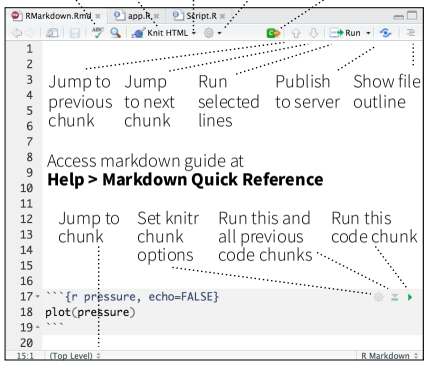
RStudio IDE : : **CHEAT SHEET**

Documents and Apps

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

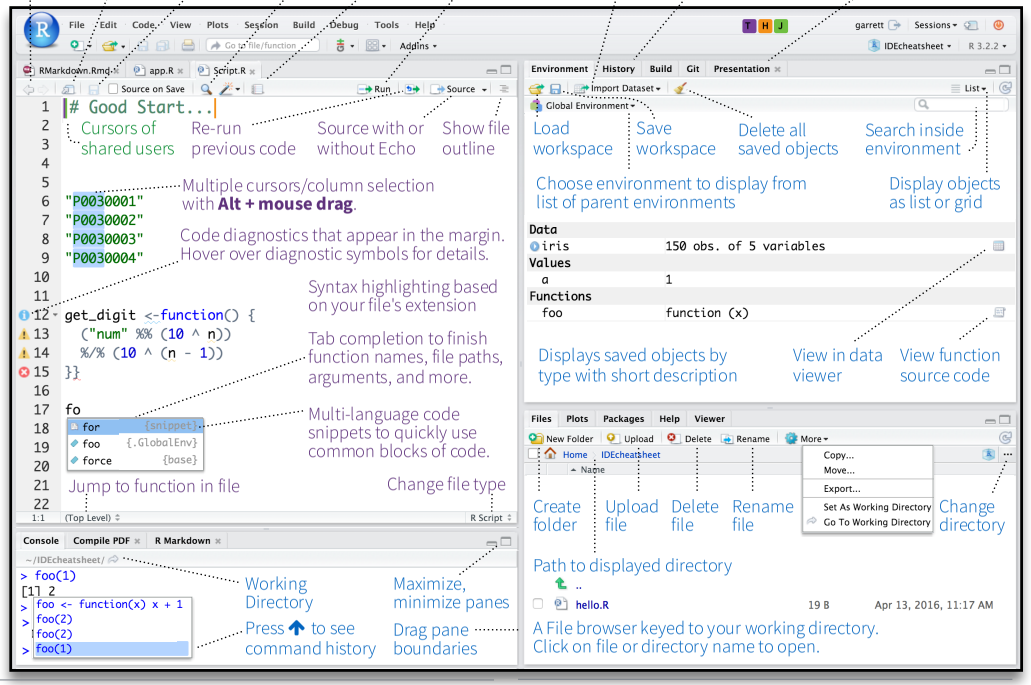
Check spelling | Render output | Choose output format | Choose output location | Insert code chunk



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

Write Code

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

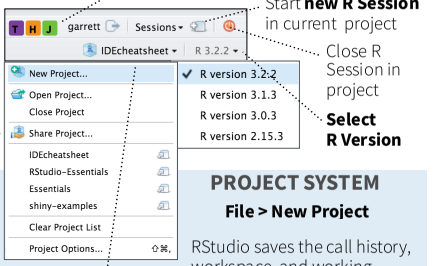


R Support

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

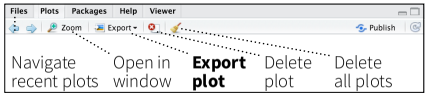


PROJECT SYSTEM

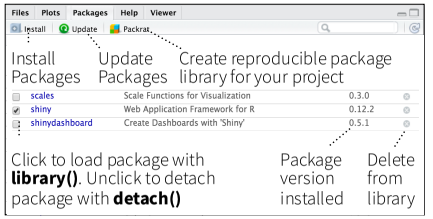
File > New Project

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

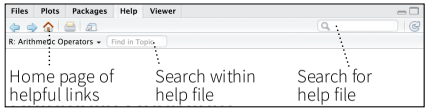
RStudio opens plots in a dedicated Plots pane



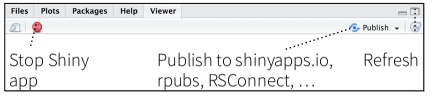
GUI Package manager lists every installed package



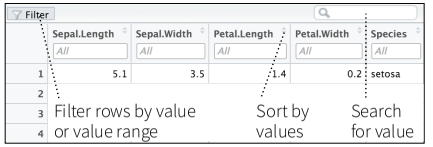
RStudio opens documentation in a dedicated Help pane



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



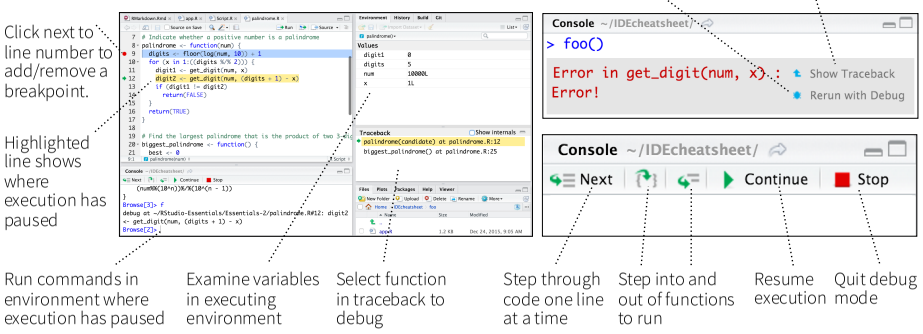
View(<data>) opens spreadsheet like view of data set



Debug Mode

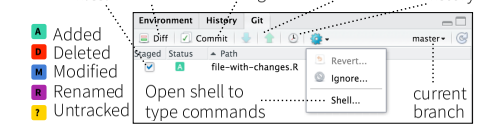
Open with **debug(), browse(),** or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

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**

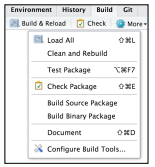


Package Writing

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

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**

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

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, ]` 

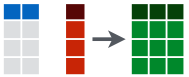
`df[2, 2]` 

`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.

Advanced R
Cheat Sheet

Created by: Arianne Colton and Sean Chen

Environment Basics

Environment – **Data structure** (with two components below) that powers lexical scoping

Create environment: env1<-new.env()

1. **Named list** ("Bag of names") – each name points to an object stored elsewhere in memory.

   If an object has no names pointing to it, it gets automatically deleted by the garbage collector.

   - Access with: **ls('env1')**
2. **Parent environment** – used to implement lexical scoping. If a name is not found in an environment, then R will look in its parent (and so on).
   - Access with: **parent.env('env1')**

Four special environments

1. **Empty environment** – ultimate ancestor of all environments
   - Parent: none
   - Access with: **emptyenv()**
2. **Base environment** - environment of the base package
   - Parent: empty environment
   - Access with: **baseenv()**
3. **Global environment** – the interactive workspace that you normally work in
   - Parent: environment of last attached package
   - Access with: **globalenv()**
4. **Current environment** – environment that R is currently working in (may be any of the above and others)
   - Parent: empty environment
   - Access with: **environment()**

Environments

Search Path

Search path – mechanism to look up objects, particularly functions.

- Access with : **search() –** lists all parents of the global environment (see Figure 1)
- Access any environment on the search path: **as.environment('package:base')**

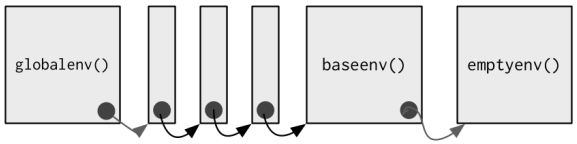


Figure 1 – The Search Path

- Mechanism : always start the search from global environment, then inside the latest attached package environment.
  - New package loading with **library()/require()** : new package is attached right after global environment. (See Figure 2)
  - Name conflict in two different package : functions with the same name, latest package function will get called.

search() :

'.GlobalEnv' ... 'Autoloads' 'package:base'

library(reshape2); search()

'.GlobalEnv' 'package:reshape2' ... 'Autoloads' 'package:base'

NOTE: Autoloads : special environment used for saving memory by only loading package objects (like big datasets) when needed

Figure 2 – Package Attachment

Binding Names to Values

Assignment – act of binding (or rebinding) a name to a value in an environment.

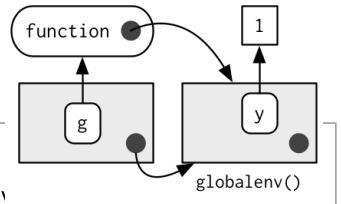
1. **<-** (Regular assignment arrow) – always creates a variable in the current environment
2. **<<-** (Deep assignment arrow) - modifies an existing variable found by walking up the parent environments

Warning: If <<- doesn't find an existing variable, it will create one in the global environment.

Function Environments

1. **Enclosing environment -** an environment where the function is created. It determines how function finds value.
   - Enclosing environment never changes, even if the function is moved to a different environment.
   - Access with: **environment('func1')**
2. **Binding environment -** all environments that the function has a binding to. It determines how we find the function.
   - Access with: **pryr::where('func1')**

   Example (for enclosing and binding environment):




```
y <- 1
e <- new.env()
e$g <- function(x) x + y
```

- function g enclosing environment is the global environment,
- the binding environment is "e".

3. **Execution environment -** new created environments to host a function call execution.
   - Two parents :

     I. Enclosing environment of the function

     II. Calling environment of the function
   - Execution environment is thrown away once the function has completed.
4. **Calling environment** - environments where the function was called.
   - Access with: **parent.frame('func1')**
   - Dynamic scoping :
     - About : look up variables in the calling environment rather than in the enclosing environment
     - Usage : most useful for developing functions that aid interactive data analysis

Updated: 2/16

Data Structures

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Note: R has no 0-dimensional or scalar types. Individual numbers or strings, are actually vectors of length one, NOT scalars.

Human readable description of any R data structure :

```
str(variable)
```

Every **Object** has a mode and a class

1. **Mode**: represents how an object is stored in memory
   - 'type' of the object from R's point of view
   - Access with: **typeof()**
2. **Class**: represents the object's abstract type
   - 'type' of the object from R's object-oriented programming point of view
   - Access with: **class()**

	typeof()	class()
strings or vector of strings	character	character
numbers or vector of numbers	numeric	numeric
list	list	list
data.frame	list	data.frame

Factors

1. Factors are built on top of integer vectors using two attributes :

```
class(x) -> 'factor'
levels(x)  # defines the set of allowed values
```

2. Useful when you know the possible values a variable may take, even if you don't see all values in a given dataset.

Warning on Factor Usage:

1. Factors look and often behave like character vectors, they are actually integers. Be careful when treating them like strings.
2. Most data loading functions automatically convert character vectors to factors. (Use argument stringAsFactors = FALSE to suppress this behavior)

Object Oriented (OO) Field Guide

Object Oriented Systems

R has three object oriented systems :

1. **S3** is a very casual system. It has no formal definition of classes. It implements generic function OO.
   - **Generic-function OO** - a special type of function called a generic function decides which method to call.

Example:	drawRect(canvas, 'blue')
Language:	R

   - **Message-passing OO** - messages (methods) are sent to objects and the object determines which function to call.

Example:	canvas.drawRect('blue')
Language:	Java, C++, and C#

2. **S4** works similarly to S3, but is more formal. Two major differences to S3 :
   - **Formal class definitions** - describe the representation and inheritance for each class, and has special helper functions for defining generics and methods.
   - **Multiple dispatch** - generic functions can pick methods based on the class of any number of arguments, not just one.
3. **Reference classes** are very different from S3 and S4:
   - **Implements message-passing OO** - methods belong to classes, not functions.
   - **Notation** - $ is used to separate objects and methods, so method calls look like **canvas$drawRect('blue')**.

S3

1. **About S3 :**
   - R's first and simplest OO system
   - Only OO system used in the base and stats package
   - Methods belong to functions, not to objects or classes.
2. **Notation :**
   - **generic.class()**

mean.Date()	Date method for the generic - mean()

3. **Useful 'Generic' Operations**
   - Get all methods that belong to the 'mean' generic:
     - **Methods('mean')**
   - List all generics that have a method for the 'Date' class :
     - **methods(class = 'Date')**
4. **S3 objects** are usually built on top of lists, or atomic vectors with attributes.
   - Factor and data frame are S3 class
   - Useful operations:

Check if object is an S3 object	is.object(x) & !isS4(x) or pryr::otype()
Check if object inherits from a specific class	inherits(x, 'classname')
Determine class of any object	**class(x)**

Base Type (C Structure)

R base types - the internal C-level types that underlie the above OO systems.

- **Includes :** atomic vectors, list, functions, environments, etc.
- **Useful operation :** Determine if an object is a base type (Not S3, S4 or RC) **is.object(x)** returns FALSE
- **Internal representation :** C structure (or struct) that includes :
  - Contents of the object
  - Memory Management Information
  - Type
    - Access with: **typeof()**

Functions

Function Basics

Functions – objects in their own right

All R functions have three parts:

body()	code inside the function
formals()	list of arguments which controls how you can call the function
environment()	"map" of the location of the function's variables (see "Enclosing Environment")

Every operation is a function call

- +, for, if, [, $, {
- x + y is the same as `+`(x, y)

Note: the backtick (`), lets you refer to functions or variables that have otherwise reserved or illegal names.

Lexical Scoping

What is Lexical Scoping?

- Looks up value of a symbol. (see "Enclosing Environment")
- **findGlobals() -** lists all the external dependencies of a function

```
f <- function() x + 1
codetools::findGlobals(f)
> '+' 'x'
```

```
environment(f) <- emptyenv()
f()
# error in f(): could not find function "+"
```

- R relies on lexical scoping to find everything, even the + operator.

Function Arguments

Arguments – passed by reference and copied on modify

1. Arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position.
2. Check if an argument was supplied : **missing()**

```
i <- function(a, b) {
  missing(a) -> # return true or false
}
```

3. Lazy evaluation – since x is not used **stop("This is an error!")** never get evaluated.

```
f <- function(x) {
  10
}
f(stop('This is an error!')) -> 10
```

4. Force evaluation

```
f <- function(x) {
  force(x)
  10
}
```

5. Default arguments evaluation

```
f <- function(x = ls()) {
  a <- 1
  x
}
```

f() -> 'a' 'x'	ls() evaluated inside f
f(ls())	ls() evaluated in global environment

Return Values

- **Last expression evaluated or explicit return().**
  Only use explicit return() when returning early.
- **Return ONLY single object.**
  Workaround is to return a list containing any number of objects.
- **Invisible return object value** - not printed out by default when you call the function.

```
f1 <- function() invisible(1)
```

Primitive Functions

What are Primitive Functions?

1. Call C code directly with **.Primitive()** and contain no R code

```
print(sum) :
> function (..., na.rm = FALSE)  .Primitive('sum')
```

2. **formals()**, **body()**, and **environment()** are all NULL
3. Only found in base package
4. More efficient since they operate at a low level

Influx Functions

What are Influx Functions?

1. Function name comes in between its arguments, like + or –
2. All user-created infix functions must start and end with %.

```
`%+%` <- function(a, b) paste0(a, b)
'new' %+% 'string'
```

3. Useful way of providing a default value in case the output of another function is NULL:

```
`%||%` <- function(a, b) if (!is.null(a)) a else b
function_that_might_return_null() %||% default value
```

Replacement Functions

What are Replacement Functions?

1. Act like they modify their arguments in place, and have the special name xxx <-
2. Actually create a modified copy. Can use **pryr::address()** to find the memory address of the underlying object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:10
second(x) <- 5L
```


Updated: 2/16

Subsetting

Subsetting returns a copy of the original data, NOT copy-on modified

Simplifying vs. Preserving Subsetting

1. **Simplifying subsetting**
   - Returns the **simplest** possible data structure that can represent the output
2. **Preserving subsetting**
   - Keeps the structure of the output the **same** as the input.
   - When you use drop = FALSE, it's preserving

	Simplifying*	Preserving
Vector	x[[1]]	x[1]
List	x[[1]]	x[1]
Factor	x[1:4, drop = T]	x[1:4]
Array	x[1,] or x[, 1]	x[1, , drop = F] or x[, 1, drop = F]
Data frame	x[, 1] or x[[1]]	x[, 1, drop = F] or x[1]

Simplifying behavior varies slightly between different data types:

1. **Atomic Vector**
   - x[[1]] is the same as x[1]
2. **List**
   - [ ] always returns a list
   - Use [[ ]] to get list contents, this returns a single value piece out of a list
3. **Factor**
   - Drops any unused levels but it remains a factor class
4. **Matrix or Array**
   - If any of the dimensions has length 1, that dimension is dropped
5. **Data Frame**
   - If output is a single column, it returns a vector instead of a data frame

Data Frame Subsetting

Data Frame – possesses the **characteristics of both lists and matrices**. If you subset with a single vector, they behave like lists; if you subset with two vectors, they behave like matrices

1. **Subset with a single vector** : Behave like lists

```
df1[c('col1', 'col2')]
```

2. **Subset with two vectors** : Behave like matrices

```
df1[, c('col1', 'col2')]
```

The results are the same in the above examples, however, results are different if subsetting with only one column. (see below)

1. **Behave like matrices**

```
str(df1[, 'col1']) -> int [1:3]
```

   - Result: the result is a vector

2. **Behave like lists**

```
str(df1['col1']) -> 'data.frame'
```

   - Result: the result remains a data frame of 1 column

$ Subsetting Operator

1. **About Subsetting Operator**
   - Useful shorthand for [[combined with character subsetting

```
x$y is equivalent to x[['y', exact = FALSE]]
```

2. **Difference vs. [[**
   - $ does partial matching, [[does not

```
x <- list(abc = 1)
x$a -> 1       # since "exact = FALSE"
x[['a']] ->    # would be an error
```

3. **Common mistake with $**
   - Using it when you have the name of a column stored in a variable

```
var <- 'cyl'
x$var

# doesn't work, translated to x[['var']]
# Instead use x[[var]]
```

Examples

1. **Lookup tables** (character subsetting)

```
x <- c('m', 'f', 'u', 'f', 'f', 'm', 'm')
lookup <- c(m = 'Male', f = 'Female', u = NA)
lookup[x]
> m   f   u   f   f   m   m
> 'Male'  'Female'  NA  'Female'  'Female'  'Male'  'Male'
unname(lookup[x])
> 'Male'  'Female'  NA  'Female'  'Female'  'Male'  'Male'
```

2. **Matching and merging by hand** (integer subsetting)

Lookup table which has multiple columns of information:

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(
  grade = 3:1,
  desc = c('Excellent', 'Good', 'Poor'),
  fail = c(F, F, T)
)
```

