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')**

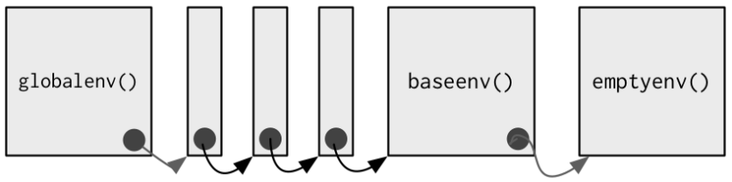


Figure 1 – The Search Path

- 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.

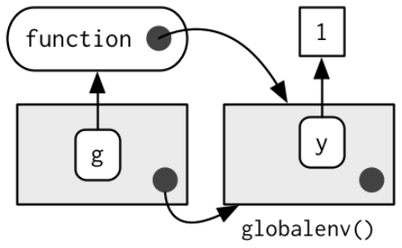
1. **<-** (Regular assignment arrow) – always creates a variable in the current environment
2. **<<-** (Deep assignment arrow) - modifies an existing variable found by walking up the parent environments

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()**

Updated: 2/16

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

Infix Functions

What are Infix 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 inspecto**r
   - 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**

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, ]`

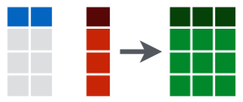
`df[2, 2]`

`nrow(df)`
Number of rows.

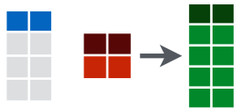
`ncol(df)`
Number of columns.

`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.

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

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

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

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

Group Cases

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

```
mtcars %>%
  group_by(cyl) %>%
  summarise(avg = mean(mpg))
```

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

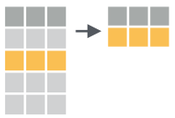
```
starwars %>%
  rowwise() %>%
  mutate(film_count = length(films))
```

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

Manipulate Cases

EXTRACT CASES

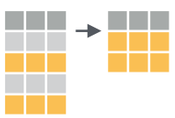
Row functions return a subset of rows as a new table.



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

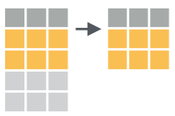
distinct(.data, …, .keep_all = FALSE**)** Remove rows with duplicate values.
distinct(mtcars, gear)

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



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

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



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

Logical and boolean operators to use with filter()

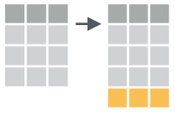
==	<	<=	is.na()	%in%	|	xor()
!=	>	>=	!is.na()	!	&	

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

ARRANGE CASES

arrange(.data, …, .by_group = FALSE**)** Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

ADD CASES



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

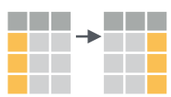
Manipulate Variables

EXTRACT VARIABLES

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

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

select(.data, …**)** Extract columns as a table.
select(mtcars, mpg, wt)



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

Use these helpers with select() and across()
e.g. select(mtcars, mpg:cyl)

contains(match**)**	**num_range(**prefix, range**)**	**:**, e.g. mpg:cyl
ends_with(match**)**	**all_of(**x**)**/**any_of(**x, …, vars**)**	**-**, e.g, -gear
starts_with(match**)**	**matches(**match**)**	**everything()**

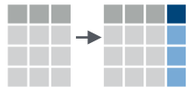
MANIPULATE MULTIPLE VARIABLES AT ONCE

across(.cols, .funs, …, .names = NULL**)** Summarise or mutate multiple columns in the same way.
summarise(mtcars, across(everything(), mean))

c_across(.cols**)** Compute across columns in row-wise data.
transmute(rowwise(UKgas), total = sum(c_across(1:2)))

MAKE NEW VARIABLES

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



mutate(.data, …, .keep = "all", .before = NULL, .after = NULL**)** Compute new column(s). Also **add_column(), add_count(),** and **add_tally().**
mutate(mtcars, gpm = 1 / mpg)

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

rename(.data, …**)** Rename columns. Use **rename_with()** to rename with a function.
rename(cars, distance = dist)

Vectorized Functions

TO USE WITH MUTATE ()

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

vectorized function

OFFSET

dplyr::**lag()** - offset elements by 1
dplyr::**lead()** - offset elements by -1

CUMULATIVE AGGREGATE

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

RANKING

dplyr::**cume_dist()** - proportion of all values <=
dplyr::**dense_rank()** - rank w ties = min, no gaps
dplyr::**min_rank()** - rank with ties = min
dplyr::**ntile()** - bins into n bins
dplyr::**percent_rank()** - min_rank scaled to [0,1]
dplyr::**row_number()** - rank with ties = "first"

MATH

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

MISCELLANEOUS

dplyr::**case_when()** - multi-case if_else()

```
starwars %>%
  mutate(type = case_when(
    height > 200 | mass > 200 ~ "large",
      species == "Droid"      ~ "robot",
      TRUE                    ~ "other")
    )
```

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

Summary Functions

TO USE WITH SUMMARISE ()

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

summary function

COUNT

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

POSITION

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

LOGICAL

mean() - proportion of TRUE's
sum() - # of TRUE's

ORDER

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

RANK

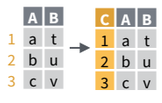
quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

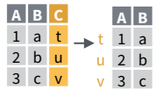
IQR() - Inter-Quartile Range
mad() - median absolute deviation
sd() - standard deviation
var() - variance

Row Names

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



tibble::**rownames_to_column()**
Move row names into col.
a <- rownames_to_column(mtcars, var = "C")

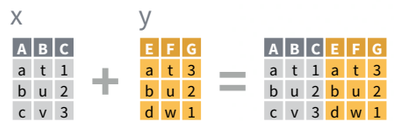


tibble::**column_to_rownames()**
Move col into row names.
column_to_rownames(a, var = "C")

Also tibble::**has_rownames()** and tibble::**remove_rownames()**.

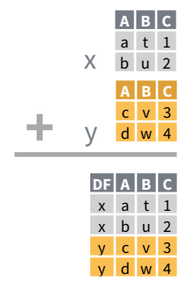
Combine Tables

COMBINE VARIABLES



bind_cols(…, .name_repair**)** Returns tables placed side by side as a single table. Column lengths must be equal. Columns will NOT be matched by id (to do that look at Relational Data below), so be sure to check that both tables are ordered the way you want before binding.

COMBINE CASES



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).

RELATIONAL DATA

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

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

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

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

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

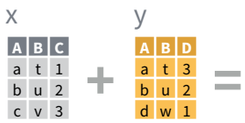
COLUMN MATCHING FOR JOINS

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

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

Use **suffix** to specify the suffix to give to unmatched columns that have the same name in both tables.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

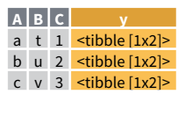
Use a "**Filtering Join**" to filter one table against the rows of another.



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

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

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



nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, …**)** Join data, nesting matches from y in a single new data frame column.

SET OPERATIONS

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

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

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

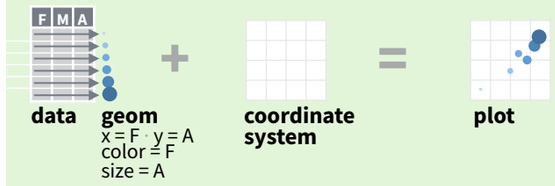
Use **setequal()** to test whether two data sets contain the exact same rows (in any order).

