(aes(fill = fl))
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(colours=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
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
r + coord\_flip()
xlim, ylim
Flipped Cartesian coordinates
r + coord\_polar(theta = "x", direction=1)
theta, start, direction
Polar coordinates
r + coord\_trans(ytrans = "sqrt")
xtrans, ytrans, limx, limy
Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.
pi + coord\_quickmap()
pi + coord\_map(projection = "ortho", orientation=c(41, -74, 0))projection, orientation, 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\_classic()
r + theme\_light()
r + theme\_linedraw()
Minimal themes
r + theme\_gray()
Grey background (default theme)
r + theme\_minimal()
Empty theme
r + theme\_dark()
dark for contrast

# 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(. ~ fl)
facet into columns based on fl
t + facet\_grid(year ~ .)
facet into rows based on year
t + facet\_grid(year ~ fl)
facet into both rows and columns
t + facet\_wrap(~ fl)
wrap facets into a rectangular layout

Set scales to let axis limits vary across facets

- t + facet\_grid(drv ~ 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 labeller to adjust facet labels

- t + facet\_grid(. ~ fl, labeller = label\_both)
fl: c fl: d fl: e fl: p fl: r
t + facet\_grid(fl ~ ., labeller = label\_bquote(alpha ^ .(fl)))
alpha^c alpha^d alpha^e alpha^p alpha^r
t + facet\_grid(. ~ fl, labeller = label\_parsed)
c d e p r

# Labels

- 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", <AES> = "New <AES> legend title")
Use scale functions to update legend labels
t + annotate(geom = "text", x = 8, y = 9, label = "A")
geom to place manual values for geom's aesthetics

# Legends

- n + theme(legend.position = "bottom")
Place legend at "bottom", "top", "left", or "right"
n + guides(fill = "none")
Set legend type for each aesthetic: colorbar, legend, or none (no legend)
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(10, 20))



# ggmap quickstart

For more functionality, see ggmap documentation and <https://dl.dropboxusercontent.com/u/24648660/ggmap%20userR%202012.pdf>

There are 2 basic steps to making a map using ggmap:

Part 1: Download map raster



Part 2: Plot raster and overlay data

## Part 1: Downloading the map raster

Start by loading the package: `library(ggmap)`

### 1. Define location: 3 ways

- location/address  
`myLocation <- "University of Washington"`
- lat/long  
`myLocation <- c(lon = -95.3632715, lat = 29.7632836)`
- bounding box lowerleftlon, lowerleftlat, upperrightlon, upperrightlat (a little glitchy for google maps)  
`myLocation <- c(-130, 30, -105, 50)`

Convert location/address its lat/long coordinates:  
`geocode("University of Washington")`

### 2. Define map source, type, and color

The `get_map` function provides a general approach for quickly obtaining maps from multiple sources. I like this option for exploring different styles of maps.

There are 4 map "sources" to obtain a map raster, and each of these sources has multiple "map types" (displayed on right).

- `stamen`: `mptype = c("terrain", "toner", "watercolor")`
- `google`: `mptype = c("roadmap", "terrain", "satellite", "hybrid")`
- `osm`: open street map
- `cloudmade`: 1000s of maps, but an api key must be obtained from <http://cloudmade.com>

```
myMap <- get_map(location=myLocation,
source="stamen", mptype="watercolor", crop=FALSE)
ggmap(myMap)
```

This will produce a map that looks something like this

NOTE: `crop = FALSE` because otherwise, with stamen plots, the map is slightly shifted when I overlay data.

The following maps show different map source/type options (except cloudmade)

The appearance of these maps may be very different depending on zoom/scale

stamen: watercolor



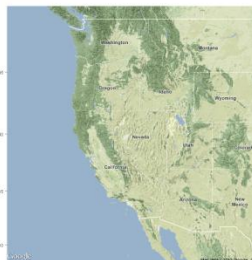
stamen: toner



stamen: terrain



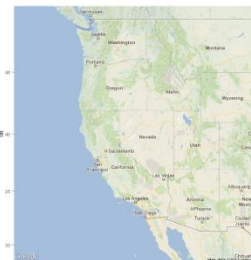
google: terrain



google: satellite



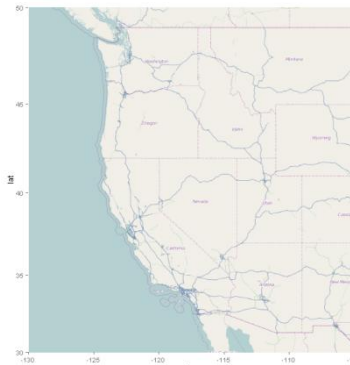
google: roadmap



google: hybrid

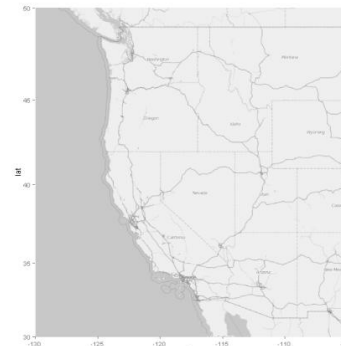


osm\*



\*Open street maps may return a '503 Service Unavailable' error. This means their server is unavailable, and you must wait for it to become available again.

All maps can be displayed in black and white  
`color = "bw"`



```
myMap <-
get_map(location=myLocation,
source="osm", color="bw")
```

If you can't get the map you want by adjusting the location/zoom variables, the functions designed for the different map sources provide more options:

```
get_googlemap,
get_openstreetmap,
get_stamenmap,
get_cloudmademap
```

- If you use Rstudio: Sometimes a plot will not display. Increasing the size of the plot window may help. `dev.off()` prior to plotting may also help.
- The `urlonly = TRUE` will return a url that will display your map in a web browser. Which is pretty cool and may be handy!
- `legend="topleft"` will inset the legend on the top left of the map is data is overlaid (page 2).

# ggmap quickstart

## Part 2: Plotting the maps and data

### 1. Plot the raster:

```
ggmap(myMap)
```

### 2. Add points with latitude/longitude coordinates:

```
ggmap(myMap)+
geom_point(aes(x = Longitude, y = Latitude), data = data,
 alpha = .5, color="darkred", size = 3)
```

alpha = transparency  
color = color  
size = size of points

The *size*, *color*, *alpha*, and *shape* of the points can be scaled relative to another variable (in this case *estArea*) within the *aes* function:

```
ggmap(myMap)+
geom_point(aes(x = Longitude, y = Latitude, size=sqrt(estArea)),
 data = data, alpha = .5, color="darkred")
```



Additional functions can be added to control scaling, e.g.:

```
ggmap(myMap)+
geom_point(aes(x = Longitude, y = Latitude, size=sqrt(estArea)),
 data = data, alpha = .5, color="darkred")+
scale_size(range=c(3,20))
```

### 3. Add polygons from shp file

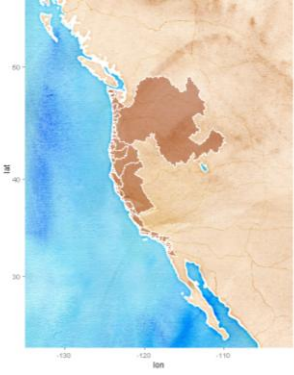
The shp file is imported into R using the *rgdal* package, and must be transformed to geographic coordinates (latitude/longitude) on the [World Geodetic System](#) of 1984 (WGS84) datum using the *rgdal* package:

```
library(rgdal)
shpData <- readOGR(dsn="C:\\Documents and Settings\\Watershed", layer="WS")
proj4string(shpData) # describes data's current coordinate reference system
to change to correct projection:
shpData <- spTransform(shpData,
 CRS("+proj=longlat +datum=WGS84"))
```

To plot the data:

```
geom_polygon(aes(x = long, y = lat, group=id),
 data = shpData, color = "white", fill = "orangered4",
 alpha = .4, size = .2)
```

color = outline color      alpha = transparency  
fill = polygon color      size = outline size



### 4. Annotate figure

```
baylor <- get_map('baylor university', zoom = 15, maptype = 'satellite')
ggmap(baylor) +
annotate('rect', xmin=-97.11, ymin=31.54, xmax=-97.12, ymax=31.55, col="red", fill="white")+
annotate('text', x=-97.12, y=31.54, label = 'Statistical Sciences', colour = l('red'), size = 8)+
annotate('segment', x=-97.12, xend=-97.12, y=31.55, yend=31.55,
 colour=l('red'), arrow = arrow(length=unit(0.3,"cm")), size = 1.5) +
labs(x = 'Longitude', y = 'Latitude') + ggtitle('Baylor University')
```

### Controlling size and color

**size**

- `scale_size(range = c(3, 20))`
- But, the following is better for display because it is based on area (rather than radius)
- `scale_area(range=c(3,20))`