First Method

```
id <- match(grades, info$grade)
info[id, ]
```

Second Method

```
rownames(info) <- info$grade
info[as.character(grades), ]
```

3. **Expanding aggregated counts** (integer subsetting)
   - **Problem**: a data frame where identical rows have been collapsed into one and a count column has been added
   - **Solution**: rep() and integer subsetting make it easy to uncollapse the data by subsetting with a repeated row index: rep(x, y) rep replicates the values in x, y times.

```
df1$countCol is c(3, 5, 1)
rep(1:nrow(df1), df1$countCol)
> 1 1 1 2 2 2 2 2 3
```

4. **Removing columns from data frames** (character subsetting)

There are two ways to remove columns from a data frame:

Set individual columns to NULL	df1$col3 <- NULL
Subset to return only columns you want	df1[c('col1', 'col2')]

5. **Selecting rows based on a condition** (logical subsetting)
   - This is the most commonly used technique for extracting rows out of a data frame.

```
df1[df1$col1 == 5 & df1$col2 == 4, ]
```


Updated: 2/16

Subsetting continued

Boolean Algebra vs. Sets
(Logical and Integer Subsetting)

1. **Using integer subsetting** is more effective when:
   - You want to find the first (or last) TRUE.
   - You have very few TRUEs and very many FALSEs; a set representation may be faster and require less storage.
2. **which()** - conversion from boolean representation to integer representation

```
which(c(T, F, T F)) -> 1 3
```

- Integer representation length : is always <= boolean representation length
- Common mistakes :
  I. Use **x[which(y)]** instead of **x[y]**
  II. **x[-which(y)]** is not equivalent to **x[!y]**

Recommendation:
Avoid switching from logical to integer subsetting unless you want, for example, the first or last TRUE value

Subsetting with Assignment

1. All subsetting operators can be combined with assignment to modify selected values of the input vector.

```
df1$col1[df1$col1 < 8] <- 0
```

2. Subsetting with nothing in conjunction with assignment :
   - Why : Preserve original object class and structure

```
df1[] <- lapply(df1, as.integer)
```

Debugging, Condition Handling and Defensive Programming

Debugging Methods

1. **traceback()** or **RStudio's error inspector**
   - Lists the sequence of calls that lead to the error
2. **browser()** or **RStudio's breakpoints tool**
   - Opens an interactive debug session at an arbitrary location in the code
3. **options(error = browser)** or **RStudio's "Rerun with Debug" tool**
   - Opens an interactive debug session where the error occurred
   - Error Options:

   options(error = recover)
   - Difference vs. 'browser': can enter environment of any of the calls in the stack

   options(error = dump_and_quit)
   - Equivalent to 'recover' for non-interactive mode
   - Creates **last.dump.rda** in the current working directory

   In batch R process :

```
dump_and_quit <- function() {
    # Save debugging info to file last.dump.rda
    dump.frames(to.file = TRUE)
    # Quit R with error status
    q(status = 1)
}
options(error = dump_and_quit)
```

In a later interactive session :

```
load("last.dump.rda")
debugger()
```

Condition Handling of Expected Errors

1. **Communicating potential problems to users:**
   I. **stop()**
      - Action : raise fatal error and force all execution to terminate
      - Example usage : when there is no way for a function to continue
   II. **warning()**
      - Action : generate warnings to display potential problems
      - Example usage : when some of elements of a vectorized input are invalid
   III. **message()**
      - Action : generate messages to give informative output
      - Example usage : when you would like to print the steps of a program execution
2. **Handling conditions programmatically**:
   I. **try()**
      - Action : gives you the ability to continue execution even when an error occurs
   II. **tryCatch()**
      - Action : lets you specify handler functions that control what happens when a condition is signaled

```
result = tryCatch(code,
    error = function(c) "error",
    warning = function(c) "warning",
    message = function(c) "message"
)
```

Use conditionMessage(c) or c$message to extract the message associated with the original error.

Defensive Programming

Basic principle : "fail fast", to raise an error as soon as something goes wrong

1. **stopifnot()** or use 'assertthat' package - check inputs are correct
2. **Avoid subset(), transform() and with()** - these are non-standard evaluation, when they fail, often fail with uninformative error messages.
3. **Avoid [and sapply()** - functions that can return different types of output.
   - Recommendation : Whenever subsetting a data frame in a function, you should always use **drop = FALSE**

Updated: 2/16

Nested Data

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

"cell" contents

nested data frame

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

n_iris

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]]

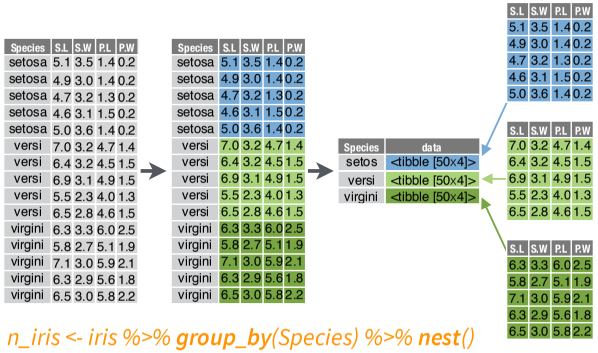
Sepal.L	Sepal.W	Petal.L	Petal.W
6.3	3.3	6.0	2.5
5.8	2.7	5.1	1.9
7.1	3.0	5.9	2.1
6.3	2.9	5.6	1.8
6.5	3.0	5.8	2.2

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

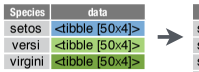


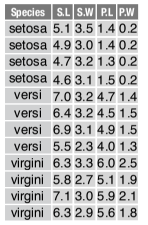
*n_iris <- iris %>% **group_by**(Species) %>% **nest**()*

nest(data, ..., .key = data)

For grouped data, moves groups into cells as data frames.

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





*n_iris %>% **unnest**()*

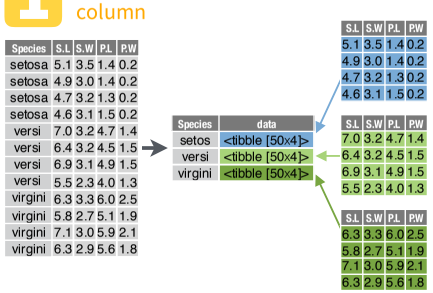
unnest(data, ..., .drop = NA, .id=NULL, .sep=NULL)

Unnests a nested data frame.

List Column Workflow

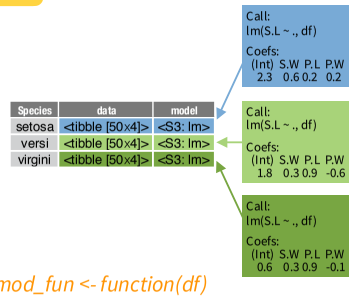
Nested data frames use a **list column**, a list that is stored as a column vector of a data frame. A typical **workflow** for list columns:

1 **Make** a list column



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
)
```

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

tibble::**tibble**(…)
Saves list input as list columns
tibble(max = c(3, 4, 5), seq = list(1:3, 1:4, 1:5))

tibble::**enframe**(x, name="name", value="value")
Converts multi-level list to tibble with list cols
enframe(list('3'=1:3, '4'=1:4, '5'=1:5), 'max', 'seq')

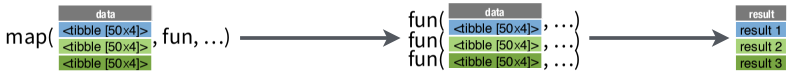
dplyr::**mutate**(.data, …) Also **transmute**()
Returns list col when result returns list.
*mtcars %>% **mutate**(seq = **map**(cyl, seq))*

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

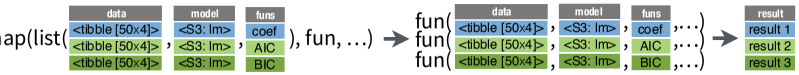
Use **map()**, **map2()**, and **pmap()** to apply a function that returns a result element-wise to the cells of a list column. **walk()**, **walk2()**, and **pwalk()** work the same way, but return a side effect. Each of these is in the **purrr** package.

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()**, and **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_int**(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))*

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**)**

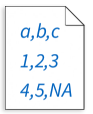
String vector to file, one element per line
write_lines(x,path, na = "NA", append = FALSE**)**

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**)**

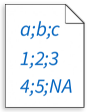
Read Tabular Data - These functions share the common arguments:

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()**)**



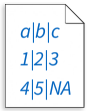
A	B	C
1	2	3
4	5	NA

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")



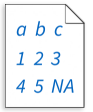
A	B	C
1	2	3
4	5	NA

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

Files with Any Delimiter
read_delim("file.txt", delim = "|"**)**
write_file(x = "a|b|c\n1|2|3\n4|5|NA", path = "file.txt")

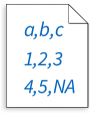


A	B	C
1	2	3
4	5	NA

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



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

A	B	C
1	2	3
4	5	NA

No header
read_csv(f, **col_names = FALSE**)

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

Provide header
read_csv(f, **col_names = c("x", "y", "z")**)

1	2	3
4	5	NA

Skip lines
read_csv(f, **skip = 1**)

A	B	C
1	2	3

Read in a subset
read_csv(f, **n_max = 1**)

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()))

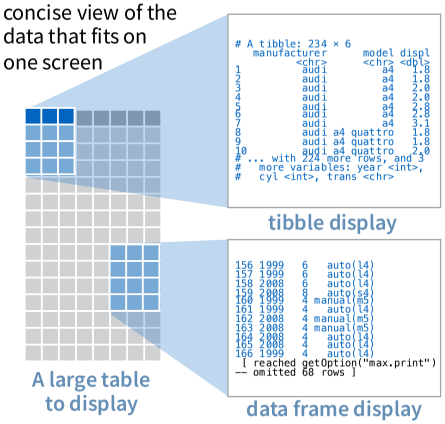
3. Else, read in as character vectors then parse with a parse_ function.
- **parse_guess()**
- **parse_character()**
- **parse_datetime()** Also **parse_date()** and **parse_time()**
- **parse_double()**
- **parse_factor()**
- **parse_integer()**
- **parse_logical()**
- **parse_number()**

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



- 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")

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")
Convert named vector to a tibble

is_tibble(x) Test whether x is a tibble.

Tidy Data with Tidyr

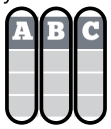
Tidy data is a way to organize tabular data. It provides a consistent data structure across packages.

A table is tidy if:

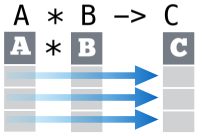
Each **variable** is in its own **column**

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.

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.

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))

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.

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(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 = "")

Data Wrangling
with dplyr and tidyr
Cheat Sheet

RStudio

Tidy Data - A foundation for wrangling in R

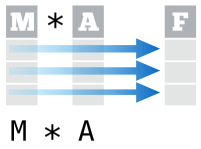
In a tidy data set:

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

&

Each **observation** is saved in its own **row**

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



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]

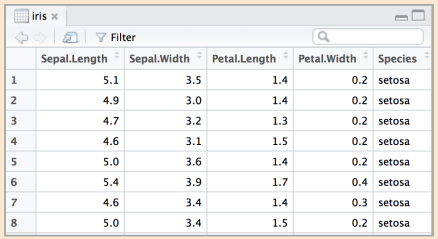
   Sepal.Length Sepal.Width Petal.Length
1           5.1         3.5          1.4
2           4.9         3.0          1.4
3           4.7         3.2          1.3
4           4.6         3.1          1.5
5           5.0         3.6          1.4
..          ...         ...          ...
Variables not shown: 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.

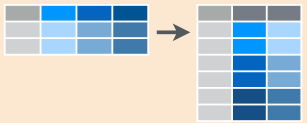
`x %>% f(y)` *is the same as* `f(x, y)`

`y %>% f(x, ., z)` *is the same as* `f(x, y, z )`

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

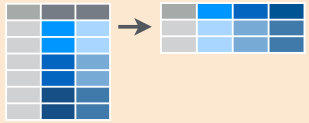
```
iris %>%
  group_by(Species) %>%
  summarise(avg = mean(Sepal.Width)) %>%
  arrange(avg)
```

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).

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

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.

devtools::install_github("rstudio/EDAWR") for data sets

Learn more with browseVignettes(package = c("dplyr", "tidyr")) • dplyr 0.4.0• tidyr 0.2.0 • Updated: 1/15

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:

dplyr::**first**
First value of a vector.

dplyr::**last**
Last value of a vector.

dplyr::**nth**
Nth value of a vector.

dplyr::**n**
# of values in a vector.

dplyr::**n_distinct**
# of distinct values in a vector.

IQR
IQR of a vector.

min
Minimum value in a vector.

max
Maximum value in a vector.

mean
Mean value of a vector.

median
Median value of a vector.

var
Variance of a vector.

sd
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:

dplyr::**lead**
Copy with values shifted by 1.

dplyr::**lag**
Copy with values lagged by 1.

dplyr::**dense_rank**
Ranks with no gaps.

dplyr::**min_rank**
Ranks. Ties get min rank.

dplyr::**percent_rank**
Ranks rescaled to [0, 1].

dplyr::**row_number**
Ranks. Ties got to first value.

dplyr::**ntile**
Bin vector into n buckets.

dplyr::**between**
Are values between a and b?

dplyr::**cume_dist**
Cumulative distribution.

dplyr::**cumall**
Cumulative `all`

dplyr::**cumany**
Cumulative `any`

dplyr::**cummean**
Cumulative `mean`

cumsum
Cumulative `sum`

cummax
Cumulative `max`

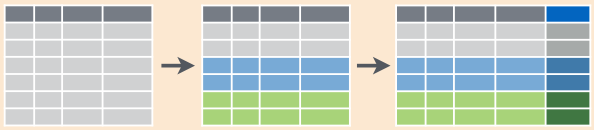
cummin
Cumulative `min`

cumprod
Cumulative `prod`

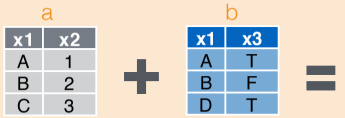
pmax
Element-wise `max`

pmin
Element-wise `min`

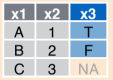
iris %>% group_by(Species) %>% mutate(…)
Compute new variables by group.



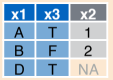
Combine Data Sets



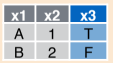
Mutating Joins



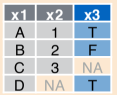
dplyr::**left_join(a, b, by = "x1")**
Join matching rows from b to a.



dplyr::**right_join(a, b, by = "x1")**
Join matching rows from a to b.

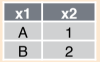


dplyr::**inner_join(a, b, by = "x1")**
Join data. Retain only rows in both sets.

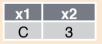


dplyr::**full_join(a, b, by = "x1")**
Join data. Retain all values, all rows.

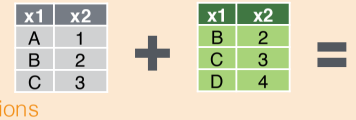
Filtering Joins



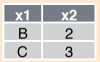
dplyr::**semi_join(a, b, by = "x1")**
All rows in a that have a match in b.



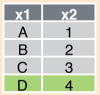
dplyr::**anti_join(a, b, by = "x1")**
All rows in a that do not have a match in b.



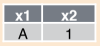
Set Operations



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

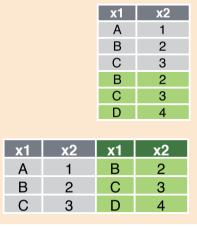


dplyr::**union(y, z)**
Rows that appear in either or both y and z.



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

Binding



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 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))

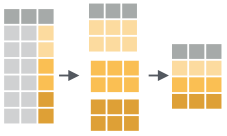
count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
count(iris, Species)

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 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))

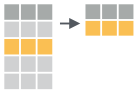
group_by(.data, ..., add = FALSE)
Returns copy of table grouped by …
g_iris <- group_by(iris, Species)

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

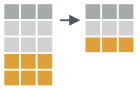
Manipulate Cases

EXTRACT CASES

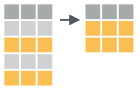
Row functions return a subset of rows as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.



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

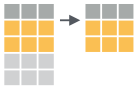


distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values. Also **distinct_()**. *distinct(iris, Species)*



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()

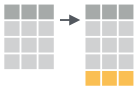
<	<=	is.na()	%in%	|	xor()
>	>=	!is.na()	!	&	

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)

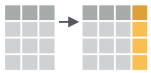
Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

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).

vectorized function



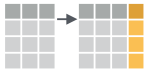
mutate(.data, …)
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

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.

vectorized function

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.

summary function

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

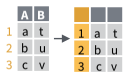
quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

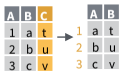
IQR() - Inter-Quartile Range
mad() - mean absolute deviation
sd() - standard deviation
var() - variance

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")

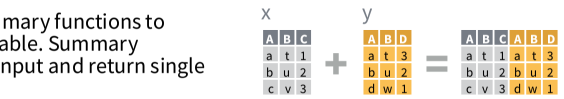


column_to_rownames()
Move col in row names.
column_to_rownames(a, var = "C")

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.

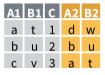
right_join(x, y, by = NULL, copy = FALSE, suffix=c(".x",".y"),…)
Join matching values from x to y.

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.

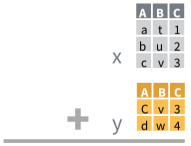
Use **by = c("col1", "col2")** to specify the column(s) to match on.
left_join(x, y, by = "A")

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)

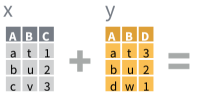
intersect(x, y, …)
Rows that appear in both x and z.

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 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.

required

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

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))
```

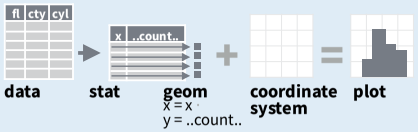
l + geom_contour(aes(z = z))
x, y, z, alpha, colour, group, linetype, size, weight

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

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 | geommappings

i + stat_density2d(aes(fill = ..level..), geom = "polygon")

variable created by stat

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.

(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

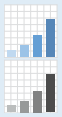
scale_*_continuous() - map cont' values to visual ones
scale_*_discrete() - map discrete values to visual ones
scale_*_identity() - use data values **as** visual ones
scale_*_manual(values = c()) - map discrete values to manually chosen visual ones
scale_*_date(date_labels = "%m/%d"), date_breaks = "2 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 & 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)

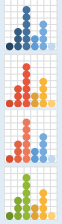


n <- d + geom_bar(aes(fill = fl))

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..))

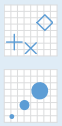
o + scale_fill_distiller(palette = "Blues")

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

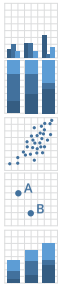
r + coord_polar(theta = "x", direction=1)
theta, start, direction
Polar coordinates

r + coord_trans(ytrans = "sqrt")
xtrans, ytrans, limx, limy
Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.

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

Position Adjustments

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



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

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

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

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

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

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

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

Themes

r + theme_bw()
White background with grid lines

r + theme_gray()
Grey background (default theme)

r + theme_dark()
dark for contrast

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

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

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(0, 100))

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.