Data Visualization with ggplot2 : : CHEAT SHEET

Basics

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

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



Complete the template below to build a graph.

```
ggplot (data = <DATA>) +                                  required
<GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),
 stat = <STAT>, position = <POSITION> ) +                 Not required, sensible defaults supplied
<COORDINATE_FUNCTION> +
<FACET_FUNCTION> +
<SCALE_FUNCTION> +
<THEME_FUNCTION>
```

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

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))

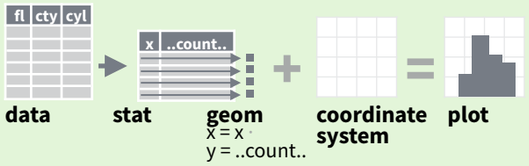
l + geom_contour(aes(z = z))
x, y, z, alpha, colour, group, linetype, size, weight

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

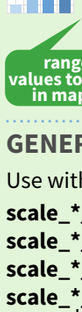
COLOR AND FILL SCALES (DISCRETE)

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..))

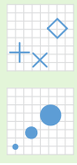
o + scale_fill_distiller(palette = "Blues"**)**

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

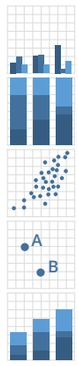
r + coord_polar(theta = "x", direction=1 **)**
theta, start, direction
Polar coordinates

r + coord_trans(xtrans = "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

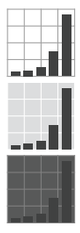
e + geom_point(position = "jitter")
Add random noise to X and Y position of each element to avoid overplotting

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(cols = vars(fl))
facet into columns based on fl

t + facet_grid(rows = vars(year))
facet into rows based on year

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

t + facet_wrap(vars(fl))
wrap facets into a rectangular layout

Set **scales** to let axis limits vary across facets

t + facet_grid(rows = vars(drv), cols = vars(fl), scales = "free")
x and y axis limits adjust to individual facets
"free_x" - x axis limits adjust
"free_y" - y axis limits adjust

Set **labeller** to adjust facet labels

t + facet_grid(cols = vars(fl), labeller = label_both)

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

t + facet_grid(rows = vars(fl), labeller = label_bquote(alpha ^ .(fl)**))**

α^c	α^d	α^e	α^p	α^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)**)**

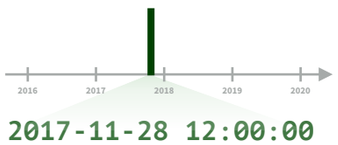
With clipping (removes unseen data points)

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

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

Dates and times with lubridate : : **CHEAT SHEET**

Date-times



2017-11-28 12:00:00
A **date-time** is a point on the timeline, stored as the number of seconds since 1970-01-01 00:00:00 UTC

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

2017-11-28
A **date** is a day stored as the number of days since 1970-01-01

d <- ***as_date(****17498****)***
"2017-11-28"

12:00:00
An hms is a **time** stored as the number of seconds since 00:00:00

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

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

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

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

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

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

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

20170131 — **ymd(), ydm()**. ymd(20170131)

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

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

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

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

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

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

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

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

fast_strptime() Faster strptime. fast_strptime('9/1/01', '%y/%m/%d')

parse_date_time() Easier strptime. parse_date_time("9/1/01", "ymd")

GET AND SET COMPONENTS

Use an accessor function to get a component.

```
d ## "2017-11-28"
day(d) ## 28
```

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

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

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

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

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

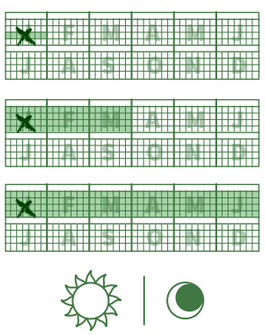
2018-01-31 11:59:59 — **day(**x**)** Day of month. day(dt)
wday(x, label, abbr**)** Day of week.
qday(x**)** Day of quarter.

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

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

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

2018-01-31 11:59:59 UTC — **tz(**x**)** Time zone. tz(dt)



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

quarter(x**)** Quarter. quarter(dt)

semester(x, with_year = FALSE**)** Semester. semester(dt)

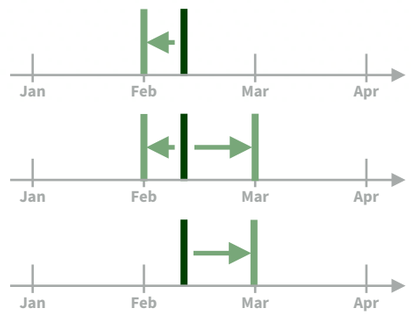
am(x**)** Is it in the am? am(dt)
pm(x**)** Is it in the pm? pm(dt)

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

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

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

Round Date-times



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

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

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

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

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

Stamp Date-times

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

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

Tip: use a date with day > 12

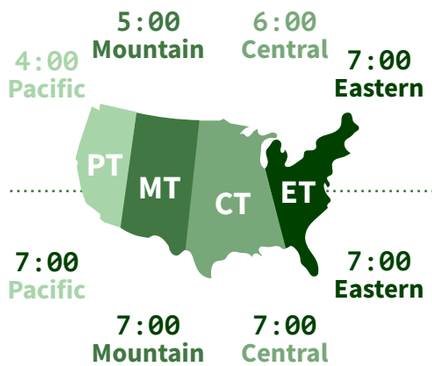
Time Zones

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

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

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

Sys.timezone() Gets current time zone.



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

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

Math with Date-times – Lubridate provides three classes of timespans to facilitate math with dates and date-times.

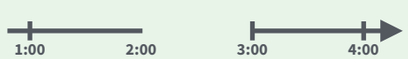
Math with date-times relies on the **timeline**, which behaves inconsistently. Consider how the timeline behaves during:

A normal day

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

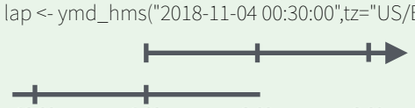
The start of daylight savings (spring forward)

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



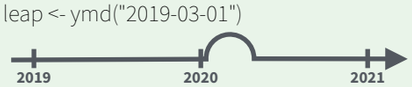
The end of daylight savings (fall back)

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



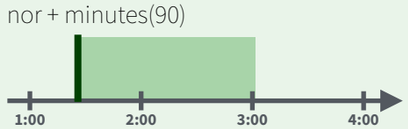
Leap years and leap seconds

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

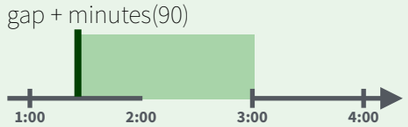


Periods track changes in clock times, which ignore time line irregularities.