**color**

- Continuous variables: color gradient between n colors, e.g.:
- `scale_colour_gradientn(colours = rainbow_hcl(7))`
- Discrete variables, e.g.:
- `scale_colour_manual(values=rainbow_hcl(7))`
- `scale_colour_manual(values=c("8" = "red", "4" = "blue", "6" = "green"))`
- \*Use *colorspace* and *RColorBrewer* to choose color combinations



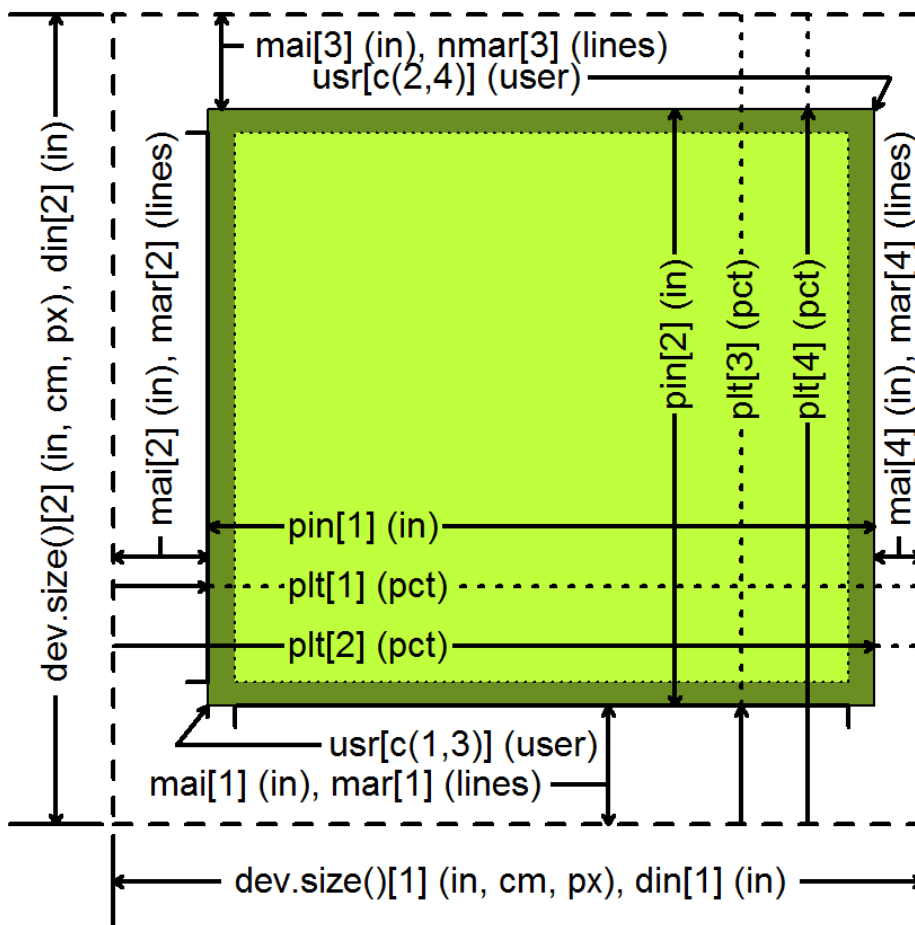


# How Big is Your Graph?

## An R Cheat Sheet

### Introduction

All functions that open a device for graphics will have **height** and **width** arguments to control the size of the graph and a **pointsize** argument to control the relative font size. In **knitr**, you control the size of the graph with the chunk options, **fig.width** and **fig.height**. This sheet will help you with calculating the size of the graph and various parts of the graph within R.



### Your graphics device

**dev.size()** (width, height)  
**par("din")** (*r.o.*) (width, height) in inches

Both the **dev.size** function and the **din** argument of **par** will tell you the size of the graphics device. The **dev.size** function will report the size in

1. inches (**units="in"**), the default
2. centimeters (**units="cm"**)
3. pixels (**units="px"**)

Like several other **par** arguments, **din** is read only (*r.o.*) meaning that you can ask its current value (**par("din")**) but you cannot change it (**par(din=c(5,7))** will fail).

### Your plot margins

**par("mai")** (bottom, left, top, right) in inches  
**par("mar")** (bottom, left, top, right) in lines

Margins provide you space for your axes, axis, labels, and titles.

A "line" is the amount of vertical space needed for a line of text.

If your graph has no axes or titles, you can remove the margins (and maximize the plotting region) with

```
par(mar=rep(0,4))
```

### Your plotting region

**par("pin")** (width, height) in inches  
**par("plt")** (left, right, bottom, top) in pct

The **pin** argument **par** gives you the size of the plotting region (the size of the device minus the size of the margins) in inches.

The **plt** argument gives you the percentage of the device from the left/bottom edge up to the left edge of the plotting region, the right edge, the bottom edge, and the top edge. The first and third values are equivalent to the percentage of space devoted to the left and bottom margins. Subtract the second and fourth values from 1 to get the percentage of space devoted to the right and top margins.

### Your x-y coordinates

**par("usr")** (xmin, ymin, xmax, ymax)

Your x-y coordinates are the values you use when plotting your data. This normally is not the same as the values you specified with the **xlim** and **ylim** arguments in **plot**. By default, R adds an extra 4% to the plotting range (see the dark green region on the figure) so that points right up on the edges of your plot do not get partially clipped. You can override this by setting **xaxs="n"** and/or the **yaxs="n"** in **par**.

Run **par("usr")** to find the minimum X value, the maximum X value, the minimum Y value, and the maximum Y value. If you assign new values to **usr**, you will update the x-y coordinates to the new values.

### Getting a square graph

**par("pty")**

You can produce a square graph manually by setting the width and height to the same value and setting the margins so that the sum of the top and bottom margins equal the sum of the left and right margins. But a much easier way is to specify **pty="s"**, which adjusts the margins so that the size of the plotting region is always square, even if you resize the graphics window.

### Converting units

For many applications, you need to be able to translate user coordinates to pixels or inches. There are some cryptic shortcuts, but the simplest way is to get the range in user coordinates and measure the proportion of the graphics device devoted to the plotting region.

```
user.range <- par("usr")[c(2,4)] -
par("usr")[c(1,3)]
```

```
region.pct <- par("plt")[c(2,4)] -
par("plt")[c(1,3)]
```

```
region.px <-
dev.size(units="px") * region.pct
```

```
px.per.xy <- region.px / user.range
```

To convert a horizontal or distance from the x-coordinate value to pixels, multiply by **px.per.xy[1]**. To convert a vertical distance, multiply by **region.px.per.xy[2]**. To convert a diagonal distance, you need to invoke Pythagoras.

```
a.px <- x.dist*px.per.xy[1]
b.px <- y.dist*px.per.xy[2]
c.px <- sqrt(a.px^2+b.px^2)
```

To rotate a string to match the slope of a line segment, you need to convert the distances to pixels, calculate the arctangent, and convert from radians to degrees.

```
segments(x0, y0, x1, y1)
delta.x <- (x1 - x0) * px.per.xy[1]
delta.y <- (y1 - y0) * px.per.xy[2]
angle.radians <- atan2(delta.y, delta.x)
angle.degrees <- angle.radians * 180 / pi
text(x1, y1, "TEXT", srt=angle.degrees)
```

## Panels

`par("fig")` (width, height) in pct  
`par("fin")` (width, height) in inches

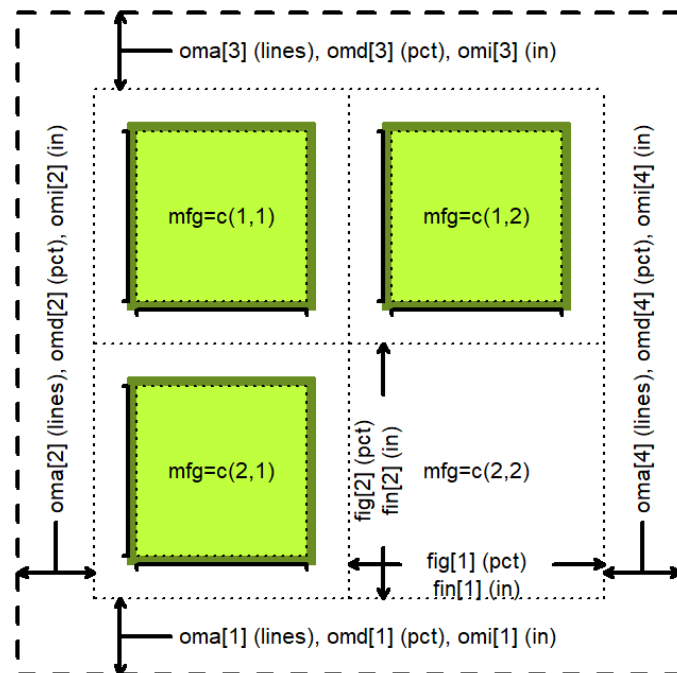
If you display multiple plots within a single graphics window (e.g., with the `mfw` or `mfc` arguments of `par` or with the `layout` function), then the `fig` and `fin` arguments will tell you the size of the current subplot window in percent or inches, respectively.