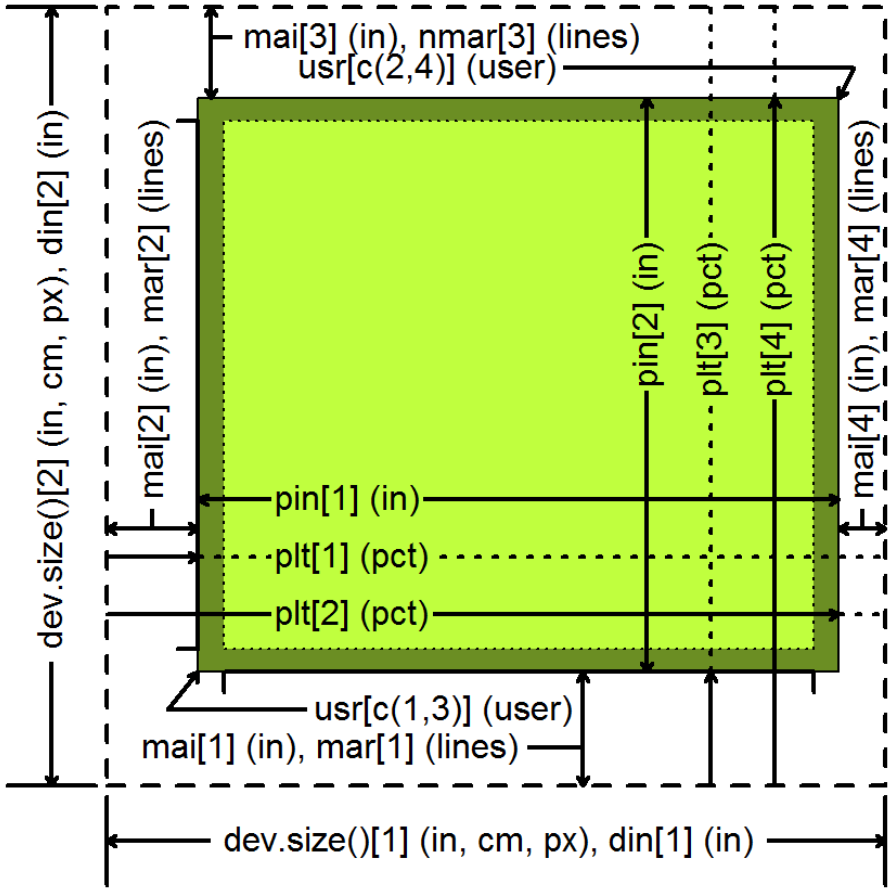
Your graphics device

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="i"** and/or the **yaxs="i"** 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 Pyhthagoras.

```
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[y]
angle.radians <- atan2(delta.y, delta.x)
angle.degrees <- angle.radians * 180 / pi
text(x1, y1, "TEXT", srt=angle.degrees)
```

 Steve Simon, P.Mean Consulting • mail@pmean.com • https://www.rstudio.com/resources/cheatsheets/

Panels

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

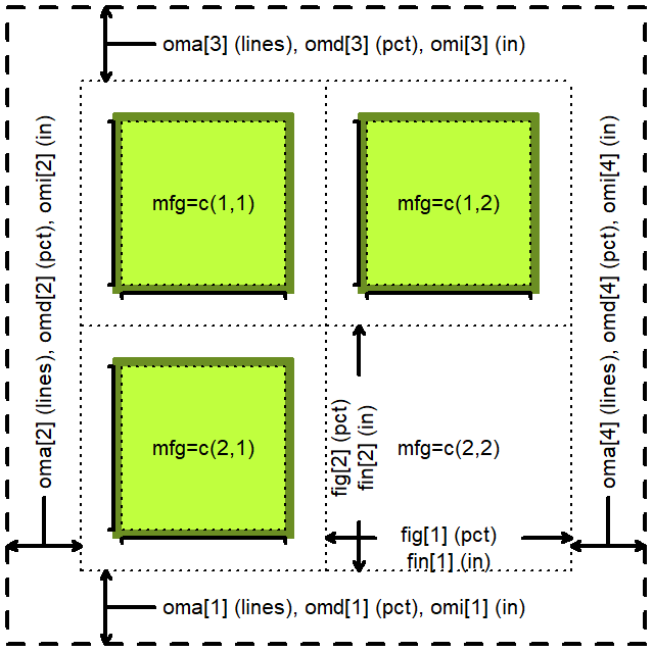
If you display multiple plots within a single graphics window (e.g., with the **mfrow** or **mfcol** arguments of **par** or with the **layout** function), then the **fig** and **fin** arguments will tell you the size of the current subplot window in percent or inches, respectively.

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 of 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.

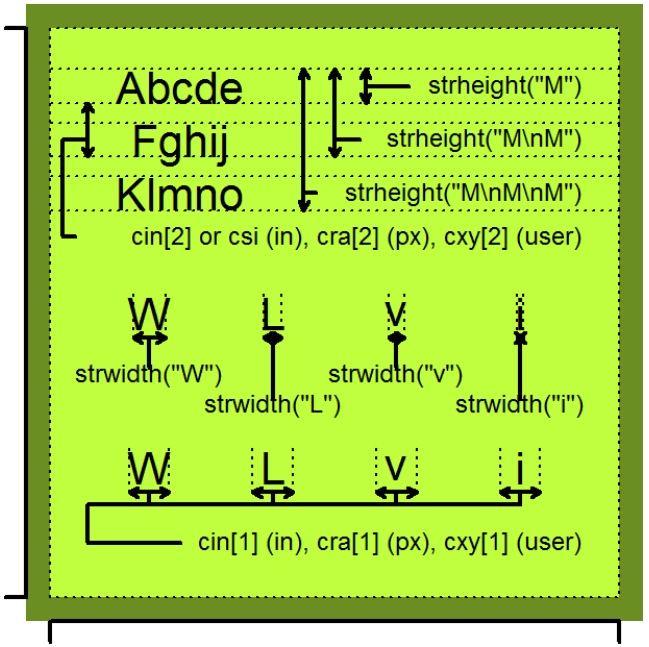
par("cin") *(r.o.)* (width, height) in inches
par("csi") *(r.o.)* height in inches
par("cra") *(r.o.)* (width, height) in pixels
par("cxy") *(r.o.)* (width, height) in xy coordinates

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 with 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")
px.per.xy <- par("cra") / par("cxy")
xy.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.

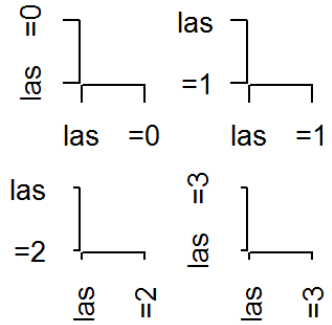
3. Use the **cex** argument to increase or decrease the relative size of your fonts.

If your axes don't fit

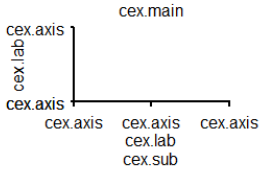
There are several possible solutions.

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.



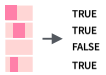
3. change the relative size of the font
   a. **cex.axis** for the tick mark labels.
   b. **cex.lab** for **xlab** and **ylab**.
   c. **cex.main** for the main title
   d. **cex.sub** for the subtitle.



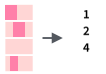
Work with strings with stringr : : CHEAT SHEET

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")*



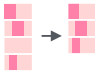
str_which(string, **pattern**) Find the indexes of strings that contain a pattern match. *str_which(fruit, "a")*

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)*



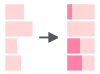
str_subset(string, **pattern**) Return only the strings that contain a pattern match. *str_subset(fruit, "b")*

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)*



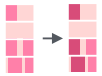
str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. *str_pad(fruit, 17)*

str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. *str_trunc(fruit, 3)*

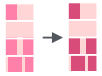
str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. *str_trim(fruit)*

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")[1] Convert strings to lower case. *str_to_lower(sentences)*

str_to_upper(string, locale = "en")[1] Convert strings to upper case. *str_to_upper(sentences)*

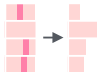
str_to_title(string, locale = "en")[1] Convert strings to title case. *str_to_title(sentences)*

Join and Split

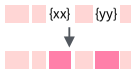
str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. *str_c(letters, LETTERS)*

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, ...)[1] 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, ...)[1] 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)*

[1] See 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)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	\.	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	\!	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	\?	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	\\	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	\(	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	\)	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	\{	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	\}	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	\n	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	\t	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	\s	any whitespace (**\S** *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(){}
\\d	\d	any digit (**\D** *for non-digits*)	see("\\d")	abc ABC 123 .!?\(){}
\\w	\w	any word character (**\W** *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(){}
\\b	\b	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:] [1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:] [1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:] [1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:] [1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:] [1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:] [1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:] [1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:] [1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:] [1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

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

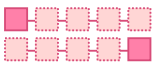
ALTERNATES

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

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

ANCHORS

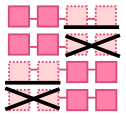
anchor <- function(rx) str_view_all("aaa", rx)



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

LOOK AROUNDS

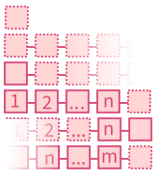
look <- function(rx) str_view_all("bacad", rx)



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

QUANTIFIERS

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



regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly **n**	quant("a{2}")	.a.aa.aaa
a{n, }	**n** or more	quant("a{2,}")	.a.aa.aaa
a{n, m}	between **n** and **m**	quant("a{2,4}")	.a.aa.aaa

GROUPS

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

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

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

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

string (type this)	regexp (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))	
\\1	**\1** (etc.)	first () group, etc.	ref("(a)(b)\\2\\1")	abbaab

[:space:]
↵ new line

[:blank:]
space
tab

.

[:graph:]

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

[:alnum:]

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

[:alpha:]

[:lower:]
a b c d e f
g h i j k l
m n o p q r
s t u v w x
z

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

R Markdown : : CHEAT SHEET

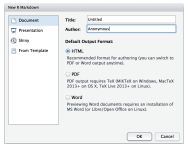
What is R Markdown?

.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.

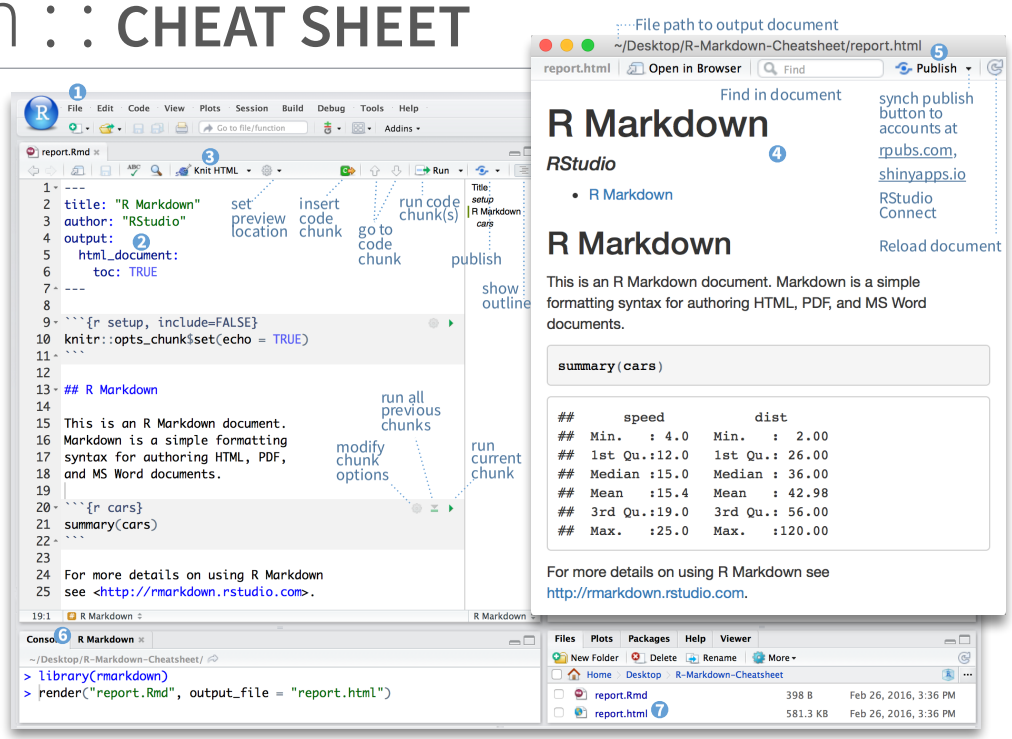
Reproducible Research • At the click of a button, or the type of a command, you can rerun the code in an R Markdown file to reproduce your work and export the results as a finished report.

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



.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 ```` ```{r} ````

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

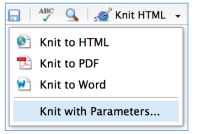
2. **Call parameters** • Call parameter values in code as params$<name>

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

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

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

Today's date is `r params$d`



render

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

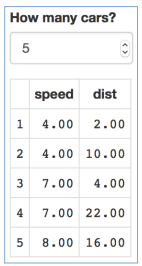
input - file to render **output_format**	**output_options** - List of render options (as in YAML)	**output_file** **output_dir**	**params** - list of params to use	**envir** - environment to evaluate code chunks in	**encoding** - of input file

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

```
---
output: html_document
runtime: shiny
---

```{r, echo = FALSE}
numericInput("n",
  "How many cars?", 5)

renderTable({
  head(cars, input$n)
})
```
```



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

NOTE: *Your report will 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.*

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 ```` ```{r} ```` and ```` ``` ````. Place chunk options within curly braces, after **r**. Insert with

```
```{r echo=TRUE}
getRversion()
```
```

```
getRversion()
## [1] '3.2.3'
```

GLOBAL OPTIONS

Set with knitr::**opts_chunk$set()**, e.g.

```
```{r include=FALSE}
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

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^2^~2~
~~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-->

\textbf{Tex ignored in HTML}
<em>HTML ignored in pdfs</em>

<http://www.rstudio.com>
[link](www.rstudio.com)
Jump to [Header 1](#anchor)
image:

![Caption](smallorb.png)

* 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

| 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.
```

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

$$E = mc^2$$

> block quote

Header1

Header 2

Header 3

Header 4

Header 5

Header 6

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

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. ↩

Set render options with YAML

When you render, R Markdown

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
```

Indent 2 spaces

Indent 4 spaces

html tabsets

Use tablet css class to place sub-headers into tabs

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

Tabset

Tab 1 | Tab 2

text 1

End tabset

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	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	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	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									

Create a Reusable Template

1. **Create a new package** with a 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
—
```

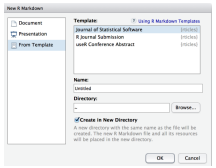


Table Suggestions

Several functions format R data into tables

Table with kable

eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62

	eruptions	waiting
1	3.60	79.00
2	1.80	54.00
3	3.33	74.00
4	2.28	62.00

Table with xtable

Table with stargazer

	eruptions	waiting
1	3.600	79
2	1.800	54
3	3.333	74
4	2.283	62

```
data <- faithful[1:4, ]
```

```{r results = 'asis'}
knitr::kable(data, caption = "Table with kable")
```

```{r results = "asis"}
print(xtable::xtable(data, caption = "Table with xtable"),
      type = "html", html.table.attributes = "border=0"))
```

```{r results = "asis"}
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
csl: 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.
```

Basic Regular Expressions in R
Cheat Sheet

Character Classes

[[:digit:]] or \\d	Digits; `[0-9]`
\\D	Non-digits; `[^0-9]`
[[:lower:]]	Lower-case letters; `[a-z]`
[[:upper:]]	Upper-case letters; `[A-Z]`
[[:alpha:]]	Alphabetic characters; `[A-z]`
[[:alnum:]]	Alphanumeric characters `[A-z0-9]`
\\w	Word characters; `[A-z0-9_]`
\\W	Non-word characters
[[:xdigit:]] or \\x	Hexadec. digits; `[0-9A-Fa-f]`
[[:blank:]]	Space and tab
[[:space:]] or \\s	Space, tab, vertical tab, newline, form feed, carriage return
\\S	Not space; `[^[:space:]]`
[[:punct:]]	Punctuation characters; `!"#$%&'()*+,-./:;<=>?@[]^_`{|}~`
[[:graph:]]	Graphical char.; `[[:alnum:][:punct:]]`
[[:print:]]	Printable characters; `[[:alnum:][:punct:]\\s]`
[[:cntrl:]] or \\c	Control characters; `\n, \r` etc.

Special Metacharacters

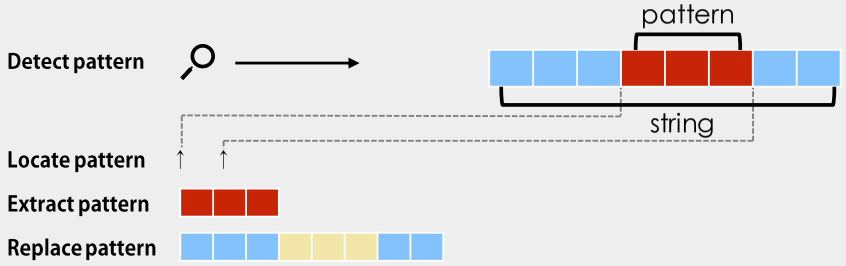
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed

Lookaraounds and Conditionals*

(?=)	Lookahead (requires PERL = TRUE), e.g. `(?=yx)`: position followed by `'xy'`
(?!)	Negative lookahead (PERL = TRUE); position NOT followed by pattern
(?<=)	Lookbehind (PERL = TRUE), e.g. `(?<=yx)`: position following `'xy'`
(?<!)	Negative lookbehind (PERL = TRUE); position NOT following pattern
?(if)then	If-then-condition (PERL = TRUE); use lookaheads, optional char. etc in if-clause
?(if)then|else	If-then-else-condition (PERL = TRUE)

*see, e.g. http://www.regular-expressions.info/lookaround.html
http://www.regular-expressions.info/conditional.html

Functions for Pattern Matching




