RStudio IDE Cheat Sheet

learn more at www.rstudio.com

RStudio

The RStudio IDE is an Integrated Development Environment in R that comes in three versions

Desktop IDE
A local version of the IDE for your desktop

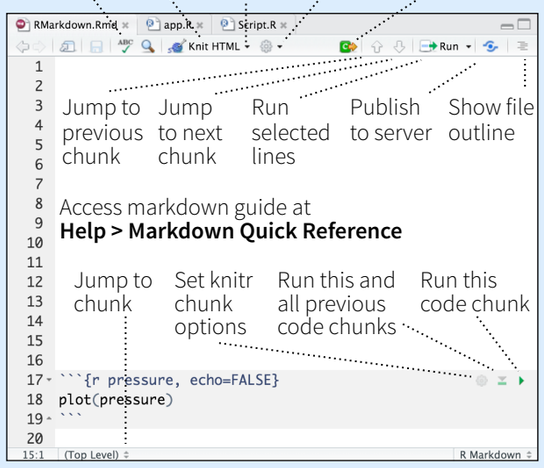
Open Source Server
for larger compute resources and remote access

Professional Server
for teams that share large compute resources, large data, and uniform environments for collaboration

Download all at **www.rstudio.com**. Each provides the same useful interface:

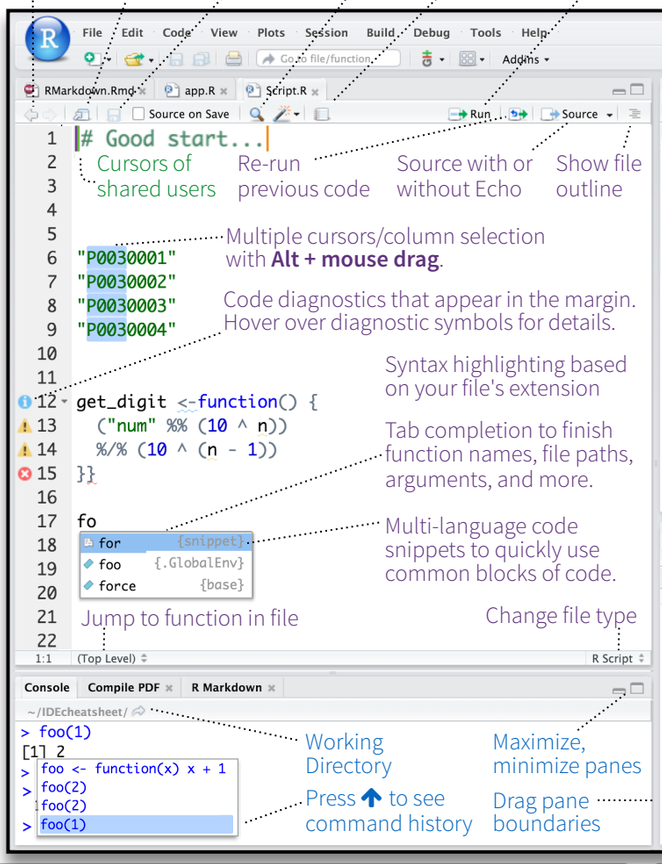
Documents and Apps

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



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

Write Code

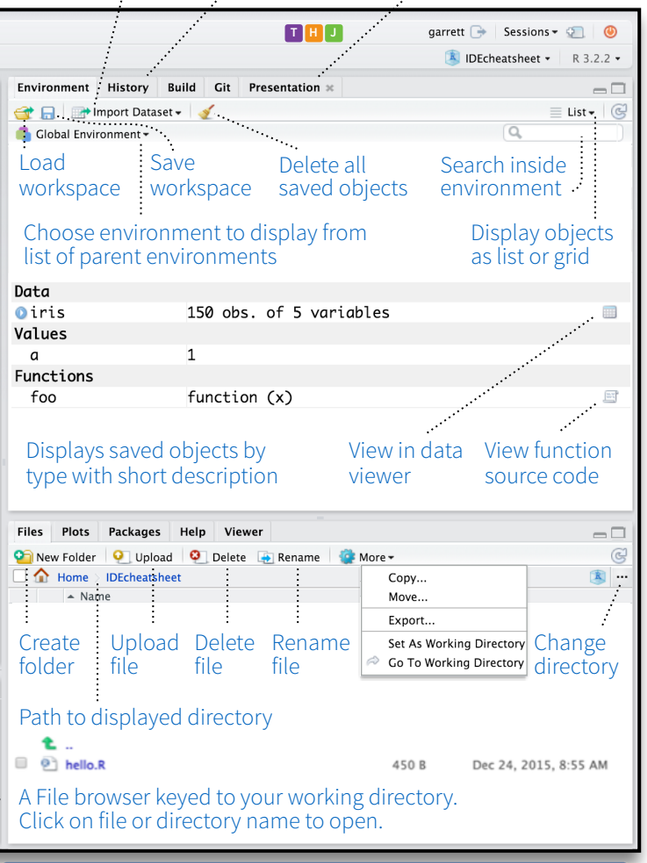


R Support

Import data file with wizard

History of past commands to run/add to source

Display .RPres slideshows
File > New File > R Presentation



RStudio Pro Features

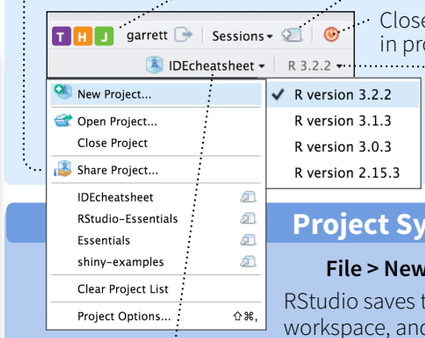
Share Project with Collaborators

Active shared collaborators

Start **new R Session** in current project

Close R Session in project

Select R Version



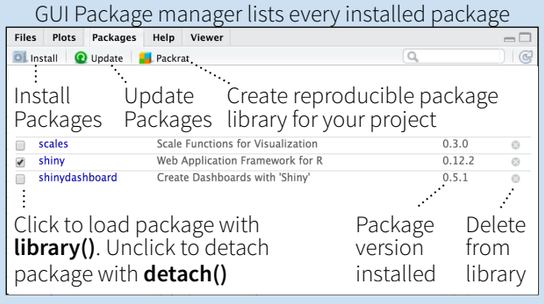
Project System

File > New Project

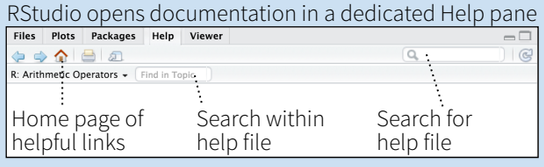
RStudio saves the call history, workspace, and working directory associated with a project. It reloads each when you re-open a project.

RStudio opens plots in a dedicated Plots pane

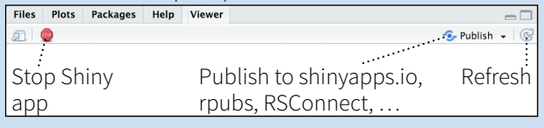
GUI Package manager lists every installed package



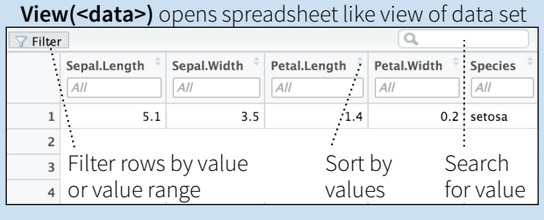
RStudio opens documentation in a dedicated Help pane



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