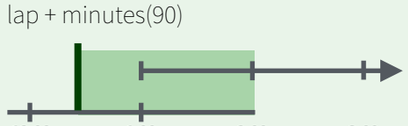
nor + minutes(90)



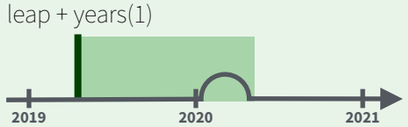
gap + minutes(90)



lap + minutes(90)

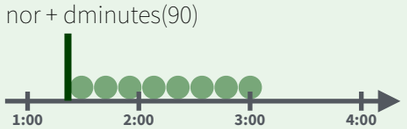


leap + years(1)

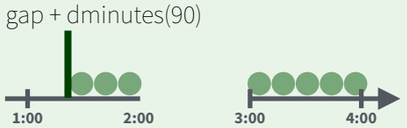


Durations track the passage of physical time, which deviates from clock time when irregularities occur.

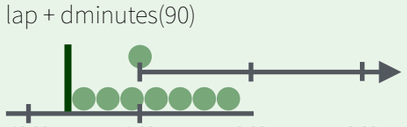
nor + dminutes(90)



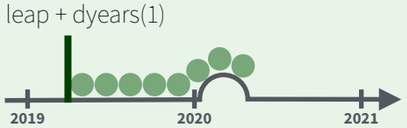
gap + dminutes(90)



lap + dminutes(90)

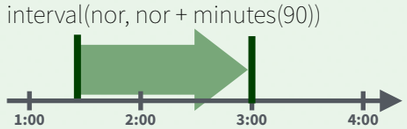


leap + dyears(1)

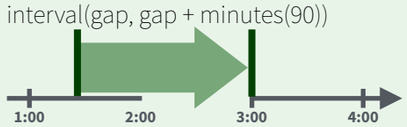


Intervals represent specific intervals of the timeline, bounded by start and end date-times.

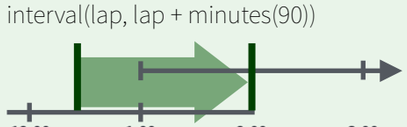
interval(nor, nor + minutes(90))



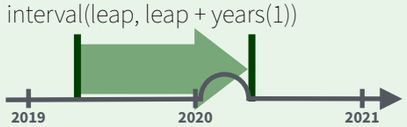
interval(gap, gap + minutes(90))



interval(lap, lap + minutes(90))



interval(leap, leap + years(1))



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

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

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

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

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

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

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

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

PERIODS

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

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

```
p <- months(3) + days(12)
p
"3m 12d 0H 0M 0S"
```

Number of months | Number of days | etc.

years(x = 1**)** x years.
months(x**)** x months.
weeks(x = 1**)** x weeks.
days(x = 1**)** x days.
hours(x = 1**)** x hours.
minutes(x = 1**)** x minutes.
seconds(x = 1**)** x seconds.
milliseconds(x = 1**)** x milliseconds.
microseconds(x = 1**)** x microseconds
nanoseconds(x = 1**)** x nanoseconds.
picoseconds(x = 1**)** x picoseconds.

period(num = NULL, units = "second", ...**)** An automation friendly period constructor. period(5, unit = "years")

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

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

DURATIONS

Add or subtract durations to model physical processes, like battery life. Durations are stored as seconds, the only time unit with a consistent length. **Difftimes** are a class of durations found in base R.

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

```
dd <- ddays(14)
dd
"1209600s (~2 weeks)"
```

Exact length in seconds | Equivalent in common units

dyears(x = 1**)** 31536000x seconds.
dmonths(x = 1**)** 2629800x seconds.
dweeks(x = 1**)** 604800x seconds.
ddays(x = 1**)** 86400x seconds.
dhours(x = 1**)** 3600x seconds.
dminutes(x = 1**)** 60x seconds.
dseconds(x = 1**)** x seconds.
dmilliseconds(x = 1**)** x $\times 10^{-3}$ seconds.
dmicroseconds(x = 1**)** x $\times 10^{-6}$ seconds.
dnanoseconds(x = 1**)** x $\times 10^{-9}$ seconds.
dpicoseconds(x = 1**)** x $\times 10^{-12}$ seconds.

duration(num = NULL, units = "second", …**)** An automation friendly duration constructor. duration(5, unit = "years")

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

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

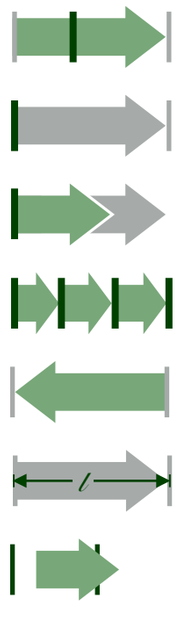
INTERVALS

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

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

Start Date | End Date

```
i <- interval(ymd("2017-01-01"), d)
j <- d %--% ymd("2017-12-31")
## 2017-01-01 UTC--2017-11-28 UTC
## 2017-11-28 UTC--2017-12-31 UTC
```



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

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

int_aligns(int1, int2**)** Do two intervals share a boundary? Also **int_overlaps()**. int_aligns(i, j)

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

int_flip(int**)** Reverse the direction of an interval. Also **int_standardize()**. int_flip(i)

int_length(int**)** Length in seconds. int_length(i)

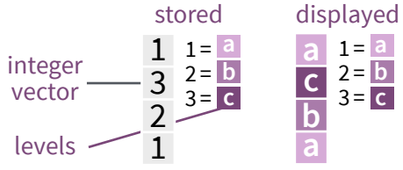
int_shift(int, by**)** Shifts an interval up or down the timeline by a timespan. int_shift(i, days(-1))

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

Factors with forcats : : **CHEAT SHEET**

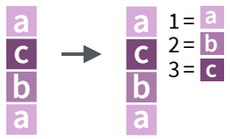
The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors



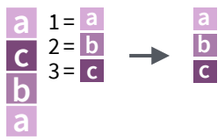
R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the levels associated with them.

Create a factor with factor()



factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA**)** Convert a vector to a factor. Also **as_factor()**.
f <- factor(c("a", "c", "b", "a"), levels = c("a", "b", "c"))

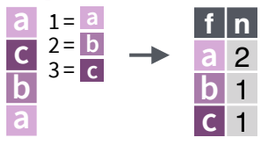
Return its levels with levels()



levels(x**)** Return/set the levels of a factor. levels(f); levels(f) <- c("x","y","z")

Use unclass() to see its structure

Inspect Factors

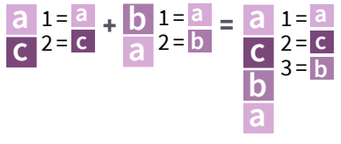


fct_count(f, sort = FALSE, prop = FALSE**)** Count the number of values with each level. fct_count(f)

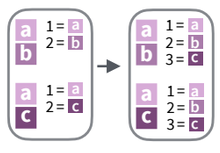
fct_match(f, lvls**)** Check for lvls in f. fct_match(f, "a")

fct_unique(f**)** Return the unique values, removing duplicates. fct_unique(f)

Combine Factors

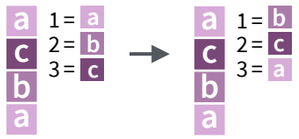


fct_c(…**)** Combine factors with different levels. Also **fct_cross()**.
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
fct_c(f1, f2)