```
> string <- c("Hiphopopotamus", "Rhymenoceros", "time for bottomless lyrics")
> pattern <- "t.m"
```

Detect Patterns

grep(pattern, string)
```
[1] 1 3
```
grep(pattern, string, value = TRUE)
```
[1] "Hiphopopotamus"
[2] "time for bottomless lyrics"
```
grepl(pattern, string)
```
[1]  TRUE FALSE  TRUE
```
stringr::str_detect(string, pattern)
```
[1]  TRUE FALSE  TRUE
```

Locate Patterns

regexpr(pattern, string)
find starting position and length of first match

gregexpr(pattern, string)
find starting position and length of all matches

stringr::**str_locate(string, pattern)**
find starting and end position of first match

stringr::**str_locate_all(string, pattern)**
find starting and end position of all matches

Split a String using a Pattern

strsplit(string, pattern) or stringr::**str_split(string, pattern)**

Extract Patterns

regmatches(string, regexpr(pattern, string))
extract first match
```
[1] "tam" "tim"
```
regmatches(string, gregexpr(pattern, string))
extracts all matches, outputs a list
```
[[1]] "tam" [[2]] character(0) [[3]] "tim" "tom"
```
stringr::**str_extract(string, pattern)**
extract first match
```
[1] "tam" NA  "tim"
```
stringr::**str_extract_all(string, pattern)**
extract all matches, outputs a list

stringr::**str_extract_all(string, pattern, simplify = TRUE)**
extract all matches, outputs a matrix

stringr::**str_match(string, pattern)**
extract first match + individual character groups

stringr::**str_match_all(string, pattern)**
extract all matches + individual character groups

Replace Patterns

sub(pattern, replacement, string)
replace first match

gsub(pattern, replacement, string)
replace all matches

stringr::**str_replace(string, pattern, replacement)**
replace first match

stringr::**str_replace_all(string, pattern, replacement)**
replace all matches

Character Classes and Groups

.	Any character except \n
|	Or, e.g. (a|b)
[…]	List permitted characters, e.g. [abc]
[a-z]	Specify character ranges
[^…]	List excluded characters
(…)	Grouping, enables back referencing using \\N where N is an integer

Anchors

^	Start of the string
$	End of the string
\\b	Empty string at either edge of a word
\\B	NOT the edge of a word
\\<	Beginning of a word
\\>	End of a word

Quantifiers

*	Matches at least 0 times
+	Matches at least 1 time
?	Matches at most 1 time; optional string
{n}	Matches exactly n times
{n,}	Matches at least n times
{,n}	Matches at most n times
{n,m}	Matches between n and m times

General Modes

By default R uses *POSIX extended regular expressions*. You can switch to *PCRE regular expressions* using `PERL = TRUE` for base or by wrapping patterns with `perl()` for stringr.

All functions can be used with literal searches using `fixed = TRUE` for base or by wrapping patterns with `fixed()` for stringr.

All base functions can be made case insensitive by specifying `ignore.cases = TRUE`.

Escaping Characters

Metacharacters (`. * +` etc.) can be used as literal characters by escaping them. Characters can be escaped using \\ or by enclosing them in `\\Q...\\E`.

Case Conversions

Regular expressions can be made case insensitive using `(?i)`. In backreferences, the strings can be converted to lower or upper case using `\\L` or `\\U` (e.g. `\\L\\1`). This requires `PERL = TRUE`.

Greedy Matching

By default the asterisk * is greedy, i.e. it always matches the longest possible string. It can be used in lazy mode by adding `?`, i.e. `*?`.

Greedy mode can be turned off using `(?U)`. This switches the syntax, so that `(?U)a*` is lazy and `(?U)a*?` is greedy.

Note

Regular expressions can conveniently be created using rex::**rex()**.

Updated: 09/16

R Markdown Cheat Sheet

learn more at rmarkdown.rstudio.com

rmarkdown 0.2.50 Updated: 8/14

RStudio

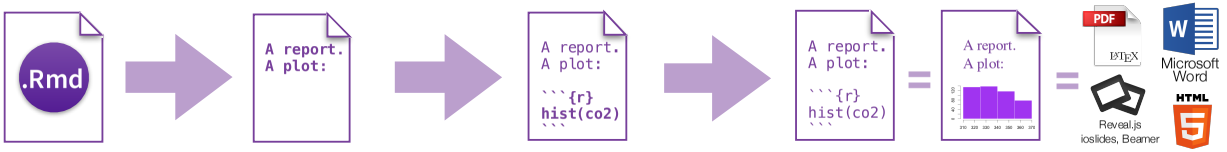
1. Workflow R Markdown is a format for writing reproducible, dynamic reports with R. Use it to embed R code and results into slideshows, pdfs, html documents, Word files and more. To make a report:

i. Open - Open a file that uses the .Rmd extension.

ii. Write - Write content with the easy to use R Markdown syntax

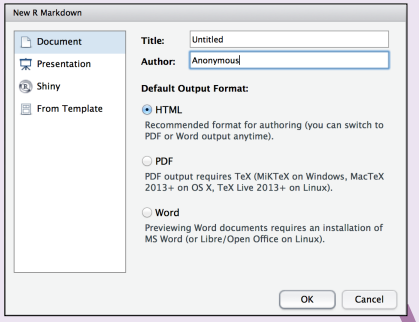
iii. Embed - Embed R code that creates output to include in the report

iv. Render - Replace R code with its output and transform the report into a slideshow, pdf, html or ms Word file.



2. Open File Start by saving a text file with the extension .Rmd, or open an RStudio Rmd template

- In the menu bar, click **File ▸ New File ▸ R Markdown...**
- A window will open. Select the class of output you would like to make with your .Rmd file
- Select the specific type of output to make with the radio buttons (you can change this later)
- Click OK



3. Markdown Next, write your report in plain text. Use markdown syntax to describe how to format text in the final report.

syntax

```
Plain text
End a line with two spaces to start a new paragraph.
*italics* and _italics_
**bold** and __bold__
superscript^2^
~~strikethrough~~
[link](www.rstudio.com)

# Header 1

## Header 2

### Header 3

#### Header 4

##### Header 5

###### Header 6

endash: --
emdash: ---
ellipsis: ...
inline equation: $A = \pi*r^{2}$
image: ![](path/to/smallorb.png)

horizontal rule (or slide break):

***

> block quote

* unordered list
* item 2
    + sub-item 1
    + sub-item 2

1. ordered list
2. item 2
    + sub-item 1
    + sub-item 2

Table Header  | Second Header
------------- | -------------
Table Cell    | Cell 2
Cell 3        | Cell 4
```

becomes

Plain text

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

italics and *italics*

bold and **bold**

superscript^2

~~strikethrough~~

link

Header 1

Header 2

Header 3

Header 4

Header 5

Header 6

endash: –

emdash: —

ellipsis: …

inline equation: $A = \pi * r^2$

image:

horizontal rule (or slide break):

> block quote

- unordered list
- item 2
    - sub-item 1
    - sub-item 2

1. ordered list
2. item 2
    - sub-item 1
    - sub-item 2

Table Header	Second Header
Table Cell	Cell 2
Cell 3	Cell 4

4. Choose Output Write a YAML header that explains what type of document to build from your R Markdown file.

YAML

A YAML header is a set of key: value pairs at the start of your file. Begin and end the header with a line of three dashes (- - -)

```
---
title: "Untitled"
author: "Anonymous"
output: html_document
---

This is the start of my
report. The above is metadata
saved in a YAML header.
```

The RStudio template writes the YAML header for you

The output value determines which type of file R will build from your .Rmd file (in Step 6)

output: html_document	html file (web page)
output: pdf_document	pdf document
output: word_document	Microsoft Word .docx
output: beamer_presentation	beamer slideshow (pdf)
output: ioslides_presentation	ioslides slideshow (html)

5. Embed Code Use knitr syntax to embed R code into your report. R will run the code and include the results when you render your report.

inline code

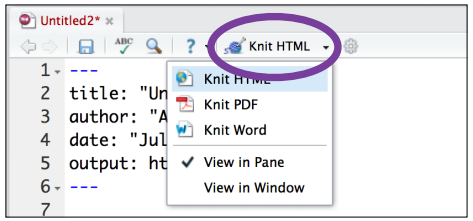
Surround code with back ticks and r.
R replaces inline code with its results.

```
Two plus two equals `r 2 + 2`.
```

Two plus two equals 4.

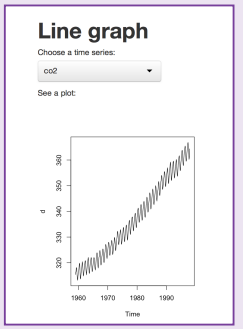
code chunks

Start a chunk with ```{r}.
End a chunk with ```

```
Here's some code
```{r}
dim(iris)
```
```

Here's some code

```
dim(iris)
```

```
## [1] 150   5
```

display options

Use knitr options to style the output of a chunk.
Place options in brackets above the chunk.

```
Here's some code
```{r eval=FALSE}
dim(iris)
```
```

Here's some code

```
dim(iris)
```

```
Here's some code
```{r echo=FALSE}
dim(iris)
```
```

Here's some code

```
## [1] 150   5
```

option	default	effect
eval	TRUE	Whether to evaluate the code and include its results
echo	TRUE	Whether to display code along with its results
warning	TRUE	Whether to display warnings
error	FALSE	Whether to display errors
message	TRUE	Whether to display messages
tidy	FALSE	Whether to reformat code in a tidy way when displaying it
results	"markup"	"markup", "asis", "hold", or "hide"
cache	FALSE	Whether to cache results for future renders
comment	"##"	Comment character to preface results with
fig.width	7	Width in inches for plots created in chunk
fig.height	7	Height in inches for plots created in chunk

For more details visit yihui.name/knitr/

6. Render Use your .Rmd file as a blueprint to build a finished report.

Render your report in one of two ways

1. Run **rmarkdown::render("<file path>")**
2. Click the **knit HTML** button at the top of the RStudio scripts pane



When you render, R will

- execute each embedded code chunk and insert the results into your report
- build a new version of your report in the output file type
- open a preview of the output file in the viewer pane
- save the output file in your working directory

7. Interactive Docs Turn your report into an interactive Shiny document in 3 steps

1 Add **runtime: shiny** to the YAML header

```
---
title: "Line graph"
output: html_document
runtime: shiny
---
```

2 In the code chunks, add Shiny **input** functions to embed widgets. Add Shiny **render** functions to embed reactive output

```
---
title: "Line graph"
output: html_document
runtime: shiny
---

Choose a time series:
```{r echo = FALSE}
selectInput("data", "",
  c("co2", "lh"))
```
See a plot:
```{r echo = FALSE}
renderPlot({
  d <- get(input$data)
  plot(d)
})
```
```

3 Render with **rmarkdown::run** or click **Run Document** in RStudio



* *Note: your report will be a Shiny app, which means you must choose an html output format, like* ***html_document*** *(for an interactive report) or* ***ioslides_presentation*** *(for an interactive slideshow).*

8. Publish Share your report where users can visit it online

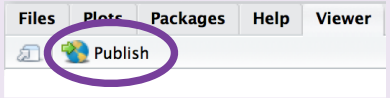
Rpubs.com

Share non-interactive documents on RStudio's free R Markdown publishing site
www.rpubs.com

ShinyApps.io

Host an interactive document on RStudio's server. Free and paid options
www.shinyapps.io

Click the "Publish" button in the RStudio preview window to publish to rpubs.com with one click.



9. Learn More

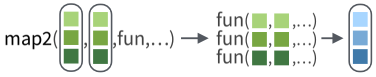
Documentation and examples - rmarkdown.rstudio.com
Further Articles - shiny.rstudio.com/articles
- blog.rstudio.com
- @rstudio

Apply functions with purrr : : **CHEAT SHEET**

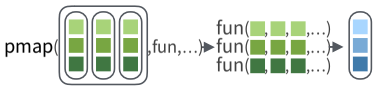
Apply Functions

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

map(.x, .f, …) Apply a function to each element of a list or vector. *map(x, is.logical)*



map2(.x, ,y, .f, …) Apply a function to pairs of elements from two lists, vectors. *map2(x, y, sum)*



pmap(.l, .f, …) Apply a function to groups of elements from list of lists, vectors. *pmap(list(x, y, z), sum, na.rm = TRUE)*

invoke_map(.f, .x = list(NULL), …, .env=NULL) Run each function in a list. Also **invoke**. *l <- list(var, sd); invoke_map(l, x = 1:9)*

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
map	list
map_chr	character vector
map_dbl	double (numeric) vector
map_dfc	data frame (column bind)
map_dfr	data frame (row bind)
map_int	integer vector
map_lgl	logical vector
walk	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*

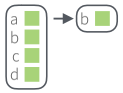
~ . becomes **function(x)** **x**. e.g. *map(l, ~ 2 +.)* becomes *map(l, function(x) 2 + x)*

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

~ ..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

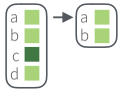


pluck(.x, ..., .default=NULL) Select an element by name or index, *pluck(x,"b")* ,or its attribute with **attr_getter**. *pluck(x,"b",attr_getter("n"))*

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

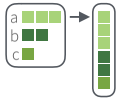
discard(.x, .p, …) Select elements that do not pass a logical test. *discard(x, is.na)*

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

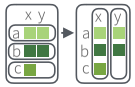


head_while(.x, .p, …) Return head elements until one does not pass. Also **tail_while**. *head_while(x, is.character)*

RESHAPE LISTS



flatten(.x) Remove a level of indexes from a list. Also **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. *flatten(x)*



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

SUMMARISE LISTS

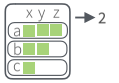
every(.x, .p, …) Do all element pass a test? *every(x, is.character)*

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

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

detect(.x, .f, ..., .right=FALSE, .p) Find first element to pass. *detect(x, is.character)*

detect_index(.x, .f, ..., .right = FALSE, .p) Find index of first element to pass. *detect_index(x, is.character)*



depth(x) Return depth (number of levels of indexes). *depth(x)*

JOIN (TO) LISTS

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

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

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

TRANSFORM LISTS

modify(.x, .f, ...) Apply function to each element. Also **map**, **map_chr**, **map_dbl**, **map_dfc**, **map_dfr**, **map_int**, **map_lgl**. *modify(x, ~.+ 2)*

modify_at(.x, .at, .f, ...) Apply function to elements by name or index. Also **map_at**. *modify_at(x, "b", ~.+ 2)*

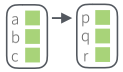
modify_if(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. *modify_if(x, is.numeric,~.+2)*

modify_depth(.x,.depth,.f,...) Apply function to each element at a given level of a list. *modify_depth(x, 1, ~.+ 2)*

WORK WITH LISTS

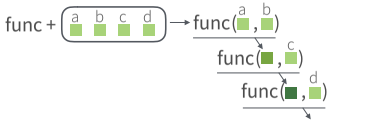
array_tree(array, margin = NULL) Turn array into list. Also **array_branch**. *array_tree(x, margin = 3)*

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

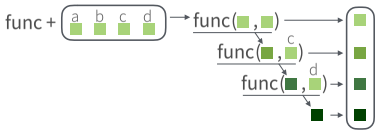


set_names(x, nm = x) Set the names of a vector/list directly or with a function. *set_names(x, c("p", "q", "r"))* *set_names(x, tolower)*

Reduce Lists



reduce(.x, .f, ..., .init) Apply function recursively to each element of a list or vector. Also **reduce_right**, **reduce2**, **reduce2_right**. *reduce(x, sum)*



accumulate(.x, .f, ..., .init) Reduce, but also return intermediate results. Also **accumulate_right**. *accumulate(x, sum)*

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.

"cell" contents

nested data frame

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

n_iris

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

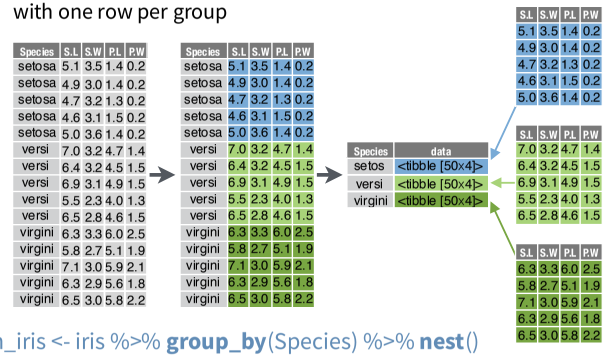
n_iris$data[[3]]

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:

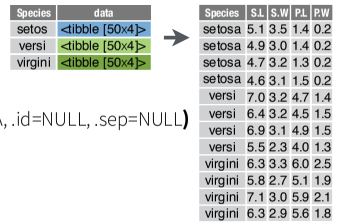
1. Group the data frame into groups with **dplyr::group_by()**
2. Use **nest()** to create a nested data frame with one row per group



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.

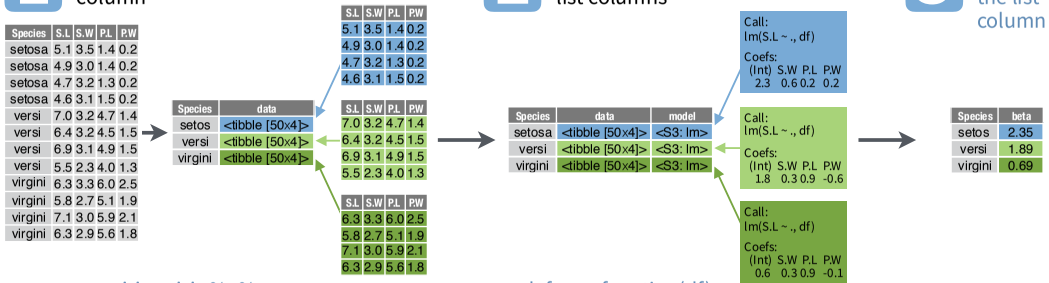
List Column Workflow

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

2 **Work with** list columns

3 **Simplify** the list column



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

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

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

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)

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

tibble::**tibble(**…**)**
Saves list input as list columns
tibble(max = c(3, 4, 5), seq = list(1:3, 1:4, 1:5))

tibble::**enframe(**x, name="name", value="value"**)**
Converts multi-level list to 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))

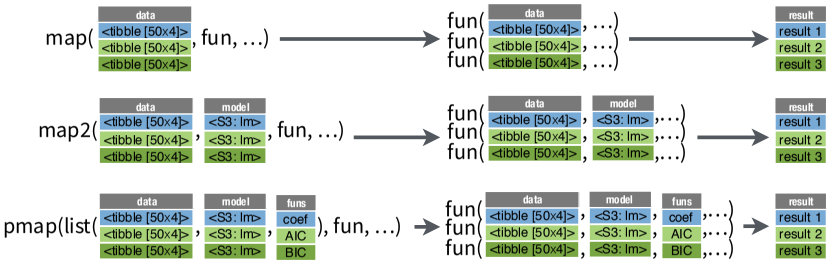
dplyr::**summarise(**.data, …**)**
Returns list col when result is wrapped with **list()**
mtcars %>% group_by(cyl) %>%
summarise(q = **list**(quantile(mpg)))

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))

caret Package

Cheat Sheet

Specifying the Model

Possible syntaxes for specifying the variables in the model:

```
train(y ~ x1 + x2, data = dat, ...)
train(x = predictor_df, y = outcome_vector, ...)
train(recipe_object, data = dat, ...)
```

- `rfe`, `sbf`, `gafs`, and `safs` only have the `x/y` interface.
- The `train` formula method will **always** create dummy variables.
- The `x/y` interface to `train` will not create dummy variables (but the underlying model function might).

Remember to:

- Have column names in your data.
- Use factors for a classification outcome (not 0/1 or integers).
- Have valid R names for class levels (not "0"/"1")
- Set the random number seed prior to calling `train` repeatedly to get the same resamples across calls.
- Use the `train` option `na.action = na.pass` if you will being imputing missing data. Also, use this option when predicting new data containing missing values.

To pass options to the underlying model function, you can pass them to `train` via *the ellipses*:

```
train(y ~ ., data = dat, method = "rf",
      # options to `randomForest`:
      importance = TRUE)
```

Parallel Processing

The `foreach` package is used to run models in parallel. The `train` code does not change but a "`do`" package must be called first.

```
# on MacOS or Linux
library(doMC)
registerDoMC(cores=4)
```

```
# on Windows
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)
```

The function `parallel::detectCores` can help too.

Preprocessing

Transformations, filters, and other operations can be applied to the *predictors* with the `preProc` option.