`par("oma")` (bottom, left, top, right) in lines  
`par("omd")` (bottom, left, top, right) in pct  
`par("omi")` (bottom, left, top, right) in inches

Each subplot will have margins specified by `mai` or `mar`, but no outer margin around the entire set of plots, unless you specify them using `oma`, `omd`, or `omi`. You can place text in the outer margins using the `mtext` function with the argument `outer=TRUE`.

`par("mfg")` (r, c) or (r, c, maxr, maxc)

The `mfg` argument of `par` will allow you to jump to a subplot in a particular row and column. If you query with `par("mfg")`, you will get the current row and column followed by the maximum row and column.



## Character and string sizes

### `strheight()`

The `strheight` functions will tell you the height of a specified string in inches (`units="inches"`), x-y user coordinates (`units="user"`) or as a percentage of the graphics device (`units="figure"`).

For a single line of text, `strheight` will give you the height of the letter "M". If you have a string with one or more linebreaks (`"\n"`), the `strheight` function will measure the height of the letter "M" plus the height of one or more additional lines. The height of a line is dependent on the line spacing, set by the `lheight` argument of `par`. The default line height (`lheight=1`), corresponding to single spaced lines, produces a line height roughly 1.5 times the height of "M".

### `strwidth()`

The `strwidth` function will produce different widths to individual characters, representing the proportional spacing used by most fonts (a "W" using much more space than an "i"). For the width of a string, the `strwidth` function will sum up the lengths of the individual characters in the string.

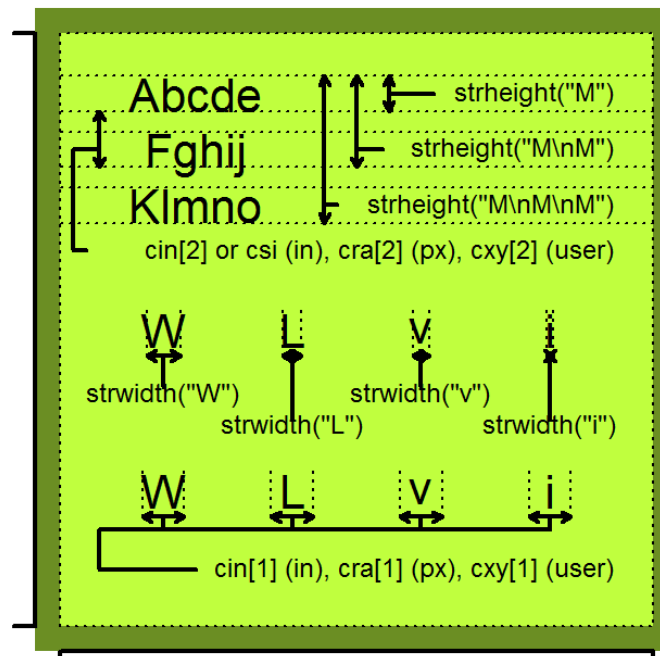
`par("cin")` (r.o.) (width, height) in inches  
`par("csi")` (r.o.) height in inches  
`par("cra")` (r.o.) (width, height) in pixels  
`par("cxy")` (r.o.) (width, height) in xy coordinates

The single value returned by the `csi` argument of `par` gives you the height of a line of text in inches. The second of the two values returned by `cin`, `cra`, and `cxy` gives you the height of a line, in inches, pixels, or xy (user) coordinates.

The first of the two values returned by the `cin`, `cra`, and `cxy` arguments to `par` gives you the approximate width of a single character, in inches, pixels, or xy (user) coordinates. The width, very slightly smaller than the actual width of the letter "W", is a rough estimate at best and ignores the variable width of individual letters.

These values are useful, however, in providing fast ratios of the relative sizes of the differing units of measure

```
px.per.in <- par("cra") / par("cin")
py.per.in <- par("cxy") / par("cin")
```



## If your fonts are too big or too small

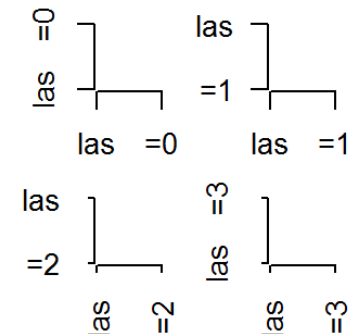
Fixing this takes a bit of trial and error.

1. Specify a larger/smaller value for the `pointsize` argument when you open your graphics device.
2. Trying opening your graphics device with different values for `height` and `width`. Fonts that look too big might be better proportioned in a larger graphics window.
3. Use the `cex` argument to increase or decrease the relative size of your fonts.

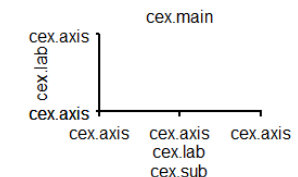
## If your axes don't fit

There are several possible solutions.

1. You can assign wider margins using the `mar` or `mai` argument in `par`.
2. You can change the orientation of the axis labels with `las`. Choose among
  - a. `las=0` both axis labels parallel
  - b. `las=1` both axis labels horizontal
  - c. `las=2` both axis labels perpendicular
  - d. `las=3` both axis labels vertical.



3. change the relative size of the font
  - a. `cex.axis` for the tick mark labels.
  - b. `cex.lab` for `xlab` and `ylab`.
  - c. `cex.main` for the main title
  - d. `cex.sub` for the subtitle.

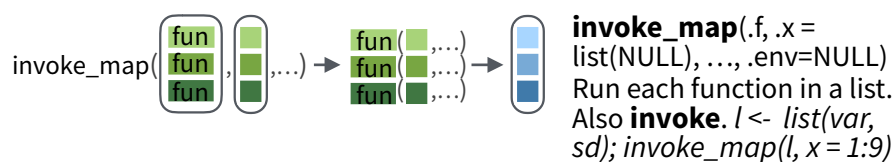
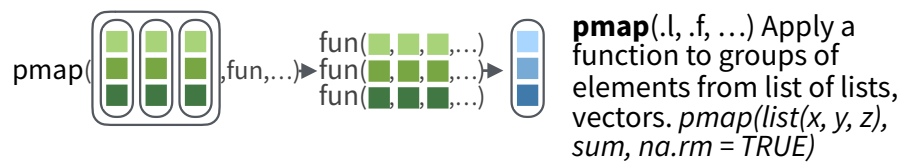
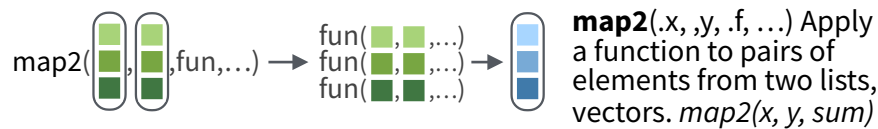
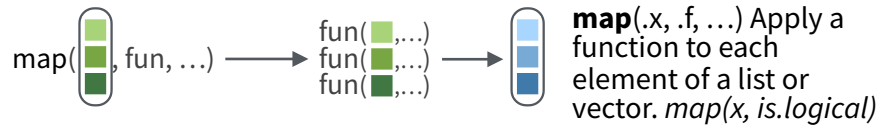


# Apply functions with purrr : : CHEAT SHEET



## Apply Functions

Map functions apply a function iteratively to each element of a list or vector.



**lmap**(.x, .f, ...) Apply function to each list-element of a list or vector.  
**imap**(.x, .f, ...) Apply .f to each element of a list or vector and its index.

### OUTPUT

**map()**, **map2()**, **pmap()**, **imap** and **invoke\_map** each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. **map2\_chr**, **pmap\_lgl**, etc.

Use **walk**, **walk2**, and **pwalk** to trigger side effects. Each return its input invisibly.

function	returns
<b>map</b>	list
<b>map_chr</b>	character vector
<b>map_dbl</b>	double (numeric) vector
<b>map_dfc</b>	data frame (column bind)
<b>map_dfr</b>	data frame (row bind)
<b>map_int</b>	integer vector
<b>map_lgl</b>	logical vector
<b>walk</b>	triggers side effects, returns the input invisibly

### SHORTCUTS - within a purrr function:

"name" becomes **function(x) x\$name**. e.g. *map(l, "a")* extracts \$a from each element of l

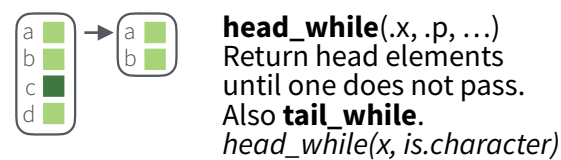
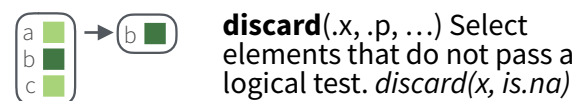
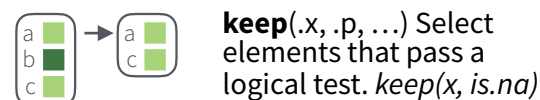
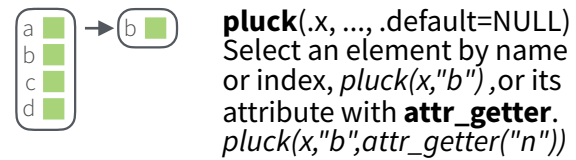
~ .x .y becomes **function(x, y) .x .y**. e.g. *map2(l, p, ~.x+.y)* becomes *map2(l, p, function(l, p) l + p)*