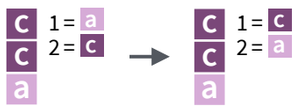


fct_unify(fs, levels = lvls_union(fs)**)** Standardize levels across a list of factors.
fct_unify(list(f2, f1))

Change the order of levels

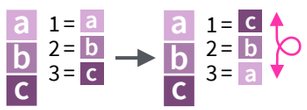


fct_relevel(.f, ..., after = 0L**)**
Manually reorder factor levels.
fct_relevel(f, c("b", "c", "a"))

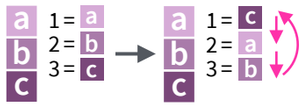


fct_infreq(f, ordered = NA**)**
Reorder levels by the frequency in which they appear in the data (highest frequency first). Also **fct_inseq()**.
f3 <- factor(c("c", "c", "a"))
fct_infreq(f3)

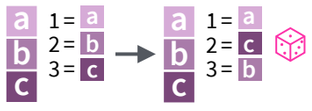
fct_inorder(f, ordered = NA**)**
Reorder levels by order in which they appear in the data.
fct_inorder(f2)



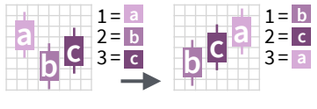
fct_rev(f**)** Reverse level order.
f4 <- factor(c("a","b","c"))
fct_rev(f4)



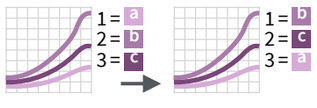
fct_shift(f**)** Shift levels to left or right, wrapping around end.
fct_shift(f4)



fct_shuffle(f, n = 1L**)** Randomly permute order of factor levels.
fct_shuffle(f4)

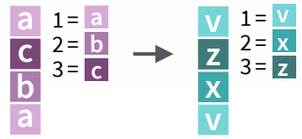


fct_reorder(.f, .x, .fun = median, ..., .desc = FALSE**)** Reorder levels by their relationship with another variable.
boxplot(data = PlantGrowth, weight ~ reorder(group, weight))

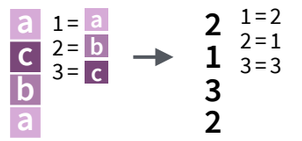


fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE**)** Reorder levels by their final values when plotted with two other variables.
ggplot(diamonds,aes(carat, price, color = fct_reorder2(color, carat, price))) + geom_smooth()

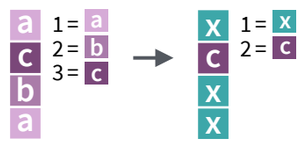
Change the value of levels



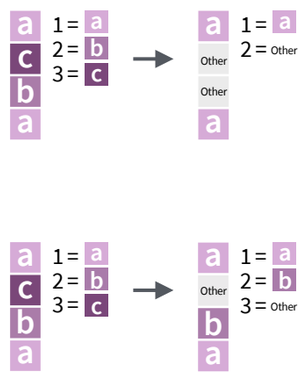
fct_recode(.f, ...**)** Manually change levels. Also **fct_relabel()** which obeys purrr::map syntax to apply a function or expression to each level.
fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))



fct_anon(f, prefix = ""**)**
Anonymize levels with random integers.
fct_anon(f)



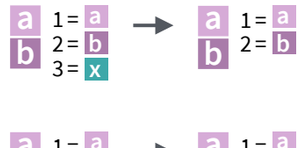
fct_collapse(.f, …, other_level = NULL**)** Collapse levels into manually defined groups.
fct_collapse(f, x = c("a", "b"))



fct_lump_min(f, min, w = NULL, other_level = "Other"**)** Lumps together factors that appear fewer than min times. Also **fct_lump_n(), fct_lump_prop(),** and **fct_lump_lowfreq()**.
fct_lump_min(f, min = 2)

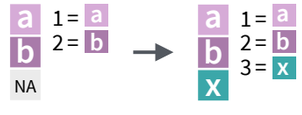
fct_other(f, keep, drop, other_level = "Other"**)** Replace levels with "other."
fct_other(f, keep = c("a", "b"))

Add or drop levels



fct_drop(f, only**)** Drop unused levels.
f5 <- factor(c("a","b"),c("a","b","x"))
f6 <- fct_drop(f5)

fct_expand(f, …**)** Add levels to a factor.
fct_expand(f6, "x")



fct_explicit_na(f, na_level="(Missing)"**)** Assigns a level to NAs to ensure they appear in plots, etc.
fct_explicit_na(factor(c("a", "b", NA)))

Cheatography

R Cheat Sheet

by [deleted] via cheatography.com/3687/cs/2469/

General

Get Help

? <Object/Function>

Find the working directory

getwd()

Setting Working directroy

setwd("~/specdata")

List files in working dir

dir()

Load code file into workspace

source("file.R")

Find the type of an object

class(my_vector)

List objects in workspace

ls()

Change page width

options(width = 160)

Operators

Assignement	var <- <New value>
Compare two objects	identical(obj1, obj2)
Equality	var1 == var2

Special Values

NA	value is **Not Available**
NaN	Not a Number
Inf	Infinity
T	True
F	False

Debugging

`traceback`

Prints function call stack

`debug( <fn> )`

Flags a function for "debug" mofr which allows you to step through execution of a function one line at a time

`browser`

Suspends the execution of a function, and outs the function in debug mode. n- next

`trace`

Allows you to insert debugging code into a function

`recover`

Allows you to modify the error behaviour so that you can browse the function call statck

Subsetting Vectors

First 10 elements	x[1:10]
Vector of all NAs	x[is.na(x)]
Vector of real values	x[!is.na(x)]
Values greater than 0	y[y > 0]
Combine conditions	x[!is.na(x) & x > 0]
3rd, 5th, 7th elements of x	x[c(3,5,7)]
All but the 2nd and 10th (neg)	x[c(-2, -10)] or x[-c(2,10)]
Access element by label	vect["bar"] or vect[c-("foo", "bar")]

Index vectors come in four different flavors -- logical vectors, vectors of positive integers, vectors of negative integers, and vectors of character strings

Vectors

Concatinate function

patients <- c("Bill","Gina","Kelly","Sean")

Matrices

Help on matrix type

? matrix

Add dimensions to vector to make a matrix

dim(my_vector) <- c(4, 5)

View dimesions of a matrix

dim(my_matrix)

View dimesions of a matrix

attributes(my_matrix)

Create a matrix. (4x5 containing 1-> 20)

my_matrix2 <- matrix(1:20,4,5)

+ Matrices can only contain a **single class** of data.
+ The first number is the number of rows and the second is the number of columns.