```
train(, preProc = c("method1", "method2"), ...)
```

Methods include:

- `"center"`, `"scale"`, and `"range"` to normalize predictors.
- `"BoxCox"`, `"YeoJohnson"`, or `"expoTrans"` to transform predictors.
- `"knnImpute"`, `"bagImpute"`, or `"medianImpute"` to impute.
- `"corr"`, `"nzv"`, `"zv"`, and `"conditionalX"` to filter.
- `"pca"`, `"ica"`, or `"spatialSign"` to transform groups.

`train` determines the order of operations; the order that the methods are declared does not matter.

The `recipes` package has a more extensive list of preprocessing operations.

Adding Options

Many `train` options can be specified using the `trainControl` function:

```
train(y ~ ., data = dat, method = "cubist",
      trControl = trainControl(<options>))
```

Resampling Options

`trainControl` is used to choose a resampling method:

```
trainControl(method = <method>, <options>)
```

Methods and options are:

- `"cv"` for K-fold cross-validation (`number` sets the # folds).
- `"repeatedcv"` for repeated cross-validation (`repeats` for # repeats).
- `"boot"` for bootstrap (`number` sets the iterations).
- `"LGOCV"` for leave-group-out (`number` and `p` are options).
- `"LOO"` for leave-one-out cross-validation.
- `"oob"` for out-of-bag resampling (only for some models).
- `"timeslice"` for time-series data (options are `initialWindow`, `horizon`, `fixedWindow`, and `skip`).

Performance Metrics

To choose how to summarize a model, the `trainControl` function is used again.

```
trainControl(summaryFunction = <R function>,
             classProbs = <logical>)
```

Custom R functions can be used but `caret` includes several: `defaultSummary` (for accuracy, RMSE, etc), `twoClassSummary` (for ROC curves), and `prSummary` (for information retrieval). For the last two functions, the option `classProbs` must be set to `TRUE`.

Grid Search

To let `train` determine the values of the tuning parameter(s), the `tuneLength` option controls how many values **per tuning** parameter to evaluate.

Alternatively, specific values of the tuning parameters can be declared using the `tuneGrid` argument:

```
grid <- expand.grid(alpha = c(0.1, 0.5, 0.9),
                    lambda = c(0.001, 0.01))

train(x = x, y = y, method = "glmnet",
      preProc = c("center", "scale"),
      tuneGrid = grid)
```

Random Search

For tuning, `train` can also generate random tuning parameter combinations over a wide range. `tuneLength` controls the total number of combinations to evaluate. To use random search:

```
trainControl(search = "random")
```

Subsampling

With a large class imbalance, `train` can subsample the data to balance the classes them prior to model fitting.

```
trainControl(sampling = "down")
```

Other values are `"up"`, `"smote"`, or `"rose"`. The latter two may require additional package installs.

 Max Kuhn • max@rstudio.com • https://github.com/topepo/

Package Development: : CHEAT SHEET

Package Structure

A package is a convention for organizing files into directories.

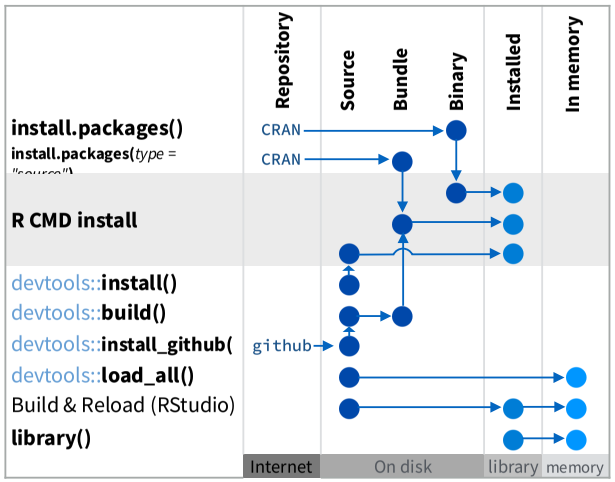
This sheet shows how to work with the 7 most common parts of an R package:

Package	
DESCRIPTION	SETUP
R/	WRITE CODE
tests/	TEST
man/	DOCUMENT
vignettes/	TEACH
data/	ADD DATA
NAMESPACE	ORGANIZE

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**
- Click on a function and press **F2** to open its definition
- Search for a function with **Ctrl + .**

Visit **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
expect_equal()	is equal within small numerical tolerance?
expect_identical()	is exactly equal?
expect_match()	matches specified string or regular
expect_output()	prints specified output?
expect_message()	displays specified message?
expect_warning()	displays specified warning?
expect_error()	throws specified error?
expect_is()	output inherits from certain class?
expect_false()	returns FALSE?
expect_true()	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

1. Add roxygen comments in your .R files
2. 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)

3. Open help pages with **?** to preview documentation
4. Repeat

.Rd FORMATTING TAGS

```
\emph{italic text}
\strong{bold text}
\code{function(args)}
\pkg{package}

\dontrun{code}
\dontshow{code}
\donttest{code}

\deqn{a + b (block)}
\eqn{a + b (inline)}

\email{name@@foo.com}
\href{url}{display}
\url{url}

\link[=dest]{display}
\linkS4class{class}
\code{\link{function}}
\code{\link[package]{function}}

\tabular{lcr}{
  left \tab centered \tab right \cr
  cell \tab cell     \tab cell  \cr
}
```

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

1. Modify your code or tests.
2. Document your package (devtools::**document()**)
3. Check NAMESPACE
4. Repeat until NAMESPACE is correct

SUBMIT YOUR PACKAGE
r-pkgs.had.co.nz/release.html

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)

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

RStudio

Shiny : : CHEAT SHEET

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 an 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 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:

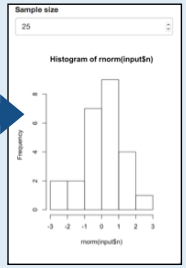
1. Refer to outputs with **output$<id>**
2. Refer to inputs with **input$<id>**
3. Wrap code in a **render*()** function before saving to output

```
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")
)
```

ui.R contains everything you would save to ui.

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

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

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

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

The directory name is the name of the app

(optional) defines objects available to both ui.R and server.R

(optional) used in showcase mode

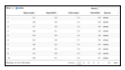
(optional) data, scripts, etc.

(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

works with

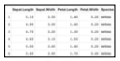


DT::**renderDataTable**(expr, options, callback, escape, env, quoted) — **dataTableOutput**(outputId, icon, …)

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

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

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



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

foo

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

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

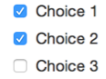
Inputs

collect values from the user

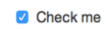
Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

actionButton(inputId, label, icon, …)

actionLink(inputId, label, icon, …)



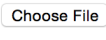
checkboxGroupInput(inputId, label, choices, selected, inline)



checkboxInput(inputId, label, value)

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

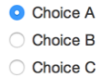
dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)



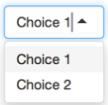
fileInput(inputId, label, multiple, accept)

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

passwordInput(inputId, label, value)

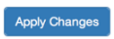


radioButtons(inputId, label, choices, selected, inline)

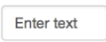


selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())

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



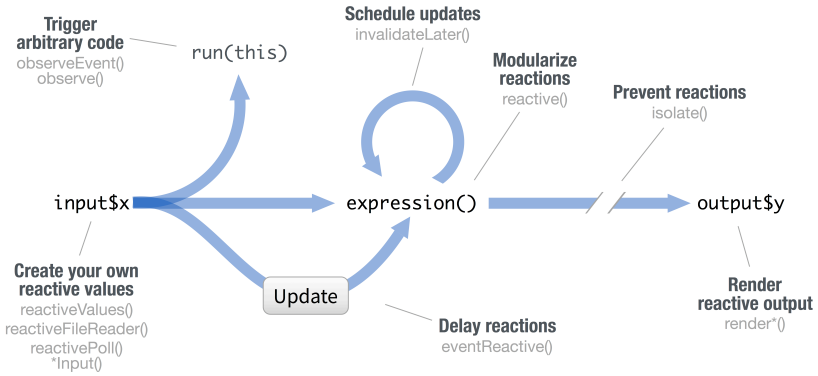
submitButton(text, icon)
(Prevents reactions across entire app)



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

```
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

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>")
)
```

CSS

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>"))
```

JS

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>"))
```

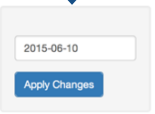
IMAGES

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.

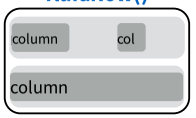
```
wellPanel(dateInput("a", ""),
          submitButton()
)
```



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

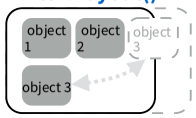
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

fluidRow()



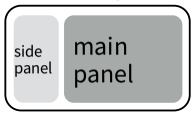

```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()



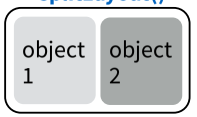

```
ui <- fluidPage(
  flowLayout( # object 1,
              # object 2,
              # object 3
  )
)
```

sidebarLayout()



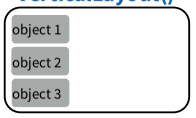

```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

splitLayout()




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

verticalLayout()



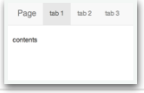

```
ui <- fluidPage(
  verticalLayout( # object 1,
                  # object 2,
                  # object 3
  )
)
```

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"))
```



RStudio

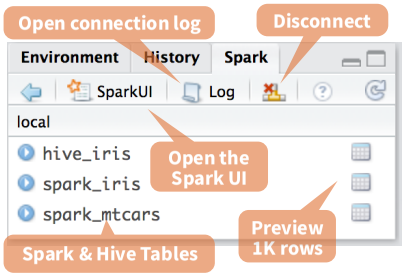
Data Science in Spark with Sparklyr : : CHEAT SHEET

Intro

sparklyr is an R interface for Apache Spark™, it provides a complete **dplyr** backend and the option to query directly using **Spark SQL** statement. With sparklyr, you can orchestrate distributed machine learning using either **Spark's MLlib** or **H2O** Sparkling Water.

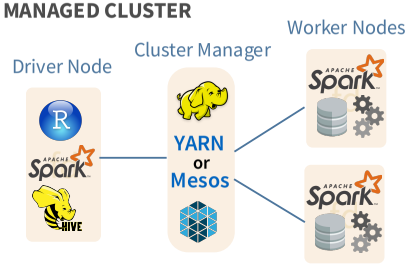
Starting with **version 1.044, RStudio Desktop, Server and Pro include integrated support for the sparklyr package**. You can create and manage connections to Spark clusters and local Spark instances from inside the IDE.

RStudio Integrates with sparklyr

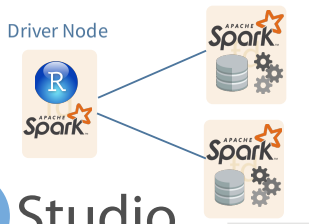


Cluster Deployment

MANAGED CLUSTER



STAND ALONE CLUSTER



RStudio

Data Science Toolchain with Spark + sparklyr

Import
- Export an R DataFrame
- Read a file
- Read existing Hive table

Tidy
- dplyr verb
- Direct Spark SQL (DBI)
- SDF function (Scala API)

Understand

Transform
Transformer function

Visualize
Collect data into R for plotting

Wrangle

Model
- Spark MLlib
- H2O Extension

Communicate
- Collect data into R
- Share plots, documents, and apps

R for Data Science, Grolemund & Wickham

Getting Started

LOCAL MODE (No cluster required)

1. Install a local version of Spark:
   spark_install ("2.0.1")
2. Open a connection
   sc <- spark_connect (master = "local")

ON A MESOS MANAGED CLUSTER

1. Install RStudio Server or Pro on one of the existing nodes
2. Locate path to the cluster's Spark directory
3. Open a connection
   spark_connect(master="[mesos URL]", version = "1.6.2", spark_home = [Cluster's Spark path])

USING LIVY (Experimental)

1. The Livy REST application should be running on the cluster
2. Connect to the cluster
   sc <- spark_connect(method = "livy", master = "http://host:port")

ON A YARN MANAGED CLUSTER

1. Install RStudio Server or RStudio Pro on one of the existing nodes, preferably an edge node
2. Locate path to the cluster's Spark Home Directory, it normally is "/usr/lib/spark"
3. Open a connection
   spark_connect(master="yarn-client", version = "1.6.2", spark_home = [Cluster's Spark path])

ON A SPARK STANDALONE CLUSTER

1. Install RStudio Server or RStudio Pro on one of the existing nodes or a server in the same LAN
2. Install a local version of Spark:
   spark_install (version = "2.0.1")
3. Open a connection
   spark_connect(master="spark://host:port", version = "2.0.1", spark_home = spark_home_dir())

Tuning Spark

EXAMPLE CONFIGURATION

```
config <- spark_config()
config$spark.executor.cores <- 2
config$spark.executor.memory <- "4G"
sc <- spark_connect (master="yarn-client",
config = config, version = "2.0.1")
```

IMPORTANT TUNING PARAMETERS with defaults

- spark.yarn.am.cores
- spark.yarn.am.memory *512m*
- spark.network.timeout *120s*
- spark.executor.memory *1g*
- spark.executor.cores *1*
- spark.executor.instances
- spark.executor.extraJavaOptions
- spark.executor.heartbeatInterval *10s*
- sparklyr.shell.executor-memory
- sparklyr.shell.driver-memory

Using sparklyr

A brief example of a data analysis using Apache Spark, R and sparklyr in local mode

```
library(sparklyr); library(dplyr); library(ggplot2);
library(tidyr);
set.seed(100)
spark_install("2.0.1")
sc <- spark_connect(master = "local")

import_iris <- copy_to(sc, iris, "spark_iris",
  overwrite = TRUE)

partition_iris <- sdf_partition(
  import_iris,training=0.5, testing=0.5)

sdf_register(partition_iris,
c("spark_iris_training","spark_iris_test"))

tidy_iris <- tbl(sc,"spark_iris_training") %>%
  select(Species, Petal_Length, Petal_Width)

model_iris <- tidy_iris %>%
  ml_decision_tree(response="Species",
  features=c("Petal_Length","Petal_Width"))

test_iris <- tbl(sc,"spark_iris_test")

pred_iris <- sdf_predict(
  model_iris, test_iris) %>%
  collect

pred_iris %>%
  inner_join(data.frame(prediction=0:2,
  lab=model_iris$model.parameters$labels)) %>%
  ggplot(aes(Petal_Length, Petal_Width, col=lab)) +
  geom_point()

spark_disconnect(sc)
```

Install Spark locally
Connect to local version
Copy data to Spark memory
Partition data
Create a hive metadata for each partition
Spark ML Decision Tree Model
Create reference to Spark table
Bring data back into R memory for plotting
Disconnect

Reactivity

COPY A DATA FRAME INTO SPARK

sdf_copy_to(sc, iris, "spark_iris")

sdf_copy_to(sc, x, name, memory, repartition, overwrite**)**

IMPORT INTO SPARK FROM A FILE

Arguments that apply to all functions:
sc, name, path, options = list(), repartition = 0, memory = TRUE, overwrite = TRUE

CSV **spark_read_csv(** header = TRUE, columns = NULL, infer_schema = TRUE, delimiter = ",", quote = "\"", escape = "\\", charset = "UTF-8", null_value = NULL**)**

JSON **spark_read_json()**

PARQUET **spark_read_parquet()**

SPARK SQL COMMANDS

DBI::**dbWriteTable**(sc, "spark_iris", iris)

DBI::**dbWriteTable(**conn, name, value**)**

FROM A TABLE IN HIVE

my_var <- **tbl_cache**(sc, name= "hive_iris")

tbl_cache(sc, name, force = TRUE**)**
Loads the table into memory

my_var <- **dplyr::tbl**(sc, name= "hive_iris")

dplyr::**tbl(**scr, …**)**
Creates a reference to the table without loading it into memory

Wrangle

SPARK SQL VIA DPLYR VERBS

Translates into Spark SQL statements

```
my_table <- my_var %>%
  filter(Species=="setosa") %>%
  sample_n(10)
```

DIRECT SPARK SQL COMMANDS

my_table <- DBI::**dbGetQuery**(sc , "SELECT * FROM iris LIMIT 10")

DBI::**dbGetQuery(**conn, statement**)**

SCALA API VIA SDF FUNCTIONS

sdf_mutate(.data)
Works like dplyr mutate function

sdf_partition(x, ..., weights = NULL, seed = sample (.Machine$integer.max, 1))
sdf_partition*(x, training = 0.5, test = 0.5)*

sdf_register(x, name = NULL)
Gives a Spark DataFrame a table name

sdf_sample(x, fraction = 1, replacement = TRUE, seed = NULL)

sdf_sort(x, columns)
Sorts by >=1 columns in ascending order

sdf_with_unique_id(x, id = "id")

sdf_predict(object, newdata)
Spark DataFrame with predicted values

ML TRANSFORMERS

ft_binarizer(my_table,input.col="Petal_Length", output.col="petal_large", threshold=1.2)

Arguments that apply to all functions:
x, input.col = NULL, output.col = NULL

ft_binarizer(threshold = 0.5)
Assigned values based on threshold

ft_bucketizer(splits)
Numeric column to discretized column

ft_discrete_cosine_transform(inverse = FALSE**)**
Time domain to frequency domain

ft_elementwise_product(scaling.col)
Element-wise product between 2 cols

ft_index_to_string()
Index labels back to label as strings

ft_one_hot_encoder()
Continuous to binary vectors

ft_quantile_discretizer(n.buckets=5L)
Continuous to binned categorical values

ft_sql_transformer(sql)

ft_string_indexer(params = NULL**)**
Column of labels into a column of label indices.

ft_vector_assembler()
Combine vectors into single row-vector

Visualize & Communicate

DOWNLOAD DATA TO R MEMORY

r_table <- **collect**(my_table)
plot(Petal_Width~Petal_Length, data=r_table)

dplyr::**collect**(x)
Download a Spark DataFrame to an R DataFrame

sdf_read_column(x, column)
Returns contents of a single column to R

SAVE FROM SPARK TO FILE SYSTEM

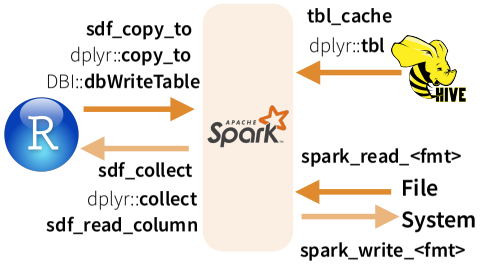
Arguments that apply to all functions: x, path

CSV **spark_read_csv(** header = TRUE, delimiter = ",", quote = "\"", escape = "\\", charset = "UTF-8", null_value = NULL**)**

JSON **spark_read_json(**mode = NULL**)**

PARQUET **spark_read_parquet(**mode = NULL**)**

Reading & Writing from Apache Spark



Extensions

Create an R package that calls the full Spark API & provide interfaces to Spark packages.