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

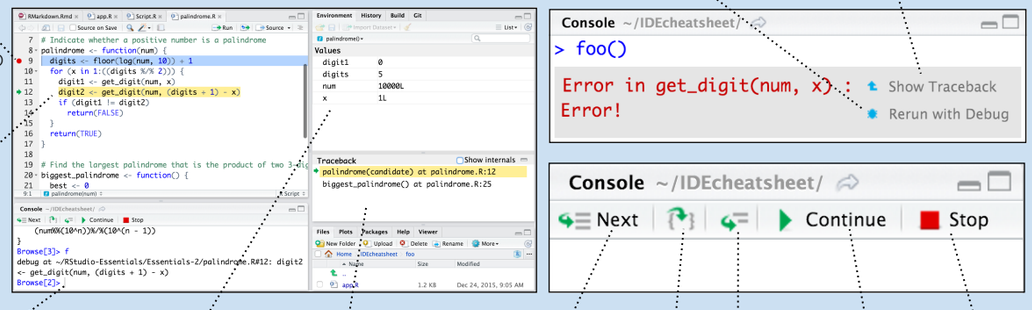


Debug Mode

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

Launch debugger mode from origin of error

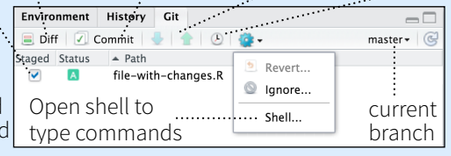
Open traceback to examine the functions that R called before the error occurred



Version Control with Git or SVN

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

Stage files:
- Added
- Deleted
- Modified
- Renamed
- Untracked

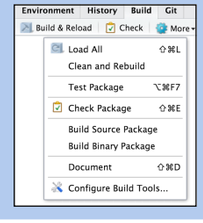


Package Writing

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



RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

More cheat sheets at **http://www.rstudio.com/resources/cheatsheets/**

Learn more at **support.rstudio.com** • RStudio IDE 0.99.832 • Updated: 01/16

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+Shift+O	Cmd+Shift+O
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+Alt+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: Ctrl+Shift	Cmd+Shift+G
Jump to Word	Ctrl+ ←/→	Option+ ←/→
Jump to Start/End	Ctrl+↑/↓	Cmd+↑/↓

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+Alt+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+Alt+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+Alt+Up
Add New Cursor Below	Ctrl+Alt+Down	Ctrl+Alt+Down
Move Active Cursor Up	Ctrl+Alt+Shift+Up	Ctrl+Alt+Shift+Up
Move Active Cursor Down	Ctrl+Alt+Shift+Down	Ctrl+Alt+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

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+Alt+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

Why RStudio Server Pro?

Do everything you would do with the open source server with a commercial license, support, and more.