~ . becomes **function(x) x**. e.g. *map(l, ~ 2 + .)* becomes *map(l, function(x) 2 + x)*

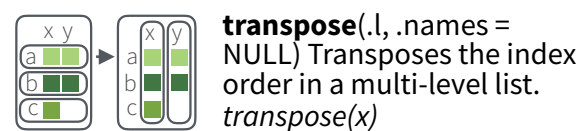
~ ..1 ..2 etc becomes **function(..1, ..2, etc) ..1 ..2 etc**. e.g. *pmap(list(a, b, c), ~ ..3 + ..1 - ..2)* becomes *pmap(list(a, b, c), function(a, b, c) c + a - b)*

## Work with Lists

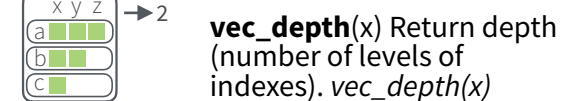
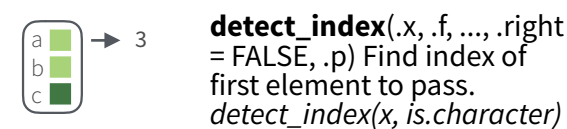
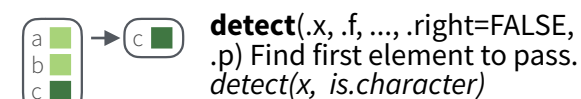
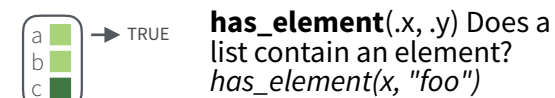
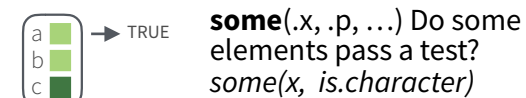
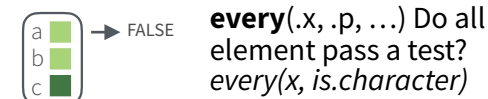
### FILTER LISTS



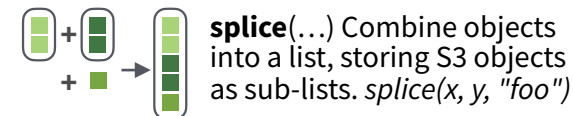
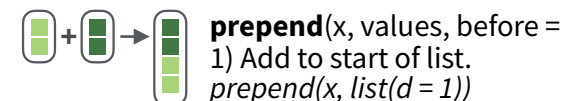
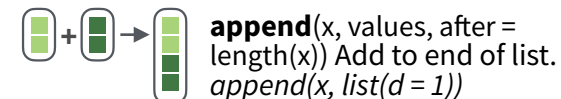
### RESHAPE LISTS



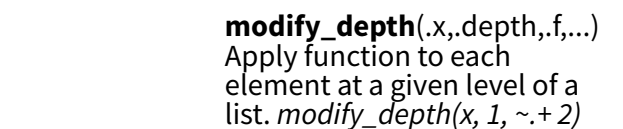
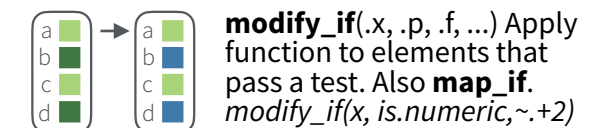
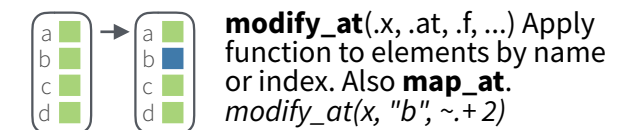
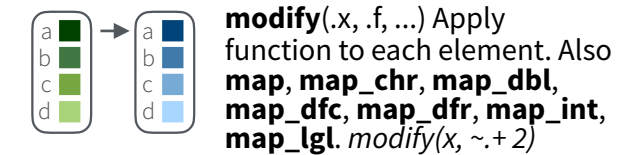
### SUMMARISE LISTS



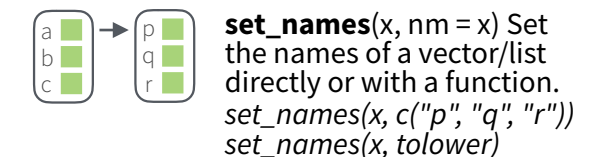
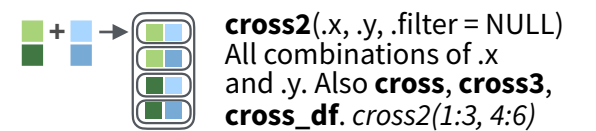
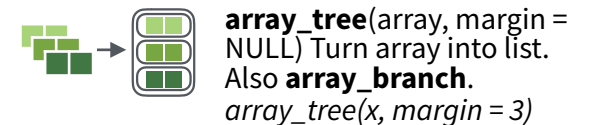
### JOIN (TO) LISTS



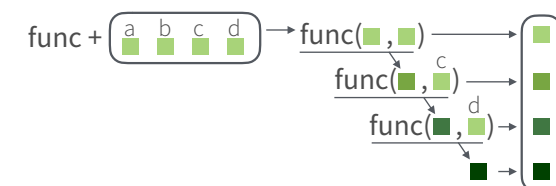
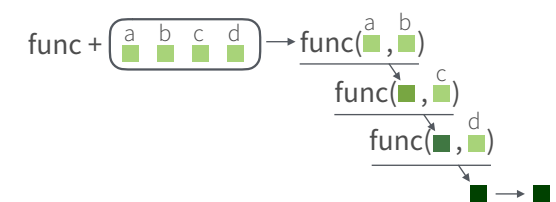
### TRANSFORM LISTS



### WORK WITH LISTS



## Reduce Lists



## Modify function behavior

**compose**() Compose multiple functions.

**lift**() Change the type of input a function takes. Also **lift\_dl**, **lift\_dv**, **lift\_ld**, **lift\_lv**, **lift\_vd**, **lift\_vl**.

**rerun**() Rerun expression n times.

**negate**() Negate a predicate function (a pipe friendly!)

**partial**() Partially apply a function, filling in some args.

**safely**() Modify func to return list of results and errors.

**quietly**() Modify function to return list of results, output, messages, warnings.

**possibly**() Modify function to return default value whenever an error occurs (instead of error).





# Nested Data

A **nested data frame** stores individual tables within the cells of a larger, organizing table.

Species	data
setosa	<tibble [50 x 4]>
versicolor	<tibble [50 x 4]>
virginica	<tibble [50 x 4]>

n\_iris

"cell" contents

Sepal.L	Sepal.W	Petal.L	Petal.W
5.1	3.5	1.4	0.2
4.9	3.0	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5.0	3.6	1.4	0.2

n\_iris\$data[[1]]

Sepal.L	Sepal.W	Petal.L	Petal.W
7.0	3.2	4.7	1.4
6.4	3.2	4.5	1.5
6.9	3.1	4.9	1.5
5.5	2.3	4.0	1.3
6.5	2.8	4.6	1.5

n\_iris\$data[[2]]

Sepal.L	Sepal.W	Petal.L	Petal.W
6.3	3.3	6.0	2.5
5.8	2.7	5.1	1.9
7.1	3.0	5.9	2.1
6.3	2.9	5.6	1.8
6.5	3.0	5.8	2.2

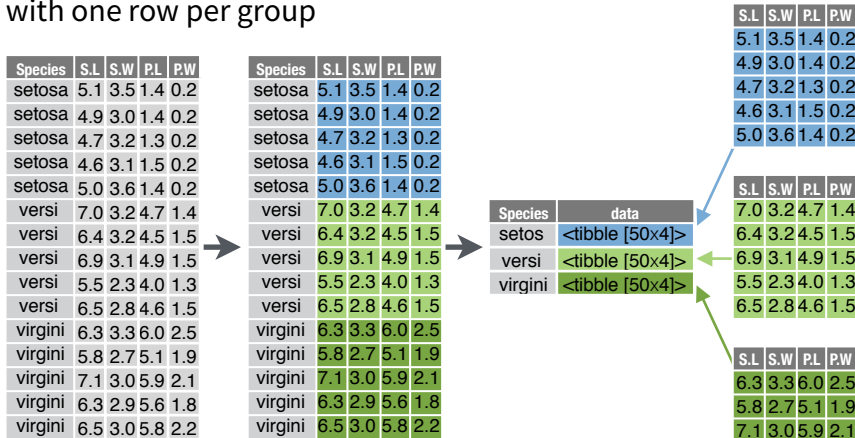
n\_iris\$data[[3]]

Use a nested data frame to:

- preserve relationships between observations and subsets of data
- manipulate many sub-tables at once with the **purrr** functions **map()**, **map2()**, or **pmap()**.