CORE TYPES

spark_connection() Connection between R and the Spark shell process

spark_jobj() Instance of a remote Spark object

spark_dataframe() Instance of a remote Spark DataFrame object

CALL SPARK FROM R

invoke() Call a method on a Java object

invoke_new() Create a new object by invoking a constructor

invoke_static() Call a static method on an object

MACHINE LEARNING EXTENSIONS

ml_create_dummy_variables()
ml_prepare_dataframe()
ml_prepare_response_features_intercept()
ml_options()
ml_model()

Model (MLlib)

ml_decision_tree(my_table, response = "Species", features = c("Petal_Length" , "Petal_Width"))

ml_als_factorization(x, user.column = "user", rating.column = "rating", item.column = "item", rank = 10L, regularization.parameter = 0.1, iter.max = 10L, ml.options = ml_options()**)**

ml_decision_tree(x, response, features, max.bins = 32L, max.depth = 5L, type = c("auto", "regression", "classification"), ml.options = ml_options()**)** Same options for: **ml_gradient_boosted_trees**

ml_generalized_linear_regression(x, response, features, intercept = TRUE, family = gaussian(link = "identity"), iter.max = 100L, ml.options = ml_options()**)**

ml_kmeans(x, centers, iter.max = 100, features = dplyr::tbl_vars(x), compute.cost = TRUE, tolerance = 1e-04, ml.options = ml_options()**)**

ml_lda(x, features = dplyr::tbl_vars(x), k = length(features), alpha = (50/k) + 1, beta = 0.1 + 1, ml.options = ml_options()**)**

ml_linear_regression(x, response, features, intercept = TRUE, alpha = 0, lambda = 0, iter.max = 100L, ml.options = ml_options()**)**
Same options for: **ml_logistic_regression**

ml_multilayer_perceptron(x, response, features, layers, iter.max = 100, seed = sample(.Machine$integer.max, 1), ml.options = ml_options()**)**

ml_naive_bayes(x, response, features, lambda = 0, ml.options = ml_options()**)**

ml_one_vs_rest(x, classifier, response, features, ml.options = ml_options()**)**

ml_pca(x, features = dplyr::tbl_vars(x), ml.options = ml_options()**)**

ml_random_forest(x, response, features, max.bins = 32L, max.depth = 5L, num.trees = 20L, type = c("auto", "regression", "classification"), ml.options = ml_options()**)**

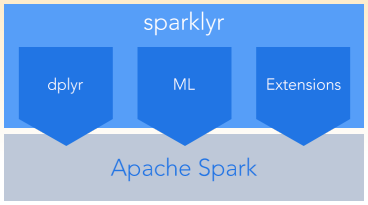
ml_survival_regression(x, response, features, intercept = TRUE,censor = "censor", iter.max = 100L, ml.options = ml_options()**)**

ml_binary_classification_eval(predicted_tbl_spark, label, score, metric = "areaUnderROC"**)**

ml_classification_eval(predicted_tbl_spark, label, predicted_lbl, metric = "f1"**)**

ml_tree_feature_importance(sc, model**)**

sparklyr is an R interface for Spark



The *eurostat* package

R tools to access open data from Eurostat database

Search and download

Data in the Eurostat database is stored in tables. Each table has an identifier, a short table_code, and a description (e.g. *tsdtr420* - People killed in road accidents).
Key eurostat functions allow to find the table_code, download the eurostat table and polish labels in the table.

Find the table code

The **search_eurostat(pattern, ...)** function scans the directory of Eurostat tables and returns codes and descriptions of tables that match pattern.

```
library("eurostat")
query <- search_eurostat("road", type = "table")
query[1:3,1:2]
##                                title     code
## 1          Goods transport by road ttr00005
## 2   People killed in road accidents tsdtr420
## 3 Enterprises with broadband access tin00090
```

Download the table

The **get_eurostat(id, time_format = "date", filters = "none", type = "code", cache = TRUE, ...)** function downloads the requested table from the *Eurostat bulk download facility* or from *The Eurostat Web Services JSON API* (if **filters** are defined). Downloaded data is cached (if **cache=TRUE**). Additional arguments define how to read the time column (**time_format**) and if table dimensions shall be kept as codes or converted to labels (**type**).

```
dat <- get_eurostat(id="tsdtr420", time_format="num")
head(dat)
##     unit    sex    geo  time values
## 1     NR      T     AT  1999   1079
## 2     NR      T     BE  1999   1397
## 3     NR      T     CZ  1999   1455
## 4     NR      T     DK  1999    514
## 5     NR      T     EL  1999   2116
## 6     NR      T     ES  1999   5738
```

Add labels

The **label_eurostat(x, lang = "en", ...)** gets definitions for Eurostat codes and replace them with labels in given language ("en", "fr" or "de").

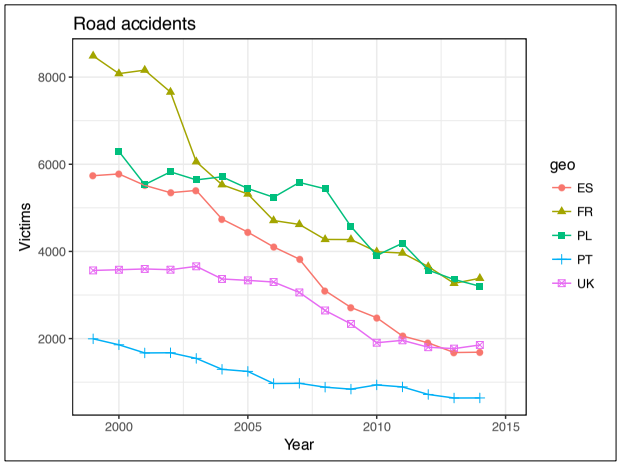
```
dat <- label_eurostat(dat)
head(dat)
##     unit    sex            geo  time values
## 1 Number  Total        Austria  1999   1079
## 2 Number  Total        Belgium  1999   1397
## 3 Number  Total Czech Republic  1999   1455
## 4 Number  Total        Denmark  1999    514
## 5 Number  Total         Greece  1999   2116
## 6 Number  Total          Spain  1999   5738
```

eurostat and plots

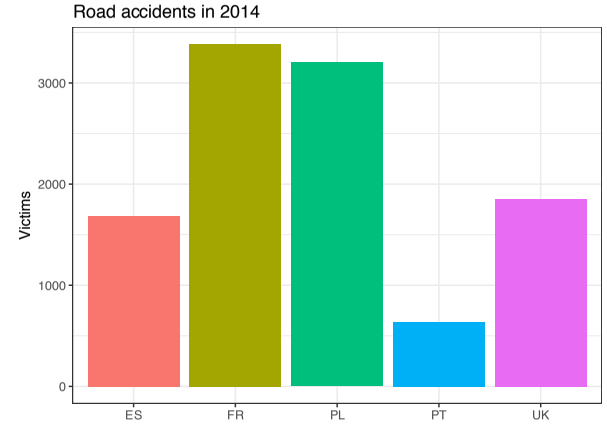
The **get_eurostat()** function returns tibbles in the long format. Packages **dplyr** and **tidyr** are well suited to transform these objects. The **ggplot2** package is well suited to plot these objects.

```
t1 <- get_eurostat("tsdtr420", filters =
            list(geo = c("UK", "FR", "PL", "ES", "PT")))

library("ggplot2")
ggplot(t1, aes(x = time, y = values, color = geo,
               group = geo, shape = geo)) +
  geom_point(size = 2) +
  geom_line() + theme_bw() +
  labs(title="Road accidents", x = "Year", y = "Victims")
```




```
library("dplyr")
t2 <- t1 %>% filter(time == "2014-01-01")
ggplot(t2, aes(geo, values, fill=geo)) +
  geom_bar(stat = "identity") + theme_bw() +
  theme(legend.position = "none")+
  labs(title="Road accidents in 2014", x="", y="Victims")
```



eurostat and maps

Fetch and process data

There are three function to work with geospatial data from GISCO. The **get_eurostat_geospatial()** returns preprocessed spatial data as sp-objects or as data frames. The **merge_eurostat_geospatial()** both downloads and merges the geospatial data with a preloaded tabular data. The **cut_to_classes()** is a wrapper for **cut()** - function and is used for categorizing data for maps with tidy labels.

```
library("eurostat")
library("dplyr")

fertility <- get_eurostat("demo_r_frate3")  %>%
  filter(time == "2014-01-01") %>%
  mutate(cat = cut_to_classes(values, n=7, decimals=1))

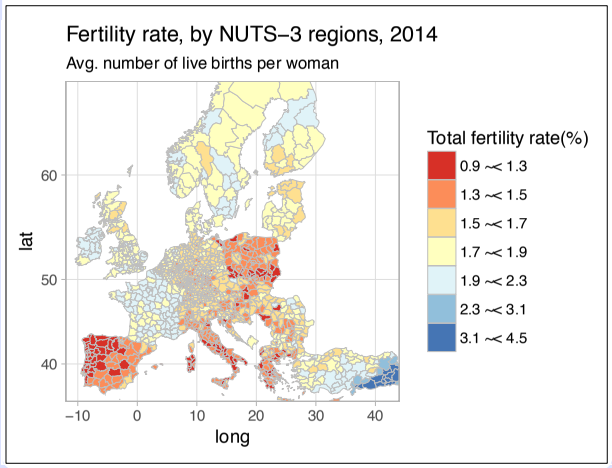
mapdata <-  merge_eurostat_geodata(fertility,
              resolution = "20")

head(select(mapdata,geo,values,cat,long,lat,order,id))
##     geo values         cat     long      lat order id
## 1 AT124   1.39 1.3 ~< 1.5 15.54245 48.90770   214 10
## 2 AT124   1.39 1.3 ~< 1.5 15.75363 48.85218   215 10
## 3 AT124   1.39 1.3 ~< 1.5 15.88763 48.78511   216 10
## 4 AT124   1.39 1.3 ~< 1.5 15.81535 48.69270   217 10
## 5 AT124   1.39 1.3 ~< 1.5 15.94094 48.67173   218 10
## 6 AT124   1.39 1.3 ~< 1.5 15.90833 48.59815   219 10
```

Draw a cartogram

The object returned by **merge_eurostat_geospatial()** are ready to be plotted with ggplot2 package. The **coord_map()** function is useful to set the projection while **labs()** adds annotations o the plot.

```
library("ggplot2")
ggplot(mapdata, aes(x = long, y = lat, group = group))+
  geom_polygon(aes(fill=cat), color="grey", size = .1)+
  scale_fill_brewer(palette = "RdYlBu") +
  labs(title="Fertility rate, by NUTS-3 regions, 2014",
       subtitle="Avg. number of live births per woman",
       fill="Total fertility rate(%)") + theme_light()+
  coord_map(xlim=c(-12,44), ylim=c(35,67))
```



This onepager presents the **eurostat** package
Leo Lahti, Janne Huovari, Markus Kainu, Przemyslaw Biecek 2014-2017 package version 2.2.43 URL: *https://github.com/rOpenGov/eurostat*

Data & Variable Transformation with sjmisc Cheat Sheet

sjmisc complements dplyr, and helps with data transformation tasks and recoding variables.

sjmisc works together seamlessly with dplyr and pipes. All functions are designed to support labelled data.

Design Philosophy

The design of sjmisc functions follows the tidyverse-approach: first argument is always the data (either a data frame **or** vector), followed by variable names to be processed by the functions.

The returned object for each function equals the type of the data-argument.

Vector input

- If the data-argument is a vector, functions return a vector.

```
rec(mtcars$carb, rec = "1,2=1; 3,4=2; else=3")
```

Data frame input

- If the data-argument is a data frame, functions return a data frame.

```
rec(mtcars, carb, rec = "1,2=1; 3,4=2; else=3")
```

The ...-ellipses Argument

Apply functions to a single variable, selected variables or to a complete data frame.

Variable selection is powered by dplyr's select(): Separate variables with comma, or use dplyr's select-helpers to select variables, e.g. ?rec:

```
rec(mtcars, one_of(c("gear", "carb")),
    rec = "min:3=1; 4:max=2")

rec(mtcars, gear, carb, rec = "min:3=1; 4:max=2")
```

Descriptives and Summaries

Most of the sjmisc functions (including recode-functions) also work on grouped data frames:

```
library(dplyr)
efc %>%
  group_by(e16sex, c172code) %>%
  frq(e42dep)
```

Frequency Tables

frq(x, ..., sort.frq = c("none", "asc", "desc"), weight.by = NULL)

Print frequency tables of (labelled) vectors. Uses variable labels as table header.

```
data(efc); frq(efc, e42dep, c161sex)
```

Use this data set in examples!

flat_table(data, ..., margin = c("counts", "cell", "row", "col"), digits = 2, show.values = FALSE)

Print contingency tables of (labelled) vectors. Uses value labels.

```
flat_table(efc, e42dep, c172code, e16sex)
```

count_na(x, ...)

Print frequency table of tagged NA values.

```
library(haven); x <- labelled(c(1:3,
tagged_na("a", "a", "z")), labels =
c("Refused" = tagged_na("a"), "N/A" =
tagged_na("z")))
count_na(x)
```

Descriptive Summary

descr(x, ..., max.length = NULL)

Descriptive summary of data frames, including variable labels in output.

```
descr(efc, contains("cop"), max.length = 20)
```

Finding Variables in a Data Frame

Use find_var() to search for variables by names, value or variable labels. Returns vector/data frame.

```
# variables with "cop" in names and variable labels
find_var(efc, pattern = "cop", out = "df")

# variables with "level" in names and value labels
find_var(efc, "level", search = "name_value")
```

Recode and Transform Variables

Recode functions add a suffix to new variables, so original variables are preserved.

By default, only the new created variables are returned. Use append = TRUE to return the original input data frame as well.

rec(x, ..., rec, as.num = TRUE, var.label = NULL, val.labels = NULL, append = FALSE, suffix = "_r")

Recode values, return result as numeric, character or categorical (factor).

```
rec(mtcars, carb, rec = "1,2=1; 3,4=2; else=3")
```

dicho(x, ..., dich.by = "median", as.num = FALSE, var.label = NULL, val.labels = NULL, append = FALSE, suffix = "_d")

Dichotomise variable by median, mean or specific value.

```
dicho(mtcars, disp)
```

split_var(x, ..., n, as.num = FALSE, val.labels = NULL, var.label = NULL, inclusive = FALSE, append = FALSE, suffix = "_g")

Split variable into equal sized groups. Unlike dplyr::ntile(), does not split original categories into different values (see examples in ?split_var).

```
split_var(mtcars, mpg, disp, n = 3)
```

group_var(x, ..., size = 5, as.num = TRUE, right.interval = FALSE, n = 30, append = FALSE, suffix = "_gr")

Split variable into groups with equal value range, or into a max. # of groups (value range per group is adjusted to match # of groups).

```
group_var(mtcars, mpg, disp, size = 5)
group_var(mtcars, mpg, size = "auto", n = 4)
```

std(x, ..., include.fac = TRUE, append = FALSE, suffix = "_z")

Z-standardise variables. Also center().

```
std(efc, e17age, c160age)
```

recode_to(x, ..., lowest = 0, highest = -1, append = FALSE, suffix = "_r0")

Shift ("renumber") categories or values.

```
recode_to(mtcars$gear)
```

Summarise Variables and Cases

The summary functions mostly mimic base R equivalents, but are designed to work together with pipes and dplyr.

row_sums(x, ..., na.rm = TRUE, var = "rowsums", append = FALSE)

Row sums of data frames.

```
row_sums(efc, c82cop1:c90cop9)
```

row_means(x, ..., n, var = "rowmeans", append = FALSE)

Row means, for at least n valid (non-NA) values.

```
row_means(efc, c82cop1:c90cop9, n = 7)
```

row_count(x, ..., count, var = "rowcount", append = FALSE)

Row-wise count # of values in data frames. Also col_count().

```
row_count(efc, c82cop1:c90cop9, count = 2)
```

Other Useful Functions

- add_columns() and replace_columns() to combine data frames, but either replace or preserve existing columns.
- set_na() and replace_na() to convert regular into missing values, or vice versa. replace_na() also replaces specific tagged NA values only.
- remove_var() and var_rename() to remove variables from data frames, or rename variables.
- group_str() to group similar string values. Useful for variables with similar, but not identically spelled string values that should be "merged".
- merge_df() to full join data frames and preserve value and variable labels.
- to_long() to gather multiple columns in data frames from wide into long format.

Use with %>% and dplyr

```
# use sjmisc-functions in pipes
mtcars %>% select(gear, carb) %>%
  rec(rec = "min:3=1; 4:max=2")

# use sjmisc-function inside mutate
mtcars %>% select(gear, carb) %>% mutate(
  carb2 = rec(carb, rec = "1,2=0;3:8=1"),
  gear2 = rec(gear, rec = "3=1;4:max=2"))
```

Leaflet Cheat Sheet

Leaflet for R

an open-source JavaScript library for mobile-friendly interactive maps

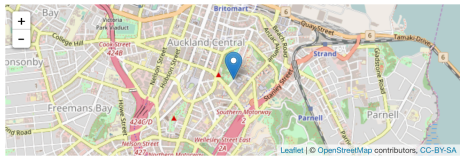
Quick Start

Installation

Use **install.packages("leaflet")** to installize the package or directly from Github *devtools::install_github("rstudio/leaflet")*.

First Map

```
m <- leaflet() %>%          # leaflet works with the pipe operator
  addTiles() %>%            # setup the default OpenStreetMap map tiles
  addMarkers(lng = 174.768, lat = -36.852, popup = "The birthplace of R")
                            # add a single point layer
m
```



Map Widget

Initialization

m <- leaflet(options = leafletOptions(...))

center	Initial geographic center of the map
zoom	Initial map zoom level
minZoom	Minimum zoom level of the map
maxZoom	Maximum zoom level of the map

Map Methods

m %>% setView(lng, lat, zoom, options = list())
Set the view of the map (center and zoom level)

m %>% fitBounds(lng1, lat1, lng2, lat2)
Fit the view into the rectangle [lng1, lat1] - [lng2, lat2]

m %>% clearBounds()
Clear the bound, automatically determine from the map elements

Data Object

Both **leaflet()** and the **map** layers have an optional data parameter that is designed to receive spatial data with the following formats:

Base R — *The arguments of all layers take normal R objects:*

```
df <- data.frame(lat = ..., lng = ... )
leaflet(df) %>% addTiles() %>% addCircles()
```

sp *package* — library(sp) *Useful functions:*
SpatialPoints, SpatialLines, SpatialPolygons, ...

maps *package* — library(maps) *Build a map of states with colors:*

```
mapStates <- map("state", fill = TRUE, plot = FALSE)
leaflet(mapStates) %>% addTiles() %>%
  addPolygons(fillColor = topo.colors(10, alpha =
  NULL), stroke = FALSE)
```

Markers

Use markers to call out points, express locations with latitude/longitude coordinates, appear as icons or as circles.
Data come from vectors or assigned data frame, or **sp** package objects.