- edit the same project at the same time as others
- switch easily from one version of R to a different version
- open and run multiple R sessions simultaneously
- see what you and others are doing on your server
- tune your resources to improve performance
- integrate with your authentication, authorization, and audit practices

Download a free 45 day evaluation at
www.rstudio.com/products/rstudio-server-pro/

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.

rev(x)
Return x reversed.

table(x)
See counts of values.

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	Ouput	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

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com

Types

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

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

Maths Functions

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

Variable Assignment

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

The Environment

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

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

Matrices

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

`m[2, ]` - Select a row

`m[ , 1]` - Select a column

`m[2, 3]` - Select an element

`t(m)`
Transpose

`m %*% n`
Matrix Multiplication

`solve(m, n)`
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.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

Also see the **dplyr** 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

List subsetting

`df$x`

`df[[2]]`

Understanding a data frame

`View(df)`	See the full data frame.
`head(df)`	See the first 6 rows.

Matrix subsetting

`df[ , 2]`

`df[2, ]`

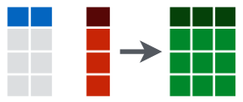
`df[2, 2]`

`nrow(df)`
Number of rows.

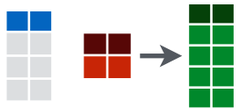
`ncol(df)`
Number of columns.

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

`cbind` - Bind columns.



`rbind` - Bind rows.



Strings

Also see the **stringr** package.

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

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

`aov`
Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.

`plot(x)`
Values of x in order.

`plot(x, y)`
Values of x against y.

`hist(x)`
Histogram of x.

Dates

See the **lubridate** package.

Data Import

with readr, tibble, and tidyr

Cheat Sheet

RStudio

R's **tidyverse** is built around **tidy data** stored in **tibbles**, an enhanced version of a data frame.

readr

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

tidyr

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)

Write functions

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

write_csv(x, path, na = "NA", append = FALSE, col_names = !append**)**
Tibble/df to comma delimited file.

write_delim(x, path, delim = " ", na = "NA", append = FALSE, col_names = !append**)**
Tibble/df to file with any delimiter.

write_excel_csv(x, path, na = "NA", append = FALSE, col_names = !append**)**
Tibble/df to a CSV for excel

write_file(x, path, append = FALSE**)**
String to file.

write_lines(x, path, na = "NA", append = FALSE**)**
String vector to file, one element per line.

write_rds(x, path, compress = c("none", "gz", "bz2", "xz"), ...**)**
Object to RDS file.

write_tsv(x, path, na = "NA", append = FALSE, col_names = !append**)**
Tibble/df to tab delimited files.

Read functions

Read tabular data to tibbles

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

read_csv()
Reads comma delimited files.
read_csv("file.csv")

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

A	B	C
1	2	3
4	5	NA

read_csv2()
Reads Semi-colon delimited files.
read_csv2("file2.csv")

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

A	B	C
1	2	3
4	5	NA

read_delim(delim, quote = "\"", escape_backslash = FALSE, escape_double = TRUE**)** Reads files with any delimiter.
read_delim("file.txt", delim = "|")

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

A	B	C
1	2	3
4	5	NA

read_fwf(col_positions**)**
Reads fixed width files.
read_fwf("file.fwf", col_positions = c(1, 3, 5))

read_tsv()
Reads tab delimited files. Also **read_table().**
read_tsv("file.tsv")

Useful arguments

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

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

A	B	C
1	2	3
4	5	NA

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

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

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

1	2	3
4	5	NA

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

A	B	C
1	2	3

Read in a subset
read_csv("file.csv",
***n_max = 1**)*

A	B	C
1	2	3
NA	NA	NA

Missing Values
read_csv("file.csv",
***na = c("4", "5", ".")**)*

Read non-tabular data

read_file(file, locale = default_locale()**)**
Read a file into a single string.

read_file_raw(file**)**
Read a file into a raw vector.

read_lines(file, skip = 0, n_max = -1L, locale = default_locale(), na = character(), progress = interactive()**)**
Read each line into its own string.

read_lines_raw(file, skip = 0, n_max = -1L, progress = interactive()**)**
Read each line into a raw vector.

read_log(file, col_names = FALSE, col_types = NULL, skip = 0, n_max = -1, progress = interactive()**)**
Apache style log files.

Parsing 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 = ""**)** and **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(**x, na = c("", "NA"), locale = default_locale()**)**
- **parse_character(**x, na = c("", "NA"), locale = default_locale()**)**
- **parse_datetime(**x, format = "", na = c("", "NA"), locale = default_locale()**)** Also **parse_date()** and **parse_time()**
- **parse_double(**x, na = c("", "NA"), locale = default_locale()**)**
- **parse_factor(**x, levels, ordered = FALSE, na = c("", "NA"), locale = default_locale()**)**
- **parse_integer(**x, na = c("", "NA"), locale = default_locale()**)**
- **parse_logical(**x, na = c("", "NA"), locale = default_locale()**)**
- **parse_number(**x, na = c("", "NA"), locale = default_locale()**)**

x$A <- parse_number(x$A)