Use a two step process to create a nested data frame:

1. Group the data frame into groups with **dplyr::group\_by()**
2. Use **nest()** to create a nested data frame with one row per group



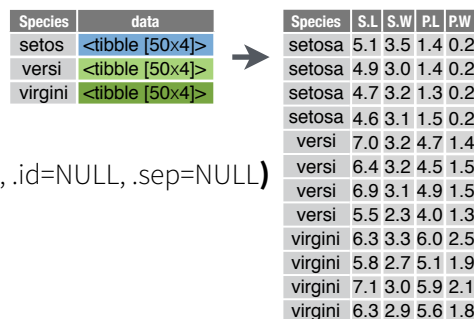
```
n_iris <- iris %>% group_by(Species) %>% nest()
```

**tidyr::nest**(data, ..., .key = data)  
For grouped data, moves groups into cells as data frames.

Unnest a nested data frame with **unnest()**:

```
n_iris %>% unnest()
```

**tidyr::unnest**(data, ..., .drop = NA, .id=NULL, .sep=NULL)  
Unnests a nested data frame.



# List Column Workflow

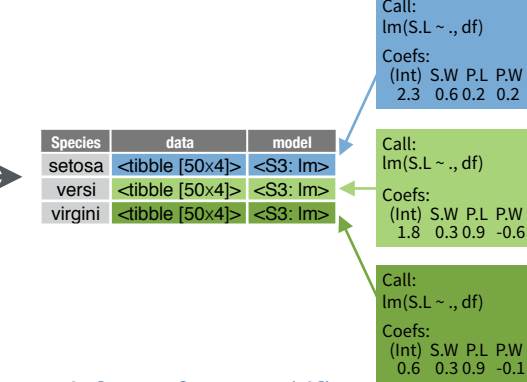
Nested data frames use a **list column**, a list that is stored as a column vector of a data frame. A typical **workflow** for list columns:

## 1 Make a list column

Species	S.L	S.W	P.L	P.W
setosa	5.1	3.5	1.4	0.2
setosa	4.9	3.0	1.4	0.2
setosa	4.7	3.2	1.3	0.2
setosa	4.6	3.1	1.5	0.2
setosa	5.0	3.6	1.4	0.2
versi	7.0	3.2	4.7	1.4
versi	6.4	3.2	4.5	1.5
versi	6.9	3.1	4.9	1.5
versi	5.5	2.3	4.0	1.3
virgini	6.3	3.3	6.0	2.5
virgini	5.8	2.7	5.1	1.9
virgini	7.1	3.0	5.9	2.1
virgini	6.3	2.9	5.6	1.8

```
n_iris <- iris %>%
 group_by(Species) %>%
 nest()
```

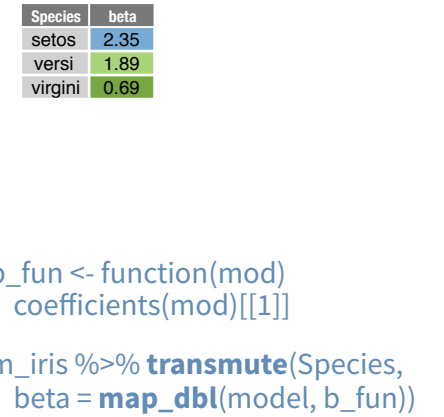
## 2 Work with list columns



```
mod_fun <- function(df)
 lm(Sepal.Length ~ ., data = df)

m_iris <- n_iris %>%
 mutate(model = map(data, mod_fun))
```

## 3 Simplify the list column



```
b_fun <- function(mod)
 coefficients(mod)[[1]]

m_iris %>% transmute(Species,
 beta = map_dbl(model, b_fun))
```

**1. MAKE A LIST COLUMN** - You can create list columns with functions in the **tibble** and **dplyr** packages, as well as **tidyr**'s **nest()**

**tibble::tribble(...)**

Makes list column when needed

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

**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 tibble with list cols

```
enframe(list('3'=1:3, '4'=1:4, '5'=1:5), 'max', 'seq')
```

**dplyr::mutate(data, ...)** Also **transmute()**

Returns list col when result returns list.

```
mtcars %>% mutate(seq = map(cyl, seq))
```

**dplyr::summarise(data, ...)**

Returns list col when result is wrapped with **list()**

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

**2. WORK WITH LIST COLUMNS** - Use the **purrr** functions **map()**, **map2()**, and **pmap()** to apply a function that returns a result element-wise to the cells of a list column. **walk()**, **walk2()**, and **pwalk()** work the same way, but return a side effect.

**purrr::map(x, .f, ...)**

Apply .f element-wise to .x as .f(.x)

```
n_iris %>% mutate(n = map(data, dim))
```

**purrr::map2(x, .y, .f, ...)**

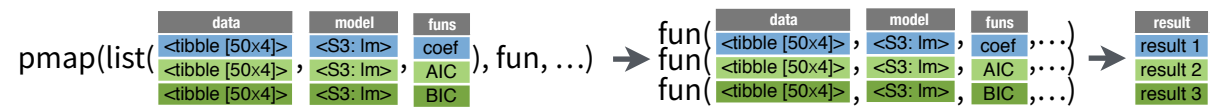
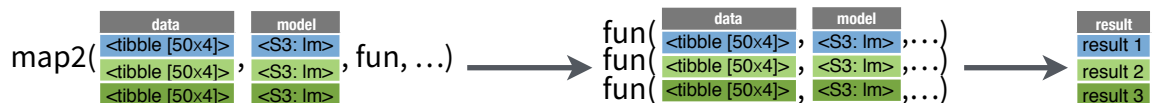
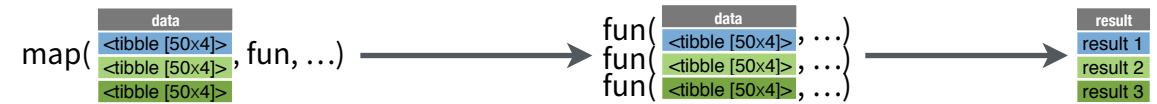
Apply .f element-wise to .x and .y as .f(.x, .y)

```
m_iris %>% mutate(n = map2(data, model, list))
```

**purrr::pmap(.l, .f, ...)**

Apply .f element-wise to vectors saved in .l

```
m_iris %>%
 mutate(n = pmap(list(data, model, data), list))
```



**3. SIMPLIFY THE LIST COLUMN** (into a regular column)

Use the **purrr** functions **map\_lgl()**, **map\_int()**, **map\_dbl()**, **map\_chr()**, as well as **tidyr**'s **unnest()** to reduce a list column into a regular column.

**purrr::map\_lgl(x, .f, ...)**

Apply .f element-wise to .x, return a logical vector

```
n_iris %>% transmute(n = map_lgl(data, is.matrix))
```

**purrr::map\_int(x, .f, ...)**

Apply .f element-wise to .x, return an integer vector

```
n_iris %>% transmute(n = map_int(data, nrow))
```

**purrr::map\_dbl(x, .f, ...)**

Apply .f element-wise to .x, return a double vector

```
n_iris %>% transmute(n = map_dbl(data, nrow))
```

**purrr::map\_chr(x, .f, ...)**

Apply .f element-wise to .x, return a character vector

```
n_iris %>% transmute(n = map_chr(data, nrow))
```



# Work with strings with stringr : : CHEAT SHEET

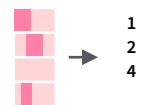


The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

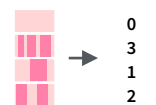
## Detect Matches



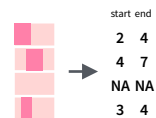
**str\_detect**(string, pattern) Detect the presence of a pattern match in a string. `str_detect(fruit, "a")`



**str\_which**(string, pattern) Find the indexes of strings that contain a pattern match. `str_which(fruit, "a")`



**str\_count**(string, pattern) Count the number of matches in a string. `str_count(fruit, "a")`

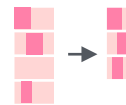


**str\_locate**(string, pattern) Locate the positions of pattern matches in a string. Also **str\_locate\_all**. `str_locate(fruit, "a")`

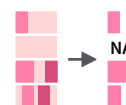
## Subset Strings



**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) Return only the strings that contain a pattern match. `str_subset(fruit, "b")`

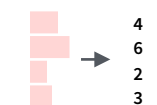


**str\_extract**(string, pattern) Return the first pattern match found in each string, as a vector. Also **str\_extract\_all** to return every pattern match. `str_extract(fruit, "[aeiou]")`

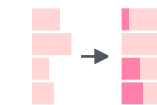


**str\_match**(string, pattern) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str\_match\_all**. `str_match(sentences, "(a|the) ([^ ]+)")`