Icon Markers

Regular Icons: default and simple

addMarkers(lng, lat, popup, label) *add basic icon markers*

makeIcon/icons (iconUrl, iconWidth, iconHeight, iconAnchorX, iconAnchorY, shadowUrl, shadowWidth, shadowHeight, ...) *customize marker icons*

iconList() *create a list of icons*

Awesome Icons: customizable with colors and icons

addAwesomeMarkers, makeAwesomeIcon, awesomeIcons, awesomeIconList

Marker Clusters: option of addMarters()

clusterOptions = markerClusterOptions()

freezeAtZoom Freeze the cluster at assigned zoom level

Circle Markers

addCircleMarkers(color, radius, stroke, opacity, ...)
Customize their color, radius, stroke, opacity

Popups and Labels

addPopups(lng, lat, ...content..., options) *Add standalone popups*
options = popupOptions(closeButton=FALSE)

addMarkers(..., popup, ...) *Show popups with markers or shapes*

addMarkers(..., label, labelOptions...) *Show labels with markers or shapes*
labelOptions = labelOptions(noHide, textOnly, textsize, direction, style)

addLabelOnlyMarkers() *Add labels without markers*

Lines and Shapes

Polygons and Polylines

addPolygons(color, weight=1, smoothFactor=0.5, opacity=1.0, fillOpacity=0.5, fillColor= ~colorQuantile("YlOrRd", ALAND)(ALAND), highlightOptions, ...)

highlightOptions(color, weight=2, bringToFront=TRUE) *highlight shapes*

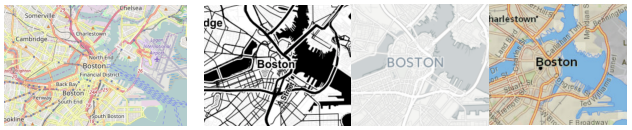
Use **rmapshaper::ms_simplify** to simplify complex shapes

Circles **addCircles**(lng, lat, weight=1, radius, ...)

Rectangles **addRectangles**(lng1, lat1, lng2, lat2, fillColor="transparent", ...)

Basemaps

addTiles() providers$Stamen.Toner, CartoDB.Positron, Esri.NatGeoWorldMap



Default Tiles — Third-Party Tiles **addProviderTiles()**

Use **addTiles()** to add a custom map tile URL template, use **addWMSTiles()** to add WMS (Web Map Service) tiles

GeoJSON and TopoJSON

There are two options to use the GeoJSON/TopoJSON data.

* To read into **sp** objects with the **geojsonio** or **rgdal** package:
geojsonio::geojson_read(...,what="sp") *rgdal::readOGR(...,"OGRGeoJSON")*
* Or to use the **addGeoJSON()** and **addTopoJSON()** functions:
addTopoJSON/addGeoJSON(... weight, color, fill, opacity, fillOpacity...)
Styles can also be tuned separately with a *style: {...}* object.

Other packages including **RJSONIO** and **jsonlite** can help fast parse or generate the data needed.

Shiny Integration

To integrate a Leaflet map into an app:
* In the UI, call **leafletOutput**("name")
* On the server side, assign a **renderLeaflet**(...) call to the output
* Inside the **renderLeaflet** expression, return a Leaflet map object

Modification

To modify an existing map or add incremental changes to the map, you can use **leafletProxy()**. This should be performed in an observer on the server side.

Other useful functions to edit your map:

fitBounds(0, 0, 11, 11) *similar to setView*
fit the view to within these bounds

addCircles(1:10, 1:10, layerId = LETTERS[1:10])
create circles with layerIds of "A", "B", "C"...

removeShape(c("B", "F")) *remove some of the circles*

clearShapes() *clear all circles (and other shapes)*

Inputs/Events

Object Events

Object event names generally use this pattern:
input$*MAPID_OBJCATEGORY_EVENTNAME*.
Triger an event changes the value of the Shiny input at this variable.
Valid values for *OBJCATEGORY* are *marker*, *shape*, *geojson* and *topojson*.
Valid values for *EVENTNAME* are *click*, *mouseover* and *mouseout*.

All of these events are set to either *NULL* if the event has never happened, or a *list()* that includes:
* lat The latitude of the object, if available; otherwise, the mouse cursor
* lng The longitude of the object, if available; otherwise, the mouse cursor
* id The layerId, if any

GeoJSON events also include additional properties:
* featureId The feature ID, if any
* properties The feature properties

Map Events

input$*MAPID*_click *when the map background or basemap is clicked*
value -- a list with lat and lng

input$*MAPID*_bounds *provide the lat/lng bounds of the visible map area*
value -- a list with north, east, south and west

input$*MAPID*_zoom *an integer indicates the zoom level*

Kejia Shi @ Data Science Institute, Columbia University in the City of New York, Kejia.Shi@Columbia.edu

Adapted from Rstudio materials https://rstudio.github.io/leaflet/ 2017/05/01

Creating Survival Plots

Informative and Elegant with *survminer*

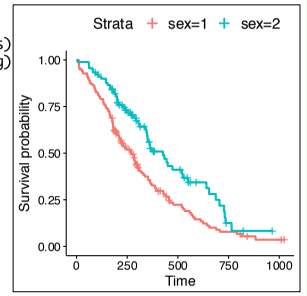
Survival Curves

The **ggsurvplot()** function creates **ggplot2** plots from **survfit** objects.

```
library("survival")
fit <- survfit(Surv(time,status)
               ~ sex, data = lung)

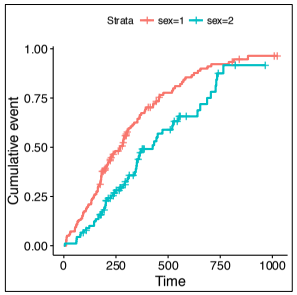
class(fit)
## [1] "survfit"

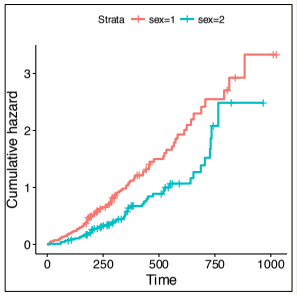
library("survminer")
ggsurvplot(fit, data = lung)
```



Use the **fun** argument to set the transformation of the survival curve. E.g. **"event"** for cumulative events, **"cumhaz"** for the cumulative hazard function or **"pct"** for survival probability in percentage.

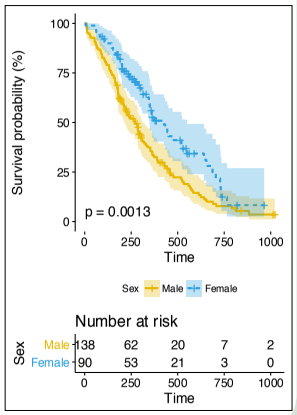
```
ggsurvplot(fit, data = lung, fun = "event")
ggsurvplot(fit, data = lung, fun = "cumhaz")
```





With lots of graphical parameters you have full control over look and feel of the survival plots; position and content of the legend; additional annotations like p-value, title, subtitle.

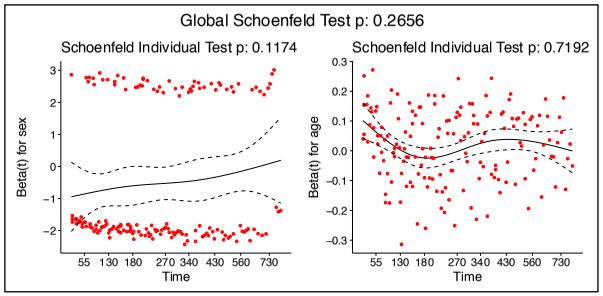
```
ggsurvplot(fit, data = lung,
     conf.int = TRUE,
     pval = TRUE,
     fun = "pct",
     risk.table = TRUE,
     size = 1,
     linetype = "strata",
     palette = c("#E7B800",
                 "#2E9FDF"),
     legend = "bottom",
     legend.title = "Sex",
     legend.labs = c("Male",
                     "Female"))
```



Diagnostics of Cox Model

The function **cox.zph()** from **survival** package may be used to test the proportional hazards assumption for a Cox regression model fit. The graphical verification of this assumption may be performed with the function **ggcoxzph()** from the **survminer** package. For each covariate it produces plots with scaled Schoenfeld residuals against the time.

```
library("survival")
fit <- coxph(Surv(time, status) ~ sex + age, data = lung)
ftest <- cox.zph(fit)
ftest
##            rho chisq     p
## sex     0.1236 2.452 0.117
## age    -0.0275 0.129 0.719
## GLOBAL      NA 2.651 0.266
library("survminer")
ggcoxzph(ftest)
```

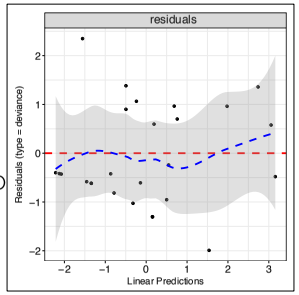


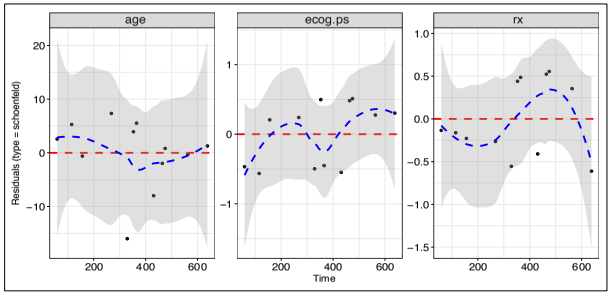
The function **ggcoxdiagnostics()** plots different types of residuals as a function of time, linear predictor or observation id. The type of residual is selected with **type** argument. Possible values are "martingale", "deviance", "score", "schoenfeld", "dfbeta", "dfbetas", and "scaledsch".
The **ox.scale** argument defines what shall be plotted on the OX axis. Possible values are "linear.predictions", "observation.id", "time".
Logical arguments **hline** and **sline** may be used to add horizontal line or smooth line to the plot.

```
library("survival")
library("survminer")
fit <- coxph(Surv(time, status) ~
             sex + age, data = lung)

ggcoxdiagnostics(fit,
     type = "deviance",
     ox.scale = "linear.predictions")

ggcoxdiagnostics(fit,
     type = "schoenfeld",
     ox.scale = "time")
```





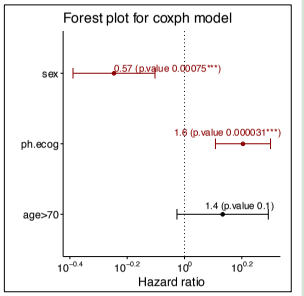
Summary of Cox Model

The function **ggforest()** from the **survminer** package creates a forest plot for a Cox regression model fit. Hazard ratio estimates along with confidence intervals and p-values are plotter for each variable.

```
library("survival")
library("survminer")
lung$age <- ifelse(lung$age > 70, ">70","<= 70")
fit <- coxph( Surv(time, status) ~ sex + ph.ecog + age, data = lung)
fit

## Call:
## coxph(formula = Surv(time, status) ~ sex+ph.ecog+age, data=lung)
##
##            coef exp(coef) se(coef)     z       p
## sex     -0.567     0.567    0.168 -3.37 0.00075
## ph.ecog  0.470     1.600    0.113  4.16 3.1e-05
## age>70   0.307     1.359    0.187  1.64 0.10175
##
## Likelihood ratio test=31.6  on
## n= 227, number of events= 164

ggforest(fit)
```



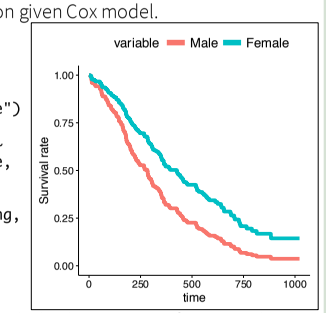
The function **ggcoxadjustedcurves()** from the **survminer** package plots Adjusted Survival Curves for Cox Proportional Hazards Model. Adjusted Survival Curves show how a selected factor influences survival estimated from a Cox model.
Note that these curves differ from Kaplan Meier estimates since they present expected ssurvival based on given Cox model.

```
library("survival")
library("survminer")

lung$sex <- ifelse(lung$sex == 1,
                   "Male", "Female")

fit <- coxph(Surv(time, status) ~
             sex + ph.ecog + age,
             data = lung)

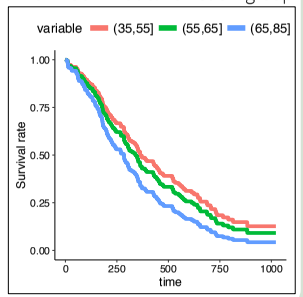
ggcoxadjustedcurves(fit, data=lung,
             variable=lung$sex)
```



Note that it is not necessary to include the grouping factor in the Cox model. Survival curves are estimated from Cox model for each group defined by the factor independently.

```
lung$age3 <- cut(lung$age,
                 c(35,55,65,85))

ggcoxadjustedcurves(fit, data=lung,
             variable=lung$age3)
```



This onepager presents the **survminer** package [Alboukadel Kassambara, Marcin Kosinski 2017] in version 0.3.
See *https://github.com/kassambara/survminer/* for more details.

quanteda Cheat Sheet

quanteda: Quantitative Analysis of Textual Data

General syntax

- **corpus_*** manage text collections/metadata
- **tokens_*** create/modify tokenized texts
- **dfm_*** create/modify doc-feature matrices
- **fcm_*** work with co-occurrence matrices
- **textstat_*** calculate text-based statistics
- **textmodel_*** fit (un-)supervised models
- **textplot_*** create text-based visualizations

Consistent grammar:

- **object()** constructor for the object type
- **object_verb()** inputs & returns object type

Extensions

quanteda works well with these companion packages:

- **readtext**: An easy way to read text data
- **spacyr**: NLP using the spaCy library
- **quantedaData**: additional textual data
- **LIWCalike**: R implementation of the Linguistic Inquiry and Word Count approach

Create a corpus from texts (corpus_*)

Read texts (txt, pdf, csv, doc, docx, json, xml)

```
my_texts <- readtext::readtext("~/link/to/path/*")
```

Construct a corpus from a character vector

```
x <- corpus(data_char_ukimmig2010, text_field = "text")
```

Explore a corpus

```
summary(data_corpus_inaugural, n = 2)
# Corpus consisting of 58 documents, showing 2 documents:
#            Text Types Tokens Sentences Year  President FirstName
# 1789-Washington   625   1538        23 1789 Washington    George
# 1793-Washington    96    147         4 1793 Washington    George
#
# Source:  Gerhard Peters and John T. Woolley. The American Presidency Project.
# Created: Tue Jun 13 14:51:47 2017
# Notes:   http://www.presidency.ucsb.edu/inaugurals.php
```

Extract or add document-level variables

```
party <- docvars(data_corpus_inaugural, "Party")
docvars(x, "serial_number") <- 1:ndoc(x)
```

Bind or subset corpora

```
corpus(x[1:5]) + corpus(x[7:9])
corpus_subset(x, Year > 1990)
```

Change units of a corpus

```
corpus_reshape(x, to = c("sentences", use_docvars = TRUE)
```

Segment texts on a pattern match

```
corpus_segment(x, pattern, valuetype, extract_pattern = TRUE)
```

Take a random sample of corpus texts

```
corpus_sample(x, size = 10, replace = FALSE)
```

Extract features (dfm_*; fcm_*)

Create a document-feature matrix (dfm) from a corpus

```
x <- dfm(data_corpus_inaugural,
         tolower = TRUE, stem = FALSE, remove_punct = TRUE,
         remove = stopwords("english"))

head(x, n = 2, nfeature = 4)
## Document-feature matrix of: 2 documents, 4 features (41.7% sparse).
##                  features
## docs              fellow-citizens senate house representatives
##   1789-Washington               1      1     2               2
##   1793-Washington               0      0     0               0
```

Create a dictionary

```
dictionary(list(negative = c("bad", "awful", "sad"),
                positive = c("good", "wonderful", "happy")))
```

Apply a dictionary

```
dfm_lookup(x, dictionary = data_dictionary_LSD2015)
```

Select features

```
dfm_select(x, dictionary = data_dictionary_LSD2015)
```

Compress a dfm by combining identical elements

```
dfm_compress(x, margin = c("both", "documents", "features"))
```

Randomly sample documents or features

```
dfm_sample(x, what = c("documents", "features"))
```

Weight or smooth the feature frequencies

```
dfm_weight(x, type = "relfreq") | dfm_smooth(x, smoothing = 0.5)
```

Sort or group a dfm

```
dfm_sort(x, margin = c("features", "documents", "both"))
dfm_group(x, groups = "President")
```

Combine identical dimension elements of a dfm

```
dfm_compress(x, margin = c("both", "documents", "features"))
```

Create a feature co-occurrence matrix (fcm)

```
x <- fcm(data_corpus_inaugural, context = "window", size = 5)
```

fcm_compress/remove/select/toupper/tolower are also available

Useful additional functions

Locate keywords-in-context

```
kwic(data_corpus_inaugural, "america")
```

Utility functions

texts(corpus)	*Show texts of a corpus*
ndoc(corpus/dfm/tokens)	*Count documents/features*
nfeature(corpus/dfm/tokens)	*Count features*
summary(corpus/dfm)	*Print summary*
head(corpus/dfm)	*Return first part*
tail(corpus/dfm)	*Return last part*

Tokenize a set of texts (tokens_*)

Tokenize texts from a character vector or corpus

```
x <- tokens("Powerful tool for text analysis.",
            remove_punct = TRUE, stem = TRUE)
```

Convert sequences into compound tokens

```
myseqs <- phrase(c("powerful", "tool", "text analysis"))
tokens_compound(x, myseqs)
```

Select tokens

```
tokens_select(x, c("powerful", "text"), selection = "keep")
```

Create ngrams and skipgrams from tokens

```
tokens_ngrams(x, n = 1:3)
tokens_skipgrams(toks, n = 2, skip = 0:1)
```

Convert case of tokens

```
tokens_tolower(x) | tokens_topupper(x)
```

Stem the terms in an object

```
tokens_wordstem(x)
```

Calculate text statistics (textstat_*)

Tabulate feature frequencies from a dfm

```
textstat_frequency(x) | topfeatures(x)
```

Identify and score collocations from a tokenized text

```
toks <- tokens(c("quanteda is a pkg for quant text analysis",
                 "quant text analysis is a growing field"))
textstat_collocations(toks, size = 3, min_count = 2)
```

Calculate readability of a corpus

```
textstat_readability(data_corpus_inaugural, measure = "Flesch")
```

Calculate lexical diversity of a dfm

```
textstat_lexdiv(x, measure = "TTR")
```

Measure distance or similarity from a dfm

```
textstat_simil(x, "2017-Trump", method = "cosine")
textstat_dist(x, "2017-Trump", margin = "features")
```

Calculate keyness statistics

```
textstat_keyness(x, target = "2017-Trump")
```

Fit text models based on a dfm (textmodel_*)

Correspondence Analysis (CA)

```
textmodel_ca(x, threads = 2, sparse = TRUE, residual_floor = 0.1)
```

Naïve Bayes classifier for texts

```
textmodel_NB(x, y = training_labels, distribution = "multinomial")
```

Wordscores text model

```
refscores <- c(seq(-1.5, 1.5, .75), NA))
textmodel_wordscores(data_dfm_lbgexample, refscores)
```

Wordfish Poisson scaling model

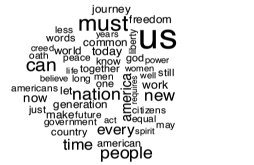
```
textmodel_wordfish(dfm(data_corpus_irishbudget2010), dir = c(6,5))
```

Textmodel methods: `predict()`, `coef()`, `summary()`, `print()`

Plot features or models (textplot_*)

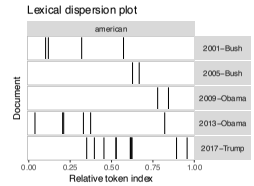
Plot features as a wordcloud

```
data_corpus_inaugural %>%
  corpus_subset(President == "Obama") %>%
  dfm(remove = stopwords("english")) %>%
  textplot_wordcloud()
```



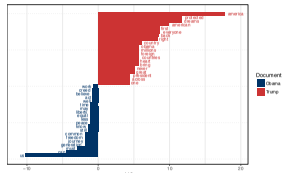
Plot the dispersion of key word(s)

```
data_corpus_inaugural %>%
  corpus_subset(Year > 1945) %>%
  kwic("american") %>%
  textplot_xray()
```



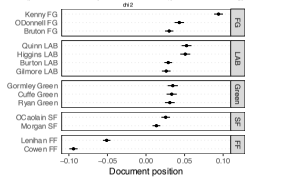
Plot word keyness

```
data_corpus_inaugural %>%
  corpus_subset(President %in%
                  c("Obama", "Trump")) %>%
  dfm(groups = "President",
      remove = stopwords("english")) %>%
  textstat_keyness(target = "Trump") %>%
  textplot_keyness()
```



Plot Wordfish, Wordscores or CA models

```
textplot_scale1d(scaling_model,
  groups = party,
  margin = c("documents"))
```



Convert dfm to a non-quanteda format

```
convert(x, to = c("lda", "tm", "stm", "austin", "topicmodels",
                  "lsa", "matrix", "data.frame"))
```


by **Stefan Müller** and **Kenneth Benoit** • *mullers@tcd.ie*, *kbenoit@lse.ac.uk*

Learn more at: http://quanteda.io • updated: 09/17

R For Data Science *Cheat Sheet*

data.table

Learn R for data science Interactively at www.DataCamp.com

data.table

data.table is an R package that provides a high-performance version of base R's `data.frame` with syntax and feature enhancements for ease of use, convenience and programming speed.