Data Frames

Create a data frame from avector and matrix

```
my_data <- data.frame(pati-
ents, my_matrix)
```

Add columns name to data frame

```
colnames(my_data) <- cnames-
_vector
```

Select rows based on column value

```
frame[ frame$col == "val", ]
```

Select columns by position

`frame[, 1:4]` cols 1 to 4

http://www.r-tutor.com/r-introduction/data-frame/data-frame-row-slice

Cheatography

R Cheat Sheet

by [deleted] via cheatography.com/3687/cs/2469/

Conversion

To number	as.number(x)
To boolean (logical)	as.logical(x)
To comples number	as.complex()

Reading Data

```
# Create empty data frame
  data <- data.frame()

  #Readfiles id is vector of
integers
  for ( i in id ){
    infile = sprintf("%s/%-
03d.csv", directory, i)
    data <- rbind(data,read.csv(
infile ))
  }
  head(data)
```

IF statement

```
if(<condition>) {
    ## do something
} else {
    ## do something else
}
if(<condition1>) {
    ## do something
} else if(<condition2>) {
    ## do something different
} else {
    ## do something different
}
```

For Statement

```
for(i in 1:10) {
    print(i)
}
```

While Statemement

```
count <- 0
while(count < 10) {
    print(count)
    count <- count + 1
}
```

Repeat Statement

```
x0 <- 1
tol <- 1e-8
repeat {
    x1 <- computeEstimate()
    if(abs(x1 - x0) < tol) {
        break
    } else {
        x0 <- x1
    }
}
```

next,return

```
for(i in 1:100) {
    if(i <= 20) {
        ## Skip the first 20
iterations
        next
    }
    ## Do something here
}
```

next is used to skip an iteration of a loop

Loop functions

lapply	Loop over a list and evaluate a function on each element
sapply	Same as lapply but try to simplify the result
tapply	Apply a function over the margins of an array

Loop functions (cont)

mapply	Multivariate version of lapply
apply	Used to evaluate a function (often an anonymous one) over the margins of an array.
rowSums	apply(x,1,sum)
rowMeans	apply(x,1,mean)
colSums	apply(x,2,sum)
colMeans	apply(x,2,mean)

```
x<- list( a = 1:5, b= rnorm(10))
lapply(x,mean)
```

An annonymous fn for extracting the 1st col of each matrix

```
>lapply(x,function(elt) elt[,1])
```

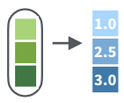
Apply functions with purrr : : **CHEAT SHEET**

Map Functions

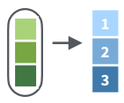
ONE LIST

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

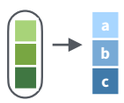
```
x <- list(1:10, 11:20, 21:30)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
map(l1, sort, decreasing = TRUE)
```



map_dbl(.x, .f, …**)**
Return a double vector.
map_dbl(x, mean)



map_int(.x, .f, ...**)**
Return an integer vector.
map_int(x, length)



map_chr(.x, .f, …**)**
Return a character vector.
map_chr(l1, paste, collapse = "")

map_lgl(.x, .f, …**)**
Return a logical vector.
map_lgl(x, is.integer)

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

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

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

TWO LISTS

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

```
y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y, ~ .x * .y)
```

map2_dbl(.x, .y, .f, …**)**
Return a double vector.
map2_dbl(y, z, ~ .x / .y)

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

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

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

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

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

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

MANY LISTS

pmap(.l, .f, …**)** Apply a function to groups of elements from a list of lists or vectors, return a list.

```
pmap(list(x, y, z), ~ ..1 * (..2 + ..3))
```

pmap_dbl(.l, .f, …**)**
Return a double vector.
pmap_dbl(list(y, z), ~ .x / .y)

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

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

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

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

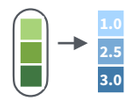
pmap_dfr(.l, .f, ..., .id = NULL**)** Return a data frame created by row-binding.
pmap_dfr(list(l1, l2),
~ as.data.frame(c(.x, .y)))

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

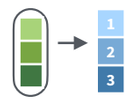
LISTS AND INDEXES

imap(.x, .f, ...**)** Apply .f to each element and its index, return a list.

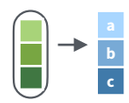
```
imap(y, ~ paste0(.y, ": ", .x))
```



imap_dbl(.x, .f, …**)**
Return a double vector.
imap_dbl(y, ~ .y)



imap_int(.x, .f, ...**)**
Return an integer vector.
imap_int(y, ~ .y)



imap_chr(.x, .f, …**)**
Return a character vector.
imap_chr(y, ~ paste0(.y, ": ", .x))

imap_lgl(.x, .f, …**)**
Return a logical vector.
imap_lgl(l1, ~ is.character(.y))

imap_dfc(.x, .f, ...**)**
Return a data frame created by column-binding.
imap_dfc(l2,
~ as.data.frame(c(.x, .y)))

imap_dfr(.x, .f, ..., .id = NULL**)**
Return a data frame created by row-binding.
imap_dfr(l2,
~ as.data.frame(c(.x, .y)))

iwalk(.x, .f, ...**)** Trigger side effects, return invisibly.
iwalk(z, ~ print(paste0(.y, ": ", .x)))

Function Shortcuts

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

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

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

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

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

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

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

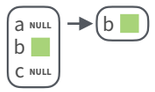
imap(list(a, b, c), ~ paste0(.y, ": ", .x)
outputs **"index: value"** for each item

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

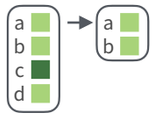
Work with Lists

Filter

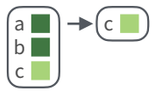
keep(.x, .p, …**)**
Select elements that pass a logical test.
Conversely, **discard()**.
keep(x, is.na)



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



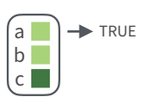
head_while(.x, .p, …**)**
Return head elements until one does not pass.
Also **tail_while()**.
head_while(x, is.character)



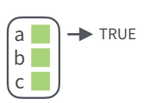
detect(.x, .f, ..., dir = c("forward", "backward"), .right = NULL, .default = NULL**)**
Find first element to pass.
detect(x, is.character)

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

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

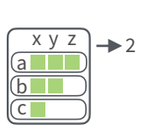


some(.x, .p, …**)**
Do some elements pass a test?
some(x, is.character)



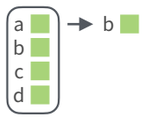
none(.x, .p, …**)**
Do no elements pass a test?
none(x, is.character)

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

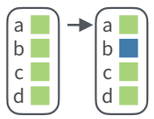


vec_depth(x**)**
Return depth (number of levels of indexes).
vec_depth(x)

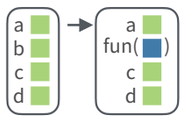
Index



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

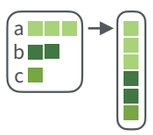


assign_in(x, where, value**)**
Assign a value to a location using pluck selection.
assign_in(x, "b", 5)
x %>% assign_in("b", 5)



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

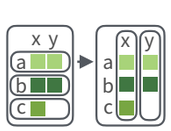
Reshape



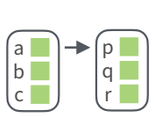
flatten(.x**)** Remove a level of indexes from a list.
Also **flatten_chr()** etc.
flatten(x)

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

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



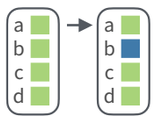
transpose(.l, .names = NULL**)**
Transposes the index order in a multi-level list.
transpose(x)



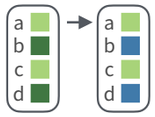
set_names(x, nm = x**)**
Set the names of a vector/list directly or with a function.
set_names(x, c("p", "q", "r"))
set_names(x, tolower)