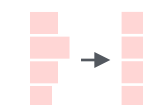
## Manage Lengths



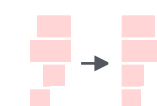
**str\_length**(string) The width of strings (i.e. number of code points, which generally equals the number of characters). `str_length(fruit)`



**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(fruit, 3)`

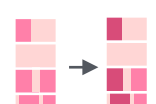


**str\_trim**(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. `str_trim(fruit)`

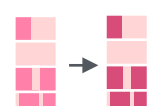
## Mutate Strings



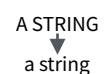
**str\_sub()** <- value. Replace substrings by identifying the substrings with `str_sub()` and assigning into the results. `str_sub(fruit, 1, 3) <- "str"`



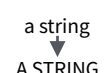
**str\_replace**(string, pattern, replacement) Replace the first matched pattern in each string. `str_replace(fruit, "a", "-")`



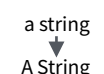
**str\_replace\_all**(string, pattern, replacement) Replace all matched patterns in each string. `str_replace_all(fruit, "a", "-")`



**str\_to\_lower**(string, locale = "en")<sup>1</sup> Convert strings to lower case. `str_to_lower(sentences)`

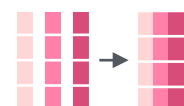


**str\_to\_upper**(string, locale = "en")<sup>1</sup> Convert strings to upper case. `str_to_upper(sentences)`



**str\_to\_title**(string, locale = "en")<sup>1</sup> Convert strings to title case. `str_to_title(sentences)`

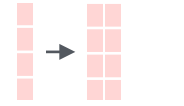
## Join and Split



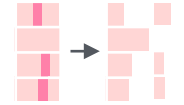
**str\_c**(..., sep = "", collapse = NULL) Join multiple strings into a single string. `str_c(letters, LETTERS)`



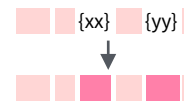
**str\_c**(..., sep = "", collapse = NULL) Collapse a vector of strings into a single string. `str_c(letters, collapse = "")`



**str\_dup**(string, times) Repeat strings times times. `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. `str_split_fixed(fruit, " ", n=2)`



**glue::glue**(..., .sep = "", .envir = parent.frame(), .open = "{", .close = "}") Create a string from strings and {expressions} to evaluate. `glue::glue("Pi is {pi}")`



**glue::glue\_data**(.x, ..., .sep = "", .envir = parent.frame(), .open = "{", .close = "}") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. `glue::glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")`

## Order Strings

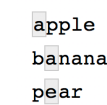


**str\_order**(x, decreasing = FALSE, na\_last = TRUE, locale = "en", numeric = FALSE, ...) <sup>1</sup> Return the vector of indexes that sorts a character vector. `x[str_order(x)]`

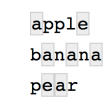


**str\_sort**(x, decreasing = FALSE, na\_last = TRUE, locale = "en", numeric = FALSE, ...) <sup>1</sup> Sort a character vector. `str_sort(x)`

## 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 first regex match in each string. `str_view(fruit, "[aeiou]")`

**str\_view\_all**(string, pattern, match = NA) View HTML rendering of all regex matches. `str_view_all(fruit, "[aeiou]")`

**str\_wrap**(string, width = 80, indent = 0, exdent = 0) 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 stringr are interpreted as regular expressions *after any special characters have been parsed*.

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes ("" or single quotes(')).

Some characters cannot be represented directly in an R string. These must be represented as **special characters**, sequences of characters that have a specific meaning, e.g.

Special Character	Represents
\\	\
\"	"
\\n	new line

Run `?""` to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

Use `writeLines()` to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
\.
```

```
writeLines("\\ is a backslash")
\ is a backslash
```

## INTERPRETATION

Patterns in stringr are interpreted as regexs To change this default, wrap the pattern in one of:

**regex()** (pattern, ignore\_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...) Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's, and/or to have . match everything including \n. `str_detect("I", regex("i", TRUE))`

**fixed()** Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). `str_detect("\u0130", fixed("i"))`

**coll()** Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow). `str_detect("\u0130", coll("i", TRUE, locale = "tr"))`

**boundary()** Matches boundaries between characters, line\_breaks, sentences, or words. `str_split(sentences, boundary("word"))`

# Regular Expressions - Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

## MATCH CHARACTERS

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

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

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

**[:space:]**  
← new line  
space  
tab

**[:blank:]**  
space  
tab

**[:graph:]**

**[:punct:]**  
.,:;?! \ | / ` = \* + - ^ \_ ~ " ' [ ] { } ( ) < > @ # \$

**[:alnum:]**

**[:digit:]**  
0 1 2 3 4 5 6 7 8 9

**[:alpha:]**

[: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
z	Z

## ALTERNATES

```
alt <- function(rx) str_view_all("abcde", rx)
```

regex	matches	example
<code>ab d</code>	or	<code>alt("ab d")</code>
<code>[abe]</code>	one of	<code>alt("[abe]")</code>
<code>^[abe]</code>	anything but	<code>alt("^[abe]")</code>
<code>[a-c]</code>	range	<code>alt("[a-c]")</code>

## ANCHORS

```
anchor <- function(rx) str_view_all("aaa", rx)
```

regex	matches	example
<code>^a</code>	start of string	<code>anchor("^a")</code>
<code>a\$</code>	end of string	<code>anchor("a\$")</code>

## LOOK AROUNDS

```
look <- function(rx) str_view_all("bacad", rx)
```

regex	matches	example
<code>a(=?c)</code>	followed by	<code>look("a(=?c)")</code>
<code>a(?!c)</code>	not followed by	<code>look("a(?!c)")</code>
<code>(?&lt;=b)a</code>	preceded by	<code>look("(?&lt;=b)a")</code>
<code>(?!b)a</code>	not preceded by	<code>look("(?!b)a")</code>

## QUANTIFIERS

```
quant <- function(rx) str_view_all(".a.aa.aaa", rx)
```

regex	matches	example
<code>a?</code>	zero or one	<code>quant("a?")</code>
<code>a*</code>	zero or more	<code>quant("a*")</code>
<code>a+</code>	one or more	<code>quant("a+")</code>
<code>a{n}</code>	exactly n	<code>quant("a{2}")</code>
<code>a{n,}</code>	n or more	<code>quant("a{2,}")</code>
<code>a{n,m}</code>	between n and m	<code>quant("a{2,4}")</code>

## GROUPS

```
ref <- function(rx) str_view_all("abbaab", rx)
```

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

regex	matches	example
<code>(ab c)e</code>	sets precedence	<code>alt("(ab c)e")</code>

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

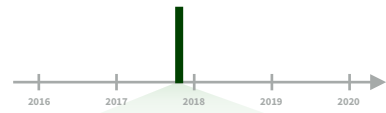
string (type this)	regex (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))
<code>\\1</code>	<code>\\1 (etc.)</code>	first () group, etc.	<code>ref("(a)(b)\\2\\1")</code>



# Dates and times with lubridate : : CHEAT SHEET



## 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 wide variety of input formats.

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")`

2:01 **hms::hms()** Also `lubridate::hms()`, **hm()** and **ms()**, which return periods. `hms::hms(sec = 0, min = 1, hours = 2)`

2017.5 **date\_decimal()**(decimal, tz = "UTC")  
Q for quarter. `date_decimal(2017.5)`

**now**(tzzone = "") Current time in tz (defaults to system tz). `now()`

**today**(tzzone = "") Current date in a tz (defaults to system tz). `today()`

**fast\_strptime()** Faster `strptime`.  
`fast_strptime("9/1/01", "%y/%m/%d")`

**parse\_date\_time()** Easier `strptime`.  
`parse_date_time("9/1/01", "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"
```