Load the package:

```
> library(data.table)
```

Creating A `data.table`

```
> set.seed(45L)
> DT <- data.table(V1=c(1L,2L),
                   V2=LETTERS[1:3],
                   V3=round(rnorm(4),4),
                   V4=1:12)
```

Create a `data.table` and call it `DT`

Subsetting Rows Using `i`

```
> DT[3:5,]
> DT[3:5]
> DT[V2=="A"]
> DT[V2 %in% c("A","C")]
```

Select 3rd to 5th row
Select 3rd to 5th row
Select all rows that have value `A` in column `V2`
Select all rows that have value `A` or `C` in column `V2`

Manipulating on Columns in `j`

```
> DT[,V2]
 [1] "A" "B" "C" "A" "B" "C" ...
> DT[,.(V2,V3)]
> DT[,sum(V1)]
 [1] 18
> DT[,.(sum(V1),sd(V3))]
   V1        V2
1: 18 0.4546055
> DT[,.(Aggregate=sum(V1),
        Sd.V3=sd(V3))]
   Aggregate     Sd.V3
1:        18 0.4546055
> DT[,.(V1,Sd.V3=sd(V3))]

> DT[,.(print(V2),
        plot(V3),
        NULL)]
```

Return `V2` as a vector

Return `V2` and `V3` as a `data.table`
Return the sum of all elements of `V1` in a vector
Return the sum of all elements of `V1` and the std. dev. of `V3` in a `data.table`

The same as the above, with new names

Select column `V2` and compute std. dev. of `V3`, which returns a single value and gets recycled
Print column `V2` and plot `V3`

Doing `j` by Group

```
> DT[,.(V4.Sum=sum(V4)),by=V1]
   V1 V4.Sum
1:  1     36
2:  2     42
> DT[,.(V4.Sum=sum(V4)),
        by=.(V1,V2)]
> DT[,.(V4.Sum=sum(V4)),
        by=sign(V1-1)]
   sign V4.Sum
1:    0     36
2:    1     42
> DT[,.(V4.Sum=sum(V4)),
        by=.(V1.01=sign(V1-1))]
> DT[1:5,.(V4.Sum=sum(V4)),
          by=V1]
> DT[,.N,by=V1]
```

Calculate sum of `V4` for every group in `V1`

Calculate sum of `V4` for every group in `V1` and `V2`
Calculate sum of `V4` for every group in `sign(V1-1)`

The same as the above, with new name for the variable you're grouping by
Calculate sum of `V4` for every group in `V1` after subsetting on the first 5 rows
Count number of rows for every group in `V1`

General form: `DT[i, j, by]`

"Take `DT`, subset rows using `i`, then calculate `j` grouped by `by`"

Adding/Updating Columns By Reference in `j` Using `:=`

```
> DT[,V1:=round(exp(V1),2)]
> DT
     V1 V2      V3 V4
1: 2.72  A -0.1107  1
2: 7.39  B -0.1427  2
3: 2.72  C -1.8893  3
4: 7.39  A -0.3571  4
...
> DT[,c("V1","V2"):=list(round(exp(V1),2),
                         LETTERS[4:6])]
> DT[,':='(V1=round(exp(V1),2),
           V2=LETTERS[4:6])][]
        V1 V2      V3 V4
1:   15.18  D -0.1107  1
2: 1619.71  E -0.1427  2
3:   15.18  F -1.8893  3
4: 1619.71  D -0.3571  4
> DT[,V1:=NULL]
> DT[,c("V1","V2"):=NULL]
> Cols.chosen=c("A","B")
> DT[,Cols.Chosen:=NULL]

> DT[,(Cols.Chosen):=NULL]
```

`V1` is updated by what is after `:=`
Return the result by calling `DT`

Columns `V1` and `V2` are updated by what is after `:=`
Alternative to the above one. With `[]`, you print the result to the screen

Remove `V1`
Remove columns `V1` and `V2`

Delete the column with column name `Cols.chosen`
Delete the columns specified in the variable `Cols.chosen`

Indexing And Keys

```
> setkey(DT,V2)
> DT["A"]
   V1 V2      V3 V4
1:  1  A -0.2392  1
2:  2  A -1.6148  4
3:  1  A  1.0498  7
4:  2  A  0.3262 10
> DT[c("A","C")]
> DT["A",mult="first"]

> DT["A",mult="last"]

> DT[c("A","D")]
   V1 V2      V3 V4
1:  1  A -0.2392  1
2:  2  A -1.6148  4
3:  1  A  1.0498  7
4:  2  A  0.3262 10
5: NA  D      NA NA
> DT[c("A","D"),nomatch=0]
   V1 V2      V3 V4
1:  1  A -0.2392  1
2:  2  A -1.6148  4
3:  1  A  1.0498  7
4:  2  A  0.3262 10
> DT[c("A","C"),sum(V4)]

> DT[c("A","C"),
     sum(V4),
     by=.EACHI]
   V2 V1
1:  A 22
2:  C 30
> setkey(DT,V1,V2)
> DT[.(2,"C")]
   V1 V2      V3 V4
1:  2  C  0.3262  6
2:  2  C -1.6148 12
> DT[.(2,c("A","C"))]
   V1 V2      V3 V4
1:  2  A -1.6148  4
2:  2  A  0.3262 10
3:  2  C  0.3262  6
4:  2  C -1.6148 12
```

A key is set on `V2`; output is returned invisibly
Return all rows where the key column (set to `V2`) has the value `A`

Return all rows where the key column (`V2`) has value `A` or `C`
Return first row of all rows that match value `A` in key column `V2`

Return last row of all rows that match value `A` in key column `V2`

Return all rows where key column `V2` has value `A` or `D`

Return all rows where key column `V2` has value `A` or `D`

Return total sum of `V4`, for rows of key column `V2` that have values `A` or `C`
Return sum of column `V4` for rows of `V2` that have value `A`, and anohter sum for rows of `V2` that have value `C`

Sort by `V1` and then by `V2` within each group of `V1` (invisible)
Select rows that have value 2 for the first key (`V1`) and the value `C` for the second key (`V2`)

Select rows that have value 2 for the first key (`V1`) and within those rows the value `A` or `C` for the second key (`V2`)

Advanced Data Table Operations

```
> DT[.N-1]
> DT[,.N]
> DT[,.(V2,V3)]
> DT[,list(V2,V3)]
> DT[,mean(V3),by=.(V1,V2)]
   V1 V2      V1
1:  1  A  0.4053
2:  1  B  0.4053
3:  1  C  0.4053
4:  2  A -0.6443
5:  2  B -0.6443
6:  2  C -0.6443
```

Return the penultimate row of the `DT`
Return the number of rows
Return `V2` and `V3` as a `data.table`
Return `V2` and `V3` as a `data.table`
Return the result of `j`, grouped by all possible combinations of groups specified in `by`

`.SD` & `.SDcols`

```
> DT[,print(.SD),by=V2]
> DT[,.SD[c(1,.N)],by=V2]
> DT[,lapply(.SD,sum),by=V2]

> DT[,lapply(.SD,sum),by=V2,
     .SDcols=c("V3","V4")]
   V2     V3 V4
1:  A -0.478 22
2:  B -0.478 26
3:  C -0.478 30
> DT[,lapply(.SD,sum),by=V2,
     .SDcols=paste0("V",3:4)]
```

Look at what `.SD` contains
Select the first and last row grouped by `V2`
Calculate sum of columns in `.SD` grouped by `V2`

Calculate sum of `V3` and `V4` in `.SD` grouped by `V2`

Calculate sum of `V3` and `V4` in `.SD` grouped by `V2`

Chaining

```
> DT <- DT[,.(V4.Sum=sum(V4)),
           by=V1]
   V1 V4.Sum
1:  1     36
2:  2     42
> DT[V4.Sum>40]
> DT[,.(V4.Sum=sum(V4)),
     by=V1][V4.Sum>40]
   V1 V4.Sum
1:  2     42
> DT[,.(V4.Sum=sum(V4)),
     by=V1][order(-V1)]
   V1 V4.Sum
1:  2     42
2:  1     36
```

Calculate sum of `V4`, grouped by `V1`

Select that group of which the sum is >40
Select that group of which the sum is >40 (chaining)

Calculate sum of `V4`, grouped by `V1`, ordered on `V1`

`set()`-Family

`set()`

Syntax: `for (i in from:to) set(DT, row, column, new value)`

```
> rows <- list(3:4,5:6)
> cols <- 1:2
> for(i in seq_along(rows))
    {set(DT,
         i=rows[[i]],
         j=cols[i],
         value=NA)}
```

Sequence along the values of `rows`, and for the values of `cols`, set the values of those elements equal to `NA` (invisible)

`setnames()`

Syntax: `setnames(DT,"old","new")[]`

```
> setnames(DT,"V2","Rating")
> setnames(DT,
           c("V2","V3"),
           c("V2.rating","V3.DC"))
```

Set name of `V2` to `Rating` (invisible)
Change 2 column names (invisible)

`setnames()`

Syntax: `setcolorder(DT,"neworder")`

```
> setcolorder(DT,
              c("V2","V1","V4","V3"))
```

Change column ordering to contents of the specified vector (invisible)

R For Data Science *Cheat Sheet*

xts

Learn R for data science Interactively at www.DataCamp.com

xts

eXtensible Time Series (xts) is a powerful package that provides an extensible time series class, enabling uniform handling of many R time series classes by extending zoo.

Load the package as follows:

```
> library(xts)
```

xts Objects

xts objects have three main components:

- **coredata**: always a matrix for xts objects, while it could also be a vector for zoo objects
- **index**: vector of any `Date`, `POSIXct`, `chron`, `yearmon`, `yearqtr`, or `DateTime` classes
- **xtsAttributes**: arbitrary attributes

Creating xts Objects

```
> xts1 <- xts(x=1:10, order.by=Sys.Date()-1:10)
> data <- rnorm(5)
> dates <- seq(as.Date("2017-05-01"),length=5,by="days")
> xts2 <- xts(x=data, order.by=dates)
> xts3 <- xts(x=rnorm(10),
              order.by=as.POSIXct(Sys.Date()+1:10),
              born=as.POSIXct("1899-05-08"))
> xts4 <- xts(x=1:10, order.by=Sys.Date()+1:10)
```

Convert To And From xts

```
> data(AirPassengers)
> xts5 <- as.xts(AirPassengers)
```

Import From Files

```
> dat <- read.csv(tmp_file)
> xts(dat, order.by=as.Date(rownames(dat),"%m/%d/%Y"))
> dat_zoo <- read.zoo(tmp_file,
                      index.column=0,
                      sep=",",
                      format="%m/%d/%Y")
> dat_zoo <- read.zoo(tmp,sep=",",FUN=as.yearmon)
> dat_xts <- as.xts(dat_zoo)
```

Export xts Objects

```
> data_xts <- as.xts(matrix)
> tmp <- tempfile()
> write.zoo(data_xts,sep=",",file=tmp)
```

Replace & Update

```
> xts2[dates] <- 0
> xts5["1961"] <- NA
> xts2["2016-05-02"] <- NA
```

Replace values in `xts2` on `dates` with `0`
Replace dates from `1961` with `NA`
Replace the value at 1 specific index with `NA`

Applying Functions

```
> ep1 <- endpoints(xts4,on="weeks",k=2)
[1]  0  5 10
> ep2 <- endpoints(xts5,on="years")
[1]   0  12  24  36  48  60  72  84  96 108 120 132 144
> period.apply(xts5,INDEX=ep2,FUN=mean)
> xts5_yearly <- split(xts5,f="years")
> lapply(xts5_yearly,FUN=mean)
> do.call(rbind,
          lapply(split(xts5,"years"),
          function(w) last(w,n="1 month")))
> do.call(rbind,
          lapply(split(xts5,"years"),
          cumsum))
> rollapply(xts5, 3, sd)
```

Take index values by time
Calculate the yearly mean
Split `xts5` by year
Create a list of yearly means
Find the last observation in each year in `xts5`
Calculate cumulative annual passengers
Apply sd to rolling margins of `xts5`

Selecting, Subsetting & Indexing

Select

```
> mar55 <- xts5["1955-03"]
```

Get value for March 1955

Subset

```
> xts5_1954 <- xts5["1954"]
> xts5_janmarch <- xts5["1954/1954-03"]
> xts5_janmarch <- xts5["/1954-03"]
> xts4[ep1]
```

Get all data from 1954
Extract data from Jan to March '54
Get all data until March '54
Subset `xts4` using `ep2`

`first()` and `last()`

```
> first(xts4,'1 week')
> first(last(xts4,'1 week'),'3 days')
```

Extract first 1 week
Get first 3 days of the last week of data

Indexing

```
> xts2[index(xts3)]
> days <- c("2017-05-03","2017-05-23")
> xts3[days]
> xts2[as.POSIXct(days,tz="UTC")]
> index <- which(.indexwday(xts1)==0|.indexwday(xts1)==6)
> xts1[index]
```

Extract rows with the index of `xts3`
Extract rows using the vector `days`
Extract rows using `days` as POSIXct
Index of weekend days
Extract weekend days of `xts1`

Missing Values

```
> na.omit(xts5)
> xts_last <- na.locf(xts2)
> xts_last <- na.locf(xts2,
                      fromLast=TRUE)
> na.approx(xts2)
```

Omit NA values in `xts5`
Fill missing values in `xts2` using last observation
Fill missing values in `xts2` using next observation
Interpolate NAs using linear approximation

Arithmetic Operations

`coredata()` or `as.numeric()`

```
> xts3 + as.numeric(xts2)
> xts3 * as.numeric(xts4)
> coredata(xts4) - xts3
> coredata(xts4) / xts3
```

Addition
Multiplication
Subtraction
Division

Shifting Index Values

```
> xts5 - lag(xts5)
> diff(xts5,lag=12,differences=1)
```

Period-over-period differences
Lagged differences

Reindexing

```
> xts1 + merge(xts2,index(xts1),fill=0)
                  e1
2017-05-04 5.231538
2017-05-05 5.829257
2017-05-06 4.000000
2017-05-07 3.000000
2017-05-08 2.000000
2017-05-09 1.000000
> xts1 - merge(xts2,index(xts1),fill=na.locf)
                  e1
2017-05-04 5.231538
2017-05-05 5.829257
2017-05-06 4.829257
2017-05-07 3.829257
2017-05-08 2.829257
2017-05-09 1.829257
```

Addition
Subtraction

Merging

```
> merge(xts2,xts1,join='inner')
                  xts2 xts1
2017-05-05 -0.8382068   10

> merge(xts2,xts1,join='left',fill=0)
                  xts2 xts1
2017-05-01  1.7482704    0
2017-05-02 -0.2314678    0
2017-05-03  0.1685517    0
2017-05-04  1.1685649    0
2017-05-05 -0.8382068   10

> rbind(xts1, xts4)
```

Inner join of `xts2` and `xts1`
Left join of `xts2` and `xts1`, fill empty spots with `0`
Combine `xts1` and `xts4` by rows

Inspect Your Data

```
> core_data <- coredata(xts2)
> index(xts1)
```

Extract core data of objects
Extract index of objects

Class Attributes

```
> indexClass(xts2)
> indexClass(convertIndex(xts,'POSIXct'))
> indexTZ(xts5)
> indexFormat(xts5) <- "%Y-%m-%d"
```

Get index class
Replacing index class
Get index class
Change format of time display

Time Zones

```
> tzone(xts1) <- "Asia/Hong_Kong"
> tzone(xts1)
```

Change the time zone
Extract the current time zone

Periods, Periodicity & Timestamps

```
> periodicity(xts5)
> to.yearly(xts5)
> to.monthly(xts3)
> to.quarterly(xts5)
> to.period(xts5,period="quarters")
> to.period(xts5,period="years")
> nmonths(xts5)
> nquarters(xts5)
> nyears(xts5)
> make.index.unique(xts3,eps=1e-4)
> make.index.unique(xts3,drop=TRUE)
> align.time(xts3,n=3600)
```

Estimate frequency of observations
Convert `xts5` to yearly OHLC
Convert `xts3` to monthly OHLC
Convert `xts5` to quarterly OHLC
Convert to quarterly OHLC
Convert to yearly OHLC
Count the months in `xts5`
Count the quarters in `xts5`
Count the years in `xts5`
Make index unique
Remove duplicate times
Round index time to the next `n` seconds

Other Useful Functions

```
> .index(xts4)
> .indexwday(xts3)
> .indexhour(xts3)
> start(xts3)
> end(xts4)
> str(xts3)
> time(xts1)
> head(xts2)
> tail(xts2)
```

Extract raw numeric index of `xts1`
Value of week(day), starting on Sunday, in index of `xts3`
Value of hour in index of `xts3`
Extract first observation of `xts3`
Extract last observation of `xts4`
Display structure of `xts3`
Extract raw numeric index of `xts1`
First part of `xts2`
Last part of `xts2`