Modify

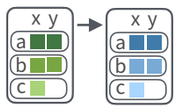
modify(.x, .f, ...**)** Apply a function to each element. Also **modify2()**, and **imodify()**.
modify(x, ~.+ 2)



modify_at(.x, .at, .f, ...**)** Apply a function to selected elements. Also **map_at()**.
modify_at(x, "b", ~.+ 2)

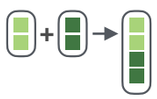


modify_if(.x, .p, .f, ...**)** Apply a function to elements that pass a test. Also **map_if()**.
modify_if(x, is.numeric,~.+2)



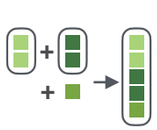
modify_depth(.x, .depth, .f, ...**)** Apply function to each element at a given level of a list. Also **map_depth()**.
modify_depth(x, 2, ~.+ 2)

Combine



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

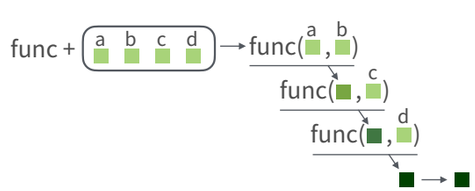
prepend(x, values, before = 1**)**
Add values to start of list.
prepend(x, list(d = 1))



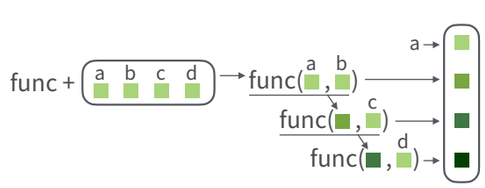
splice(…**)** Combine objects into a list, storing S3 objects as sub-lists.
splice(x, y, "foo")

Reduce

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



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



List-Columns

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

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

WORK WITH LIST-COLUMNS

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

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

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

rmarkdown : : CHEAT SHEET

What is rmarkdown?

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

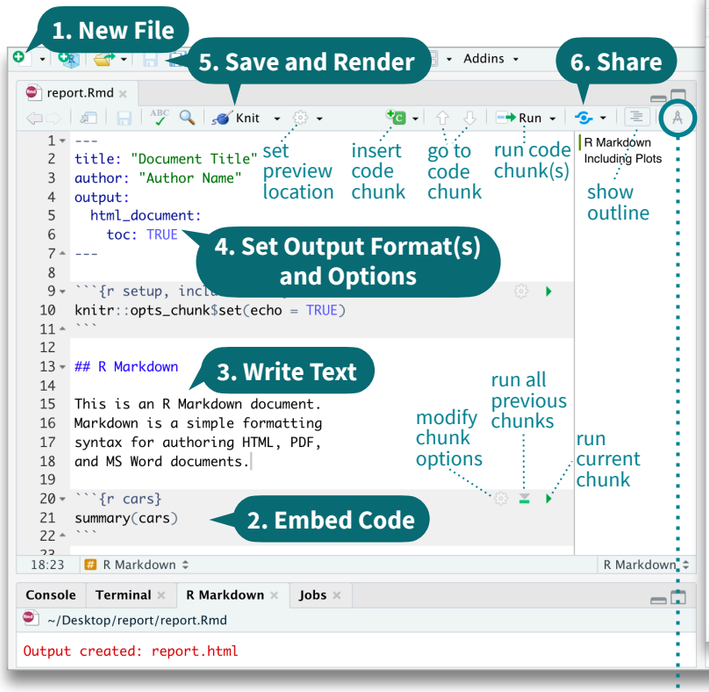
Dynamic Documents • Knit together plots, tables, and results with narrative text. Render to a variety of formats like HTML, PDF, MS Word, or MS Powerpoint.

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

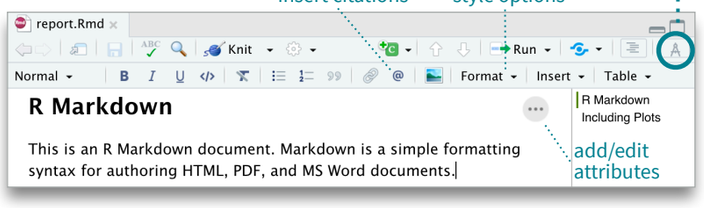
Workflow

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

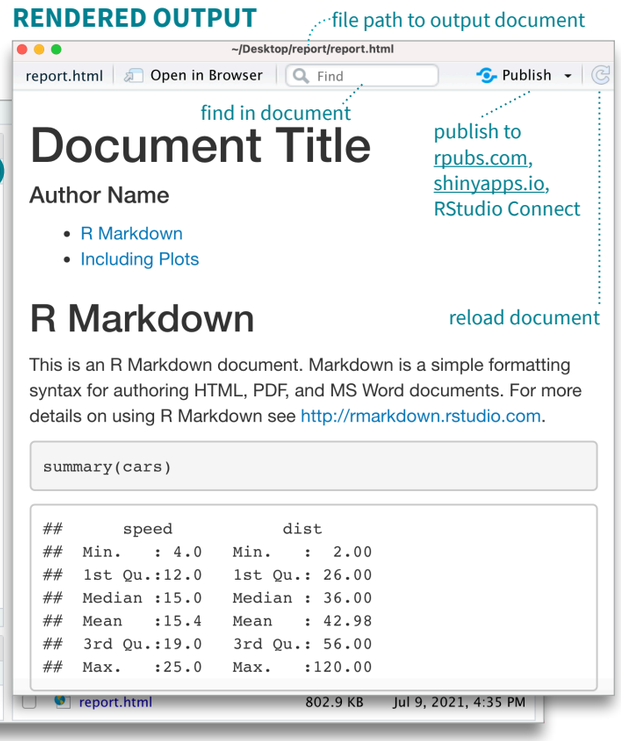
SOURCE EDITOR



VISUAL EDITOR



RENDERED OUTPUT



Insert Citations

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

BUILD YOUR BIBLIOGRAPHY

- Add BibTeX or CSL bibliographies to the YAML header.

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

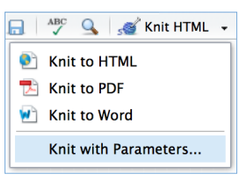
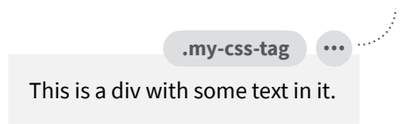
- If Zotero is installed locally, your main library will automatically be available.
- Add citations by DOI by searching "from DOI" in the **Insert Citation** dialog.

INSERT CITATIONS

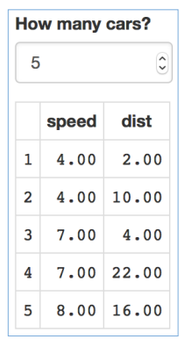
- Access the **Insert Citations** dialog in the Visual Editor by clicking the **@** symbol in the toolbar or by clicking **Insert > Citation**.
- Add citations with markdown syntax by typing **[@cite]** or **@cite**.