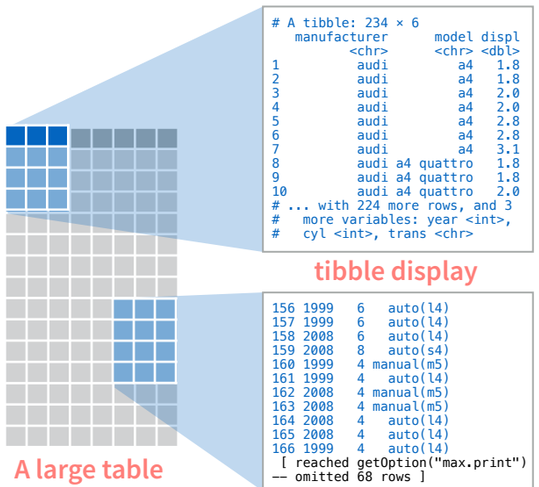
RStudio® is a trademark of RStudio, Inc. • CC BY RStudio info@rstudio.com • 844-448-1212 • rstudio.com

Learn more at **browseVignettes(package = c("readr", "tibble", "tidyr"))** • readr 1.1.0 • tibble 1.2.12 • tidyr 0.6.0 • Updated: 2017-01

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 two behaviors:

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




```
# A tibble: 234 × 6
   manufacturer      model displ
          <chr>      <chr> <dbl>
1          audi         a4   1.8
2          audi         a4   1.8
3          audi         a4   2.0
4          audi         a4   2.0
5          audi         a4   2.8
6          audi         a4   2.8
7          audi         a4   3.1
8          audi a4 quattro   1.8
9          audi a4 quattro   1.8
10         audi a4 quattro   2.0
# ... with 224 more rows, and 3
#   more variables: year <int>,
#   cyl <int>, trans <chr>
```

```
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 ]
```

- Control the default appearance with options:

  options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf**)**

- View entire data set with **View(**x, title**)** or **glimpse(**x, width = NULL, …**)**
- Revert to data frame with **as.data.frame()** (required for some older packages)

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")

Both make this tibble

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

as_tibble(x, …**)** Convert data frame to tibble.

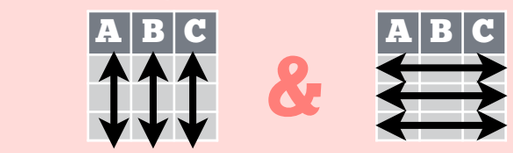
enframe(x, name = "name", value = "value"**)**
Converts named vector to a tibble with a names column and a values column.

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:

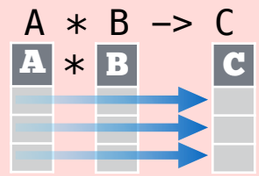


Each **variable** is in its own **column**

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. Each uses the idea of a key column: value column pair.

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.

table4a

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

→

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

key value

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 that result.

table2

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

→

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

key value

spread(table2, type, count)

Handle Missing Values

drop_na(data, ...**)**
Drop rows containing NA's in … columns.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
D	3

drop_na(x, x2)

fill(data, ..., .direction = c("down", "up")**)**
Fill in NA's in … columns with most recent non-NA values.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	1
C	1
D	3
E	3

fill(x, x2)

replace_na(data, replace = list(), ...**)**
Replace NA's by column.

x

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

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

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

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 and Combine 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.

table3

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

→

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

separate_rows(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_()**.

table3

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

→

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

separate_rows(table3, rate)

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

Collapse cells across several columns to make a single column.

table5

country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

→

country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

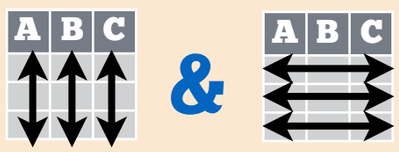
unite(table5, century, year, col = "year", sep = "")

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio info@rstudio.com • 844-448-1212 • rstudio.com

Data Transformation
with dplyr Cheat Sheet

RStudio

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).

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

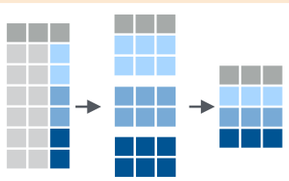
count(x, ..., wt = NULL, sort = FALSE**)**
Count number of rows in each group defined by the variables in … Also **tally()**.
count(iris, Species)