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"
```

### GET AND SET COMPONENTS

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

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

2018-01-31 11:59:59 **date(x)** Date component. `date(dt)`

2018-01-31 11:59:59 **year(x)** Year. `year(dt)`  
**isoyear(x)** The ISO 8601 year.  
**epiyear(x)** Epidemiological year.

2018-01-31 11:59:59 **month(x, label, abbr)** Month.  
`month(dt)`

2018-01-31 11:59:59 **day(x)** Day of month. `day(dt)`  
**wday(x, label, abbr)** Day of week.  
**qday(x)** Day of quarter.

2018-01-31 11:59:59 **hour(x)** Hour. `hour(dt)`

2018-01-31 11:59:59 **minute(x)** Minutes. `minute(dt)`

2018-01-31 11:59:59 **second(x)** Seconds. `second(dt)`

**week(x)** Week of the year. `week(dt)`  
**isoweek()** ISO 8601 week.  
**epiweek()** Epidemiological week.

**quarter(x, with\_year = FALSE)**  
Quarter. `quarter(dt)`

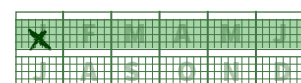
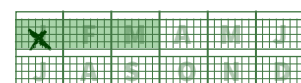
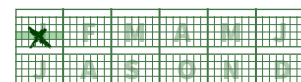
**semester(x, with\_year = FALSE)**  
Semester. `semester(dt)`

**am(x)** Is it in the am? `am(dt)`  
**pm(x)** Is it in the pm? `pm(dt)`

**dst(x)** Is it daylight savings? `dst(d)`

**leap\_year(x)** Is it a leap year?  
`leap_year(d)`

**update(object, ..., simple = FALSE)**  
`update(dt, mday = 2, hour = 1)`



## Round Date-times



**floor\_date(x, unit = "second")**  
Round down to nearest unit.  
`floor_date(dt, unit = "month")`



**round\_date(x, unit = "second")**  
Round to nearest unit.  
`round_date(dt, unit = "month")`



**ceiling\_date(x, unit = "second")**,  
`change_on_boundary = NULL`  
Round up to nearest unit.  
`ceiling_date(dt, unit = "month")`

**rollback(dates, roll\_to\_first = FALSE, preserve\_hms = TRUE)**  
Roll back to last day of previous month. `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"`

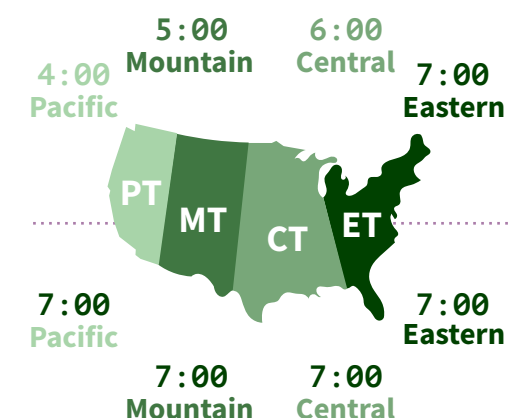
Tip: use a date with day > 12

## 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()`



**with\_tz(time, tzzone = "")** Get the **same instant** in a new time zone (a new clock time).  
`with_tz(dt, "US/Pacific")`

**force\_tz(time, tzzone = "")** Get the **same clock time** in a new time zone (a new instant).  
`force_tz(dt, "US/Pacific")`

# Math with Date-times

— Lubridate provides three classes of timespans to facilitate math with dates and date-times



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

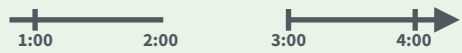
A normal day

```
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")
```



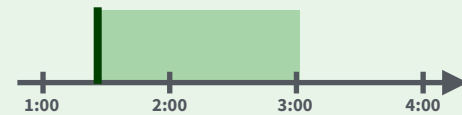
Leap years and leap seconds

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

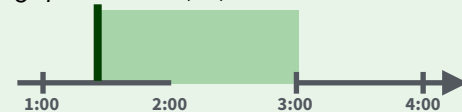


**Periods** track changes in clock times, which ignore time line irregularities.

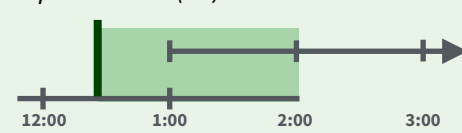
*normal + minutes(90)*



*gap + minutes(90)*



*lap + minutes(90)*

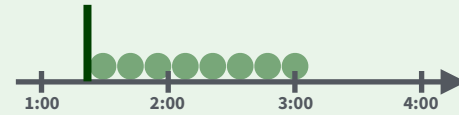


*leap + years(1)*

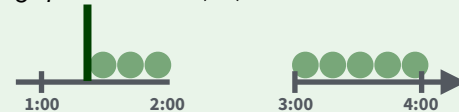


**Durations** track the passage of physical time, which deviates from clock time when irregularities occur.

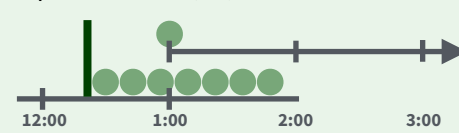
*normal + dminutes(90)*



*gap + dminutes(90)*



*lap + dminutes(90)*

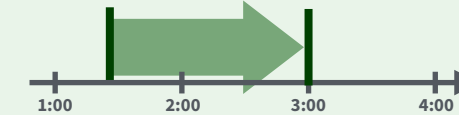


*leap + dyears(1)*



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

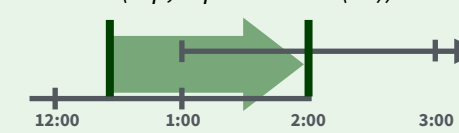
*interval(normal, normal + minutes(90))*



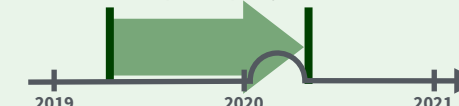
*interval(gap, gap + minutes(90))*



*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"
```



- 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**(i). *as.period(i)*

**period\_to\_seconds**(x) Convert a period to the "standard" number of seconds implied by the period. Also **seconds\_to\_period**(i). *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. **Diffimes** 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)"
```



- dyears**(x = 1) 31536000x 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<sup>-3</sup> seconds.
- dmicroseconds**(x = 1) x × 10<sup>-6</sup> seconds.
- dnanoseconds**(x = 1) x × 10<sup>-9</sup> seconds.
- dpicoseconds**(x = 1) x × 10<sup>-12</sup> 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**(i), **is.diffime**(i). *as.duration(i)*

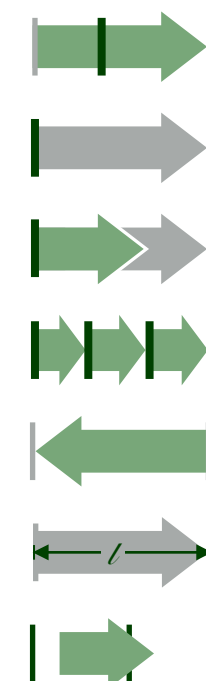
**make\_diffime**(x) Make diffime with the specified number of units. *make\_diffime(99999)*

## INTERVALS

Divide an interval by a duration to determine its physical length, divide and 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
```



**%within%** b Does interval or date-time a fall within interval b? *now() %within% i*

**int\_start**(int) Access/set the start date-time of an interval. Also **int\_end**(i). *int\_start(i) <- now(); int\_start(i)*

**int\_aligns**(int1, int2) Do two intervals share a boundary? Also **int\_overlaps**(i, j). *int\_aligns(i, j)*

**int\_diff**(times) Make the intervals that occur between the date-times in a vector. *v <- c(dt, dt + 100, dt + 1000); int\_diff(v)*

**int\_flip**(int) Reverse the direction of an interval. Also **int\_standardize**(i). *int\_flip(i)*

**int\_length**(int) Length in seconds. *int\_length(i)*

**int\_shift**(int, by) Shifts an interval up or down the timeline by a timespan. *int\_shift(i, days(-1))*

**as.interval**(x, start, ...) Coerce a timespan to an interval with the start date-time. Also **is.interval**(i). *as.interval(days(1), start = now())*





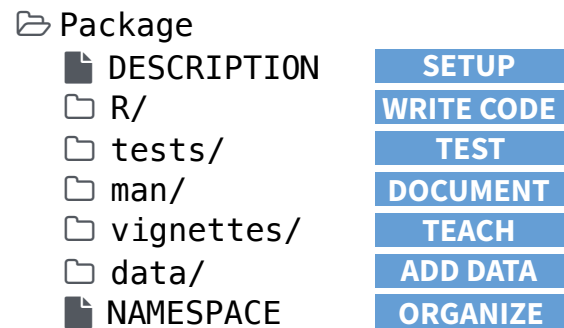
# Package Development: : CHEAT SHEET



## Package Structure

A package is a convention for organizing files into directories.

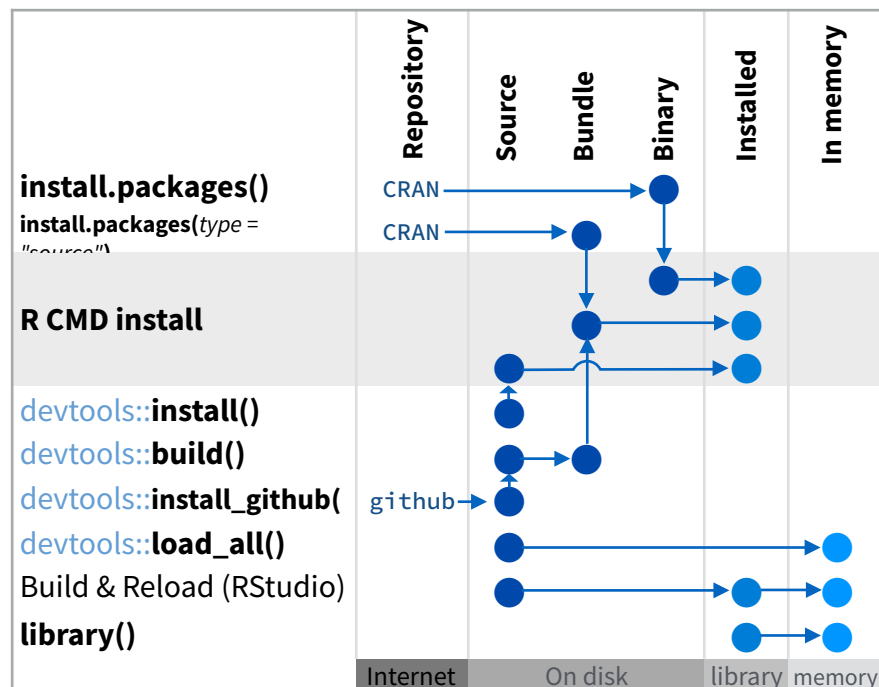
This sheet shows how to work with the 7 most common parts of an R package:



The contents of a package can be stored on disk as a:

- **source** - a directory with sub-directories (as above)
- **bundle** - a single compressed file (.tar.gz)
- **binary** - a single compressed file optimized for a specific OS

Or installed into an R library (loaded into memory during an R session) or archived online in a repository. Use the functions below to move between these states.