Insert Tables

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

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

Table with kable

eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62

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

Write with Markdown

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

Plain text. End a line with two spaces to start a new paragraph. Also end with a backslash\ to make a new line.	Plain text. End a line with two spaces to start a new paragraph. Also end with a backslash to make a new line.
italics and **bold** superscript^2^/subscript~2~ ~~strikethrough~~ escaped: * _ \\ endash: --, emdash: ---	*italics* and **bold** superscript^2/subscript$_2$ ~~strikethrough~~ escaped: * _ \ endash: –, emdash: —
# Header 1 ## Header 2 ... ###### Header 6	**Header 1** **Header 2** ... **Header 6**
- unordered list - item 2 - item 2a (indent 1 tab) - item 2b 1. ordered list 2. item 2 - item 2a (indent 1 tab) - item 2b	• unordered list • item 2 • item 2a (indent 1 tab) • item 2b 1. ordered list 2. item 2 • item 2a (indent 1 tab) • item 2b
<link url> [This is a link.](link url) [This is another link][id]. At the end of the document: [id]: link url ![Caption](image.png) or ![Caption][id2] At the end of the document: [id2]: image.png	http://www.rstudio.com/ This is a link. This is another link.  Caption.
`verbatim code` ``` multiple lines of verbatim code ```	`verbatim code` multiple lines of verbatim code
> block quotes	block quotes
equation: $e^{i \pi} + 1 = 0$ equation block: $$E = mc^{2}$$	equation: $e^{i\pi} + 1 = 0$ equation block: $$E = mc^2$$
horizontal rule: ---	horizontal rule:
| Right | Left | Default | Center | |-------:|:------|----------|:--------:| | 12 | 12 | 12 | 12 | | 123 | 123 | 123 | 123 | | 1 | 1 | 1 | 1 |	Right / Left / Default / Center 12 / 12 / 12 / 12 123 / 123 / 123 / 123 1 / 1 / 1 / 1
HTML Tabsets # Results {.tabset} ## Plots text text ## Tables more text	**Results** Plots | Tables text

Embed Code with knitr

CODE CHUNKS

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

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

SET GLOBAL OPTIONS

Set options for the entire document in the first chunk.

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

INLINE CODE

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

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

OPTION	DEFAULT	EFFECTS
echo	TRUE	display code in output document
error	FALSE	TRUE (display error messages in doc) FALSE (stop render when error occurs)
eval	TRUE	run code in chunk
include	TRUE	include chunk in doc after running
message	TRUE	display code messages in document
warning	TRUE	display code warnings in document
results	"markup"	"asis" (passthrough results) "hide" (don't display results) "hold" (put all results below all code)
fig.align	"default"	"left", "right", or "center"
fig.alt	NULL	alt text for a figure
fig.cap	NULL	figure caption as a character string
fig.path	"figure/"	prefix for generating figure file paths
fig.width & fig.height	7	plot dimensions in inches
out.width		rescales output width, e.g. "75%", "300px"
collapse	FALSE	collapse all sources & output into a single block
comment	"##"	prefix for each line of results
child	NULL	files(s) to knit and then include
purl	TRUE	include or exclude a code chunk when extracting source code with knitr::purl()

See more options and defaults by running **str(knitr::opts_chunk$get())**

RStudio

Set Output Formats and their Options in YAML

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

```
---
title: "My Document"
author: "Author Name"
output:
  html_document:
    toc: TRUE
---
```

Indent format 2 characters, indent options 4 characters

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

* Requires LaTeX, use **tinytex::install_tinytex()**

Also see **flexdashboard**, **bookdown**, **distill**, and **blogdown**.

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

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

Render

When you render a document, rmarkdown:

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

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

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

Share

Publish on RStudio Connect

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

rstudio.com/products/connect/

More Header Options

PARAMETERS

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

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

```
---
params:
  state: "hawaii"
---
```

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

```
```{r}
data <- df[, params$state]
summary(data)
```
```

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



REUSABLE TEMPLATES

1. **Create a new package** with a inst/rmarkdown/templates directory.
2. **Add a folder** containing **template.yaml** (below) and **skeleton.Rmd** (template contents).

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

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

BOOTSWATCH THEMES

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

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


```
---
title: "Document Title"
author: "Author Name"
output:
  html_document:
    theme:
      bootswatch: solar
---
```

CUSTOM THEMES

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

```
---
output:
  html_document:
    theme:
      bg: "#121212"
      fg: "#E4E4E4"
      base_font:
        google: "Prompt"
---
```

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

STYLING WITH CSS AND SCSS

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

```
---
title: "My Document"
author: "Author Name"
output:
  html_document:
    css: "style.css"
---
```

Apply CSS styling by writing HTML tags directly or:

- Use markdown to apply style attributes inline.

Bracketed Span
A [green]{.my-color} word.

Fenced Div
::: {.my-color}
All of these words
are green.
:::

A green word.

All of these words are green.

- Use the Visual Editor. Go to **Format > Div/Span** and add CSS styling directly with Edit Attributes.



INTERACTIVITY

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

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

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

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

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



	speed	dist
1	4.00	2.00
2	4.00	10.00
3	7.00	4.00
4	7.00	22.00
5	8.00	16.00

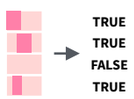
Also see Shiny Prerendered for better performance. **rmarkdown.rstudio.com/authoring_shiny_prerendered**

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

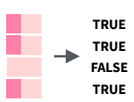
String manipulation with stringr : : **CHEAT SHEET**

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

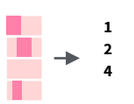
Detect Matches



str_detect(string, **pattern**, negate = FALSE**)** Detect the presence of a pattern match in a string. Also **str_like()**. str_detect(fruit, "a")

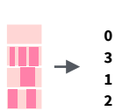


str_starts(string, **pattern**, negate = FALSE**)** Detect the presence of a pattern match at the beginning of a string. Also **str_ends()**. str_starts(fruit, "a")



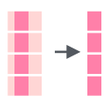
str_which(string, **pattern**, negate = FALSE**)** Find the indexes of strings that contain a pattern match. str_which(fruit, "a")

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all()**. str_locate(fruit, "a")

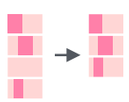


str_count(string, **pattern**) Count the number of matches in a string. str_count(fruit, "a")

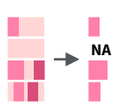
Subset Strings



str_sub(string, start = 1L, end = -1L**)** Extract substrings from a character vector. str_sub(fruit, 1, 3); str_sub(fruit, -2)



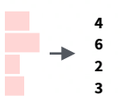
str_subset(string, **pattern**, negate = FALSE**)** Return only the strings that contain a pattern match. str_subset(fruit, "p")



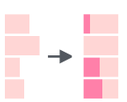
str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all()** to return every pattern match. str_extract(fruit, "[aeiou]")

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

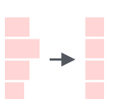
Manage Lengths



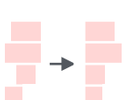
str_length(string**)** The width of strings (i.e. number of code points, which generally equals the number of characters). str_length(fruit)



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



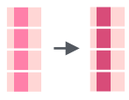
str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "..."**)** Truncate the width of strings, replacing content with ellipsis. str_trunc(sentences, 6)