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 created 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))
```

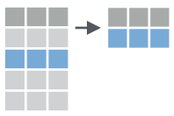
group_by(.data, ..., add = FALSE**)**
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

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

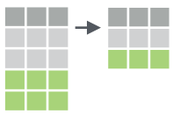
Manipulate Cases

Extract Cases

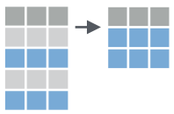
Row functions return a subset of rows as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.



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

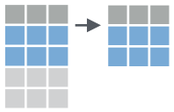


distinct(.data, ..., .keep_all = FALSE**)**
Remove rows with duplicate values. Also **distinct_()**. *distinct(iris, Species)*



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)

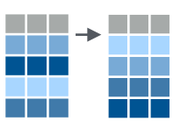
top_n(*x, n, wt***)**
Select and order top n entries (by group if grouped data). *top_n(iris, 5, Sepal.Width)*

Logical and boolean operators to use with filter()

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

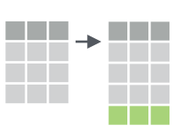
See **?base::logic** and **?Comparison** for help.

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)

Manipulate Variables

Extract Variables

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

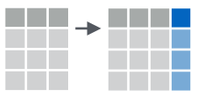
select(.data, …**)**
Extract columns by name. Also **select_if()**
select(iris, Sepal.Length, Species)

Use these helpers with select(),
e.g. *select(iris, starts_with("Sepal"))*

contains(match**)**	**num_range(**prefix, range**)**	**:**, e.g. mpg:cyl
ends_with(match**)**	**one_of(**…**)**	**-**, e.g, -Species
matches(match**)**	**starts_with(**match**)**	

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).



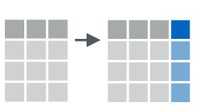
mutate(.data, …**)**
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

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

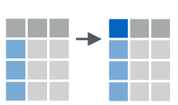
mutate_all(.tbl, .funs, ...**)**
Apply funs to every column. Use with **funs()**. *mutate_all(faithful, funs(log(.), log2(.)))*

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)

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

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

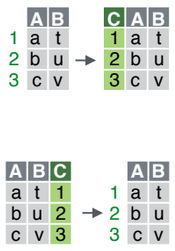
quantile() - nth quantile
min() - minimum value
max() - maximum value

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.



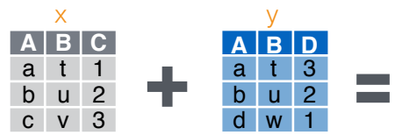
rownames_to_column()
Move row names into col.
a <- rownames_to_column(iris, var = "C")

column_to_rownames()
Move col in row names.
column_to_rownames(a, var = "C")

Also **has_rownames()**, **remove_rownames()**

Combine Tables

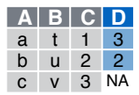
Combine Variables



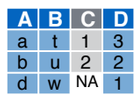
Use **bind_cols()** to paste tables beside each other as they are.

bind_cols(…**)**
Returns tables placed side by side as a single table.
BE SURE THAT ROWS ALIGN.

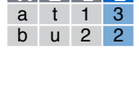
Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.



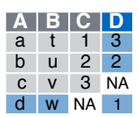
left_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…**)**
Join matching values from y to x.



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

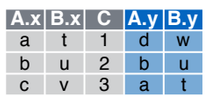


inner_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…**)**
Join data. Retain only rows with matches.

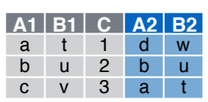


full_join(x, y, by = NULL, copy=FALSE, suffix=c(".x",".y"),…**)**
Join data. Retain all values, all rows.

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

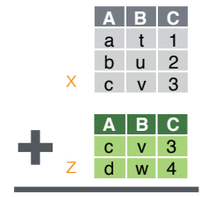


Use a named vector, **by = c("col1" = "col2")**, to match on columns with different names in each data set.
left_join(x, y, by = c("C" = "D"))



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)

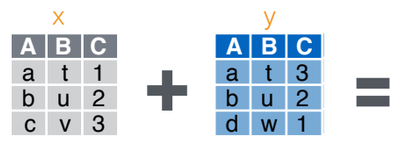
intersect(x, y, …**)**
Rows that appear in both x and z.

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.

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

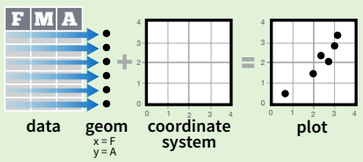
Data Visualization with ggplot2

Cheat Sheet

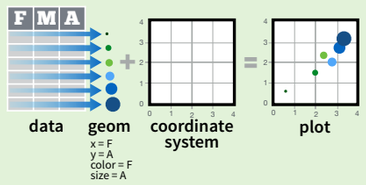
RStudio

Basics

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



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



Complete the template below to build a graph.

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

Required

Not required, sensible defaults supplied

ggplot(data = mpg, **aes(**x = cty, y = hwy**))**

Begins a plot that you finish by adding layers to. Add one geom function per layer.

aesthetic mappings · data · geom

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

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

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

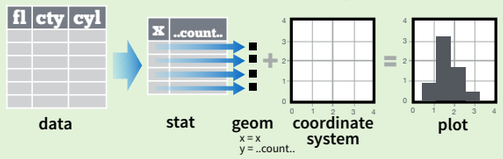
l + geom_raster(aes(fill = z), hjust=0.5, vjust=0.5, interpolate=FALSE**)**
x, y, alpha, fill

l + geom_tile(aes(fill = z)**)**
x, y, alpha, color, fill, linetype, size, width

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

Stats - An alternative way to build a layer