`devtools::add_build_ignore("file")`

Adds file to .Rbuildignore, a list of files that will not be included when package is built.



## Setup (DESCRIPTION)

The DESCRIPTION file describes your work, sets up how your package will work with other packages, and applies a copyright.

- ✓ You must have a DESCRIPTION file
- ✓ Add the packages that yours relies on with `devtools::use_package()`  
Adds a package to the Imports or Suggests field

CC0	MIT	GPL-2
No strings attached.	MIT license applies to your code if re-shared.	GPL-2 license applies to your code, and all code anyone bundles with it, if re-shared.

```
Package: mypackage
Title: Title of Package
Version: 0.1.0
Authors@R: person("Hadley", "Wickham", email = "hadley@me.com", role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: GPL-2
LazyData: true
Imports:
 dplyr (>= 0.4.0),
 ggvis (>= 0.2)
Suggests:
 knitr (>= 0.1.0)
```

**Import** packages that your package *must have* to work. R will install them when it installs your package.

**Suggest** packages that are not very essential to yours. Users can install them manually, or not, as they like.

## Write Code (R/)

All of the R code in your package goes in R/. A package with just an R/ directory is still a very useful package.

- ✓ Create a new package project with `devtools::create("path/to/name")`  
Create a template to develop into a package.
- ✓ Save your code in R/ as scripts (extension .R)

### WORKFLOW

1. Edit your code.
2. Load your code with one of `devtools::load_all()`  
Re-loads all saved files in R/ into memory.  
**Ctrl/Cmd + Shift + L** (keyboard shortcut)  
Saves all open files then calls load\_all().
3. Experiment in the console.
4. Repeat.

- Use consistent style with [r-pkgs.had.co.nz/r.html#style](http://r-pkgs.had.co.nz/r.html#style)
- Click on a function and press **F2** to open its definition
- Search for a function with **Ctrl + .**



Visit [r-pkgs.had.co.nz](http://r-pkgs.had.co.nz) to learn much more about writing and publishing packages for R

## Test (tests/)

Use tests/ to store tests that will alert you if your code breaks.

- ✓ Add a tests/ directory
- ✓ Import testthat with `devtools::use_testthat()`, which sets up package to use automated tests with testthat
- ✓ Write tests with `context()`, `test()`, and expect statements
- ✓ Save your tests as .R files in tests/testthat/

### WORKFLOW

1. Modify your code or tests.
2. Test your code with one of `devtools::test()`  
Runs all tests in tests/  
**Ctrl/Cmd + Shift + T** (keyboard shortcut)
3. Repeat until all tests pass

### Example Test

```
context("Arithmetic")
test_that("Math works", {
 expect_equal(1 + 1, 2)
 expect_equal(1 + 2, 3)
 expect_equal(1 + 3, 4)
})
```

Expect statement	Tests
<code>expect_equal()</code>	is equal within small numerical tolerance?
<code>expect_identical()</code>	is exactly equal?
<code>expect_match()</code>	matches specified string or regular
<code>expect_output()</code>	prints specified output?
<code>expect_message()</code>	displays specified message?
<code>expect_warning()</code>	displays specified warning?
<code>expect_error()</code>	throws specified error?
<code>expect_is()</code>	output inherits from certain class?
<code>expect_false()</code>	returns FALSE?
<code>expect_true()</code>	returns TRUE?

## Document (man/)

man/ contains the documentation for your functions, the help pages in your package.

- Use roxygen comments to document each function beside its definition
- Document the name of each exported data set
- Include helpful examples for each function

### WORKFLOW

- Add roxygen comments in your .R files
- Convert roxygen comments into documentation with one of:

#### devtools::document()

Converts roxygen comments to .Rd files and places them in man/. Builds NAMESPACE.

#### Ctrl/Cmd + Shift + D (Keyboard Shortcut)

- Open help pages with ? to preview documentation
- Repeat

### .Rd FORMATTING TAGS

<code>\emph{italic text}</code>	<code>\email{name@@foo.com}</code>
<code>\strong{bold text}</code>	<code>\href{url}{display}</code>
<code>\code{function(args)}</code>	<code>\url{url}</code>
<code>\pkg{package}</code>	
	<code>\link[=dest]{display}</code>
<code>\dontrun{code}</code>	<code>\linkS4class{class}</code>
<code>\dontshow{code}</code>	<code>\code{\link{function}}</code>
<code>\donttest{code}</code>	<code>\code{\link[package]{function}}</code>
<code>\deqn{a + b (block)}</code>	<code>\tabular{lcr}{</code>
<code>\eqn{a + b (inline)}</code>	<code>left \tab centered \tab right \cr</code>
	<code>cell \tab cell \tab cell \cr</code>
	<code>}</code>

## Teach (vignettes/)

vignettes/ holds documents that teach your users how to solve real problems with your tools.

- Create a vignettes/ directory and a template vignette with `devtools::use_vignette()`  
Adds template vignette as vignettes/my-vignette.Rmd.
- Append YAML headers to your vignettes (like right)
- Write the body of your vignettes in R Markdown ([rmarkdown.rstudio.com](http://rmarkdown.rstudio.com))

```

title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
 %\VignetteIndexEntry{Vignette Title}
 %\VignetteEngine{knitr::rmarkdown}
 \usepackage[utf8]{inputenc}

```

### ROXYGEN2

The **roxygen2** package lets you write documentation inline in your .R files with a shorthand syntax. devtools implements roxygen2 to make documentation.



- Add roxygen documentation as comment lines that begin with #.
- Place comment lines directly above the code that defines the object documented.
- Place a roxygen @ tag (right) after # to supply a specific section of documentation.
- Untagged lines will be used to generate a title, description, and details section (in that order)

```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' @export
add <- function(x, y) {
 x + y
}
```

### COMMON ROXYGEN TAGS

@aliases	@inheritParams	@seealso	
@concepts	@keywords	@format	
@describeIn	@param	@source	data
@examples	@rdname	@include	
@export	@return	@slot	S4
@family	@section	@field	RC

## Add Data (data/)

The data/ directory allows you to include data with your package.



- Save data as .Rdata files (suggested)
- Store data in one of **data/**, **R/Sysdata.rda**, **inst/extdata**
- Always use **LazyData: true** in your DESCRIPTION file.

#### devtools::use\_data()

Adds a data object to data/ (R/Sysdata.rda if **internal = TRUE**)

#### devtools::use\_data\_raw()

Adds an R Script used to clean a data set to data-raw/. Includes data-raw/ on .Rbuildignore.

Store data in

- data/** to make data available to package users
- R/sysdata.rda** to keep data internal for use by your functions.
- inst/extdata** to make raw data available for loading and parsing examples. Access this data with **system.file()**

## Organize (NAMESPACE)

The NAMESPACE file helps you make your package self-contained: it won't interfere with other packages, and other packages won't interfere with it.

- Export functions for users by placing **@export** in their roxygen comments
- Import objects from other packages with **package::object** (recommended) or **@import**, **@importFrom**, **@importClassesFrom**, **@importMethodsFrom** (not always recommended)

### WORKFLOW

- Modify your code or tests.
- Document your package (`devtools::document()`)
- Check NAMESPACE
- Repeat until NAMESPACE is correct

### SUBMIT YOUR PACKAGE

[r-pkgs.had.co.nz/release.html](http://r-pkgs.had.co.nz/release.html)



Notes:



RStudio Community - [rstudio.link/community](https://rstudio.link/community)

Developer Blog - [rstudio.link/developer\\_blog](https://rstudio.link/developer_blog)

Tensorflow Blog - [rstudio.link/tensorflow\\_blog](https://rstudio.link/tensorflow_blog)

R Views Blog - [rstudio.link/rviews\\_blog](https://rstudio.link/rviews_blog)

Twitter - [rstudio.link/twitter](https://rstudio.link/twitter)

GitHub - [rstudio.link/github](https://rstudio.link/github)

LinkedIn - [rstudio.link/linkedin](https://rstudio.link/linkedin)

YouTube - [rstudio.link/youtube](https://rstudio.link/youtube)

Facebook - [rstudio.link/facebook](https://rstudio.link/facebook)