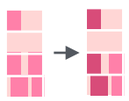
str_trim(string, side = c("both", "left", "right")**)** Trim whitespace from the start and/or end of a string. str_trim(str_pad(fruit, 17))

str_squish(string**)** Trim whitespace from each end and collapse multiple spaces into single spaces. str_squish(str_pad(fruit, 17, "both"))

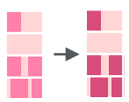
Mutate Strings



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



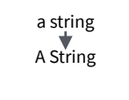
str_replace(string, **pattern**, replacement**)** Replace the first matched pattern in each string. Also **str_remove()**. str_replace(fruit, "p", "-")



str_replace_all(string, **pattern**, replacement**)** Replace all matched patterns in each string. Also **str_remove_all()**. str_replace_all(fruit, "p", "-")

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

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

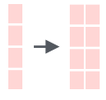


str_to_title(string, locale = "en"**)**[1] Convert strings to title case. Also **str_to_sentence()**. str_to_title(sentences)

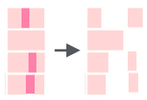
Join and Split

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

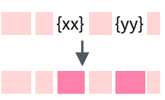
str_flatten(string, collapse = ""**)** Combines into a single string, separated by collapse. str_flatten(fruit, ", ")



str_dup(string, times**)** Repeat strings times times. Also **str_unique()** to remove duplicates. str_dup(fruit, times = 2)



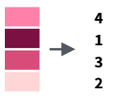
str_split_fixed(string, **pattern**, n**)** Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split()** to return a list of substrings and **str_split_n()** to return the nth substring. str_split_fixed(sentences, " ", n=3)



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

str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA"**)** Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")

Order Strings

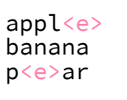


str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...**)**[1] Return the vector of indexes that sorts a character vector. fruit[str_order(fruit)]

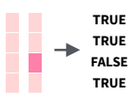
str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...**)**[1] Sort a character vector. str_sort(fruit)

Helpers

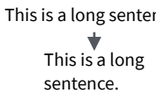
str_conv(string, encoding**)** Override the encoding of a string. str_conv(fruit,"ISO-8859-1")



str_view_all(string, **pattern**, match = NA**)** View HTML rendering of all regex matches. Also **str_view()** to see only the first match. str_view_all(sentences, "[aeiou]")



str_equal(x, y, locale = "en", ignore_case = FALSE, ...**)**[1] Determine if two strings are equivalent. str_equal(c("a", "b"), c("a", "c"))



str_wrap(string, width = 80, indent = 0, exdent = 0**)** Wrap strings into nicely formatted paragraphs. str_wrap(sentences, 20)

[1] See **bit.ly/ISO639-1** for a complete list of locales.

Need to Know

Pattern arguments in 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:]]**

[:space:]
↵ new line

[:blank:]
space
tab

.

[:graph:]

[:punct:]
. , : ; ? ! / * @ #
- _ " ' [ ] { } ( )

[:symbol:]
| ` = + ^
~ < > $

[: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
y 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
Y Z

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

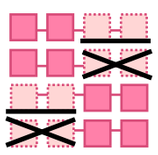
ANCHORS

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

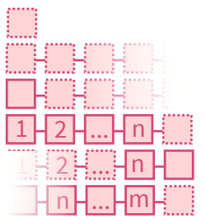
look <- function(rx) str_view_all("bacad", rx)



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

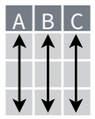
regexp	matches	example	
(ab|d)e	sets precedence	alt("(ab|d)e")	abcde

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

Data tidying with tidyr : : **CHEAT SHEET**

Tidy data is a way to organize tabular data in a consistent data structure across packages.
A table is tidy if:

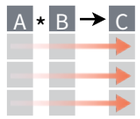


Each **variable** is in its own **column**

&

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

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



Preserve **cases** in vectorized operations

Tibbles

AN ENHANCED DATA FRAME

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

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

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

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

CONSTRUCT A TIBBLE

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

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

Both make this tibble

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

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

enframe(x, name = "name", value = "value"**)** Convert a named vector to a tibble. Also **deframe()**.

is_tibble(x**)** Test whether x is a tibble.

Reshape Data - Pivot data to reorganize values into a new layout.

table4a

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

→

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

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

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

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

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

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

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

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

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

table5

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

→

country	year
A	1999
A	2000
B	1999
B	2000

unite(data, col, …, sep = "_", remove = TRUE, na.rm = FALSE**)** Collapse cells across several columns into a single column.

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

table3

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

→

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

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", …**)** Separate each cell in a column into several columns. Also **extract()**.

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

table3

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

→

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

separate_rows(data, …, sep = "[^[:alnum:].]+", convert = FALSE**)** Separate each cell in a column into several rows.

separate_rows(table3, rate, sep = "/")

Expand Tables

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

x

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

→

x1	x2
A	1
A	2
B	1
B	2

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

x

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

→

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

complete(data, …, fill = list()**)** Add missing possible combinations of values of variables listed in … Fill remaining variables with NA.
complete(mtcars, cyl, gear, carb)

Handle Missing Values

Drop or replace explicit missing values (NA).

x

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

→

x1	x2
A	1
D	3

drop_na(data, …**)** Drop rows containing NA's in … columns.
drop_na(x, x2)

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(data, …, .direction = "down"**)** Fill in NA's in … columns using the next or previous value.
fill(x, x2)

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(data, replace**)** Specify a value to replace NA in selected columns.
replace_na(x, list(x2 = 2))

Nested Data

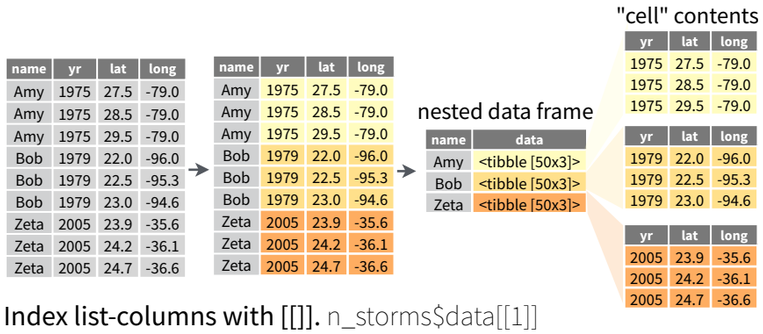
A **nested data frame** stores individual tables as a list-column of data frames within a larger organizing data frame. List-columns can also be lists of vectors or lists of varying data types. Use a nested data frame to:

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

CREATE NESTED DATA

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

1. Group the data frame with **group_by()** and use **nest()** to move the groups into a list-column.
```
n_storms <- storms %>%
  group_by(name) %>%
  nest()
```
2. Use **nest(new_col = c(x, y))** to specify the columns to group using dplyr::**select()** syntax.
```
n_storms <- storms %>%
  nest(data = c(year:long))
```



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

CREATE TIBBLES WITH LIST-COLUMNS

tibble::**tribble(**…**)** Makes list-columns when needed.
```
tribble( ~max, ~seq,
          3,   1:3,
          4,   1:4,
          5,   1:5)
```

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

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

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

OUTPUT LIST-COLUMNS FROM OTHER FUNCTIONS