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



Visualize a stat by changing the default stat of a geom function, **geom_bar(stat="count")** or by using a stat function, **stat_count(geom="bar")**, which calls a default geom to make a layer (equivalent to a geom function).

Use **..name..** syntax to map stat variables to aesthetics.

geom to use | stat function | geom mappings

i + **stat_density2d**(aes(fill = ..level..), geom = "polygon") — variable created by stat

1D distributions

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..

2D distributions

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..

3 Variables

Comparisons

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..

Functions

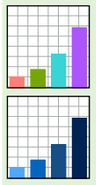
e + **stat_ecdf**(n = 40) x, y | ..x.., ..y..
e + **stat_quantile**(quantiles = c(0.1, 0.9),
formula = y ~ log(x), method = "rq") x, y | ..quantile..
e + **stat_smooth**(method = "lm", formula = y ~ x,
se=T, level=0.95) x, y | ..se.., ..x.., ..y.., ..ymin.., ..ymax..

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**()

General Purpose

Scales

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



(n <- d + geom_bar(aes(fill = fl)))

scale_ | aesthetic to adjust | prepackaged scale to use | scale specific arguments

n + **scale_fill_manual**(
values = c("skyblue", "royalblue", "blue", "navy"),
limits = c("d", "e", "p", "r"), breaks =c("d", "e", "p", "r"),
name = "fuel", labels = c("D", "E", "P", "R"))

range of values to include in mapping | title to use in legend/axis | labels to use in legend/axis | breaks to use in legend/axis

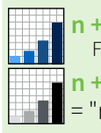
General Purpose scales
Use with most aesthetics

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 and Y location scales
Use with x or y aesthetics (x shown here)

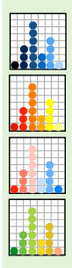
scale_x_log10() - Plot x on log10 scale
scale_x_reverse() - Reverse direction of x axis
scale_x_sqrt() - Plot x on square root scale

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)

Maps to radius of circle, or area

Coordinate Systems

r <- d + geom_bar()

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

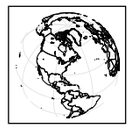
r + **coord_fixed**(ratio = 1/2)
ratio, xlim, ylim
Cartesian coordinates with fixed aspect ratio between x and y units

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.

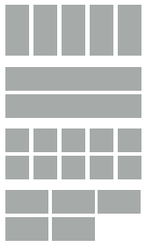
π + **coord_quickmap**()
π + **coord_map**(projection = "ortho", orientation=c(41, -74, 0))
projection, orientation, xlim, ylim
Map projections from the mapproj package (mercator (default), azequalarea, lagrange, etc.)



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)))

α^c	α^d	α^e	α^p	α^r

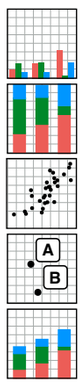
t + facet_grid(. ~ fl, **labeller = label_parsed**)

c	d	e	p	r

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)**)

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

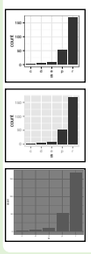
t + **annotate**(geom = "text", x = 8, y = 9, label = "A")

geom to place | manual values for geom's aesthetics

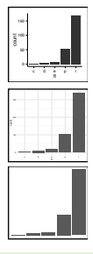
Legends

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.

Themes



r + **theme_bw**()
White background with grid lines
r + **theme_gray**()
Grey background (default theme)
r + **theme_dark**()
dark for contrast



r + **theme_classic**()
r + theme_light()
r + theme_linedraw()
r + **theme_minimal**()
Minimal themes
r + **theme_void**()
Empty theme

Zooming

Without clipping (preferred)
t + **coord_cartesian**(
xlim = c(0, 100), ylim = c(10, 20))

With clipping (removes unseen data points)

t + **xlim**(0, 100) + **ylim**(10, 20)
t + **scale_x_continuous**(limits = c(0, 100)) + **scale_y_continuous**(limits = c(0, 100))

RStudio® is a trademark of RStudio, Inc. • CC BY RStudio • info@rstudio.com • 844-448-1212 • rstudio.com

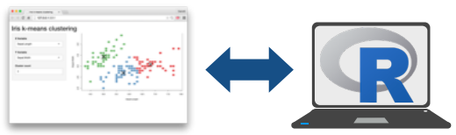
Interactive Web Apps with shiny Cheat Sheet

learn more at shiny.rstudio.com

RStudio

Basics

A **Shiny** app is a web page (**UI**) connected to a computer running a live R session (**Server**)



Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

App template

Begin writing a new app with this template. Preview the app by running the code at the R command line.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines `ui` and `server` into a functioning app. Wrap with **runApp()** if calling from a sourced script or inside a function.

Share your app

The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at **http://shinyapps.io**
2. Click the **Publish** icon in the RStudio IDE (>=0.99) or run:
   rsconnect::deployApp("<path to directory>"**)**

Build or purchase your own Shiny Server
at www.rstudio.com/products/shiny-server/

Building an App - Complete the template by adding arguments to fluidPage() and a body to the server function.

Add inputs to the UI with ***Input()** functions

Add outputs with ***Output()** functions

Tell server how to render outputs with R in the server function. To do this:

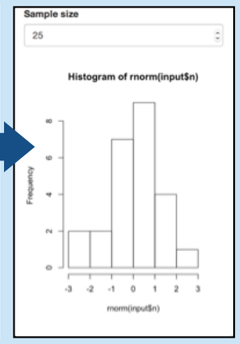
1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

shinyApp(ui = ui, server = server)
```



Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

shinyApp(ui = ui, server = server)
```

```
# ui.R
fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
```

```
# server.R
function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
```

ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

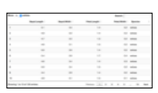
No need to call **shinyApp()**.

Save each app as a directory that contains an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.

- app-name/ — The directory name is the name of the app
  - app.R
  - global.R — (optional) defines objects available to both ui.R and server.R
  - DESCRIPTION — (optional) used in showcase mode
  - README — (optional) data, scripts, etc.
  - <other files>
  - www — (optional) directory of files to share with web browsers (images, CSS, .js, etc.) Must be named "**www**"

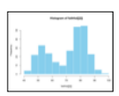
Launch apps with `runApp(<path to directory>)`

Outputs - render*() and *Output() functions work together to add R output to the UI



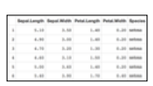
DT::**renderDataTable**(expr, options, callback, escape, env, quoted) — works with — **dataTableOutput**(outputId, icon, …)

renderImage(expr, env, quoted, deleteFile) — **imageOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)



renderPlot(expr, width, height, res, …, env, quoted, func) — **plotOutput**(outputId, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline)

renderPrint(expr, env, quoted, func, width) — **verbatimTextOutput**(outputId)



renderTable(expr,…, env, quoted, func) — **tableOutput**(outputId)

renderText(expr, env, quoted, func) — **textOutput**(outputId, container, inline)

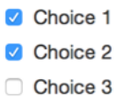
renderUI(expr, env, quoted, func) — **uiOutput**(outputId, inline, container, …) & **htmlOutput**(outputId, inline, container, …)

Inputs - collect values from the user

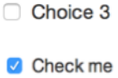
Access the current value of an input object with **input $<inputId>**. Input values are **reactive**.

actionButton(inputId, label, icon, …)

actionLink(inputId, label, icon, …)



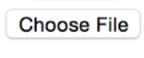
checkboxGroupInput(inputId, label, choices, selected, inline)



checkboxInput(inputId, label, value)

dateInput(inputId, label, value, min, max, format, startview, weekstart, language)

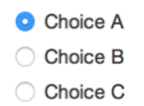
dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)



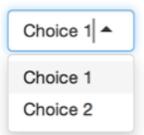
fileInput(inputId, label, multiple, accept)

numericInput(inputId, label, value, min, max, step)

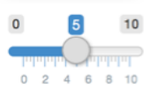
passwordInput(inputId, label, value)



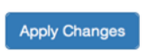
radioButtons(inputId, label, choices, selected, inline)



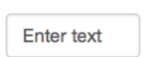
selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also **selectizeInput()**)



sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)



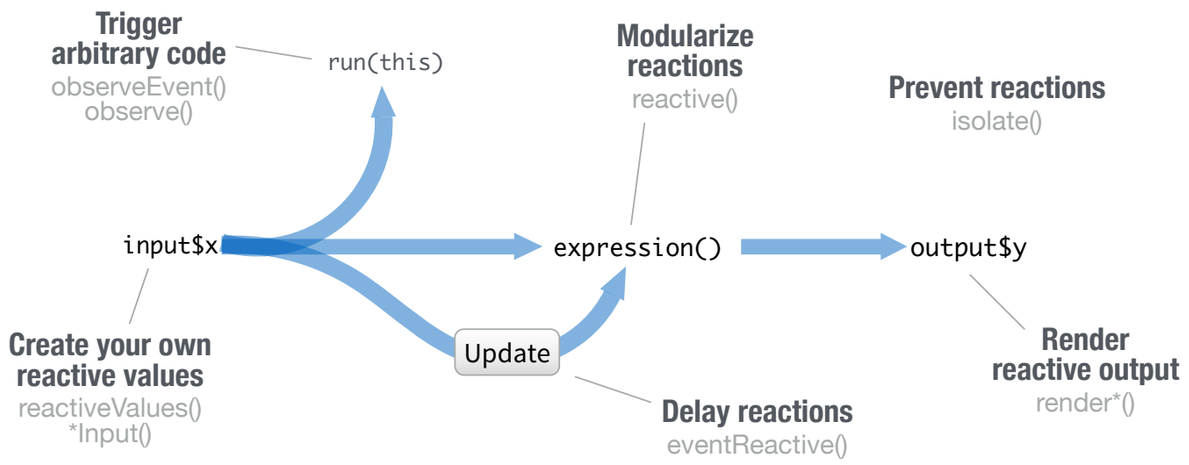
submitButton(text, icon)
(Prevents reactions across entire app)



textInput(inputId, label, value)

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error `Operation not allowed without an active reactive context.`



Create your own reactive values

```
# example snippets


ui <- fluidPage(
 textInput("a","","A")
)


server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}
```

***Input() functions**
(see front page)

reactiveValues(...)

Each input function creates a reactive value stored as **input$<inputId>**

reactiveValues() creates a list of reactive values whose values you can set.

Render reactive output

```
library(shiny)

ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   input$a
  })
}

shinyApp(ui, server)
```

render*() functions
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output$<outputId>**

Prevent reactions

```
library(shiny)

ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

server <-
function(input,output){
 output$b <-
  renderText({
   isolate({input$a})
  })
}

shinyApp(ui, server)
```

isolate(expr)

Runs a code block. Returns a **non-reactive** copy of the results.

Trigger arbitrary code

```
library(shiny)

ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go")
)

server <-
function(input,output){
 observeEvent(input$go,{
   print(input$a)
  })
}

shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, labe, suspended, priority, domain, autoDestroy, ignoreNULL)

Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

Modularize reactions

```
library(shiny)

ui <- fluidPage(
 textInput("a","","A"),
 textInput("z","","Z"),
 textOutput("b")
)

server <-
function(input,output){
 re <- reactive({
  paste(input$a,input
$z)})
 output$b <- renderText({
   re()
  })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain)

Creates a **reactive expression** that

- **caches** its value to reduce computation
- can be called by other code
- notifies its dependencies when it ha been invalidated

Call the expression with function syntax, e.g. `re()`

Delay reactions

```
library(shiny)

ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go"),
 textOutput("b")
)

server <-
function(input,output){
 re <- eventReactive(
  input$go,{input$a})
 output$b <- renderText({
   re()
  })
}

shinyApp(ui, server)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, label, domain, ignoreNULL)

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

UI

An app's UI is an HTML document. Use Shiny's functions to assemble this HTML with R.

```
fluidPage(
  textInput("a","")
)
## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##     <label for="a"></label>
##     <input id="a" type="text"
##        class="form-control" value=""/>
##   </div>
## </div>
```

Returns HTML

Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. **tags$a()**. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

tags$a	tags$data	tags$h6	tags$nav	tags$span
tags$abbr	tags$datalist	tags$head	tags$noscript	tags$strong
tags$address	tags$dd	tags$header	tags$object	tags$style
tags$area	tags$del	tags$hgroup	tags$ol	tags$sub
tags$article	tags$details	tags$hr	tags$optgroup	tags$summary
tags$aside	tags$dfn	tags$HTML	tags$option	tags$sup
tags$audio	tags$div	tags$i	tags$output	tags$table
tags$b	tags$dl	tags$iframe	tags$p	tags$tbody
tags$base	tags$dt	tags$img	tags$param	tags$td
tags$bdi	tags$em	tags$input	tags$pre	tags$textarea
tags$bdo	tags$embed	tags$ins	tags$progress	tags$tfoot
tags$blockquote	tags$eventsource	tags$kbd	tags$q	tags$th
tags$body	tags$fieldset	tags$keygen	tags$ruby	tags$thead
tags$br	tags$figcaption	tags$label	tags$rp	tags$time
tags$button	tags$figure	tags$legend	tags$rt	tags$title
tags$canvas	tags$footer	tags$li	tags$s	tags$tr
tags$caption	tags$form	tags$link	tags$samp	tags$track
tags$cite	tags$h1	tags$mark	tags$script	tags$u
tags$code	tags$h2	tags$map	tags$section	tags$ul
tags$col	tags$h3	tags$menu	tags$select	tags$var
tags$colgroup	tags$h4	tags$meta	tags$small	tags$video
tags$command	tags$h5	tags$meter	tags$source	tags$wbr

The most common tags have wrapper functions. You do not need to prefix their names with **tags$**

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="", "link"),
  HTML("<p>Raw html</p>")
)
```

To include a CSS file, use **includeCSS()**, or
1. Place the file in the **www** subdirectory
2. Link to it with

```
tags$head(tags$link(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```

To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with

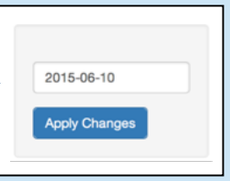
```
tags$head(tags$script(src = "<file name>"))
```

IMAGES To include an image
1. Place the file in the **www** subdirectory
2. Link to it with `img(src="<file name>")`

Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

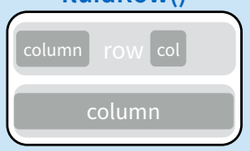
```
wellPanel(
  dateInput("a", ""),
  submitButton()
)
```



absolutePanel()	inputPanel()	tabPanel()
conditionalPanel()	mainPanel()	tabsetPanel()
fixedPanel()	navlistPanel()	titlePanel()
headerPanel()	sidebarPanel()	wellPanel()

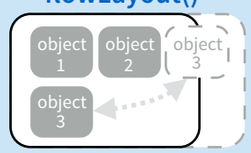
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

fluidRow()




```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()

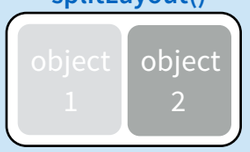



```
ui <- fluidPage(
  flowLayout( # object 1,
              # object 2,
              # object 3
  )
)
```

sidebarLayout()


```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()



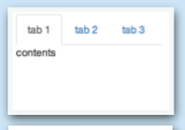

```
ui <- fluidPage(
  splitLayout( # object 1,
               # object 2
  )
)
```

verticalLayout()


```
ui <- fluidPage(
  verticalLayout( # object 1,
                  # object 2,
                  # object 3
  )
)
```

Layer tabPanels on top of each other, and navigate between them, with:

```
ui <- fluidPage( tabsetPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")))
```




```
ui <- fluidPage( navlistPanel(
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents")))
```


```
ui <- navbarPage(title = "Page",
    tabPanel("tab 1", "contents"),
    tabPanel("tab 2", "contents"),
    tabPanel("tab 3", "contents"))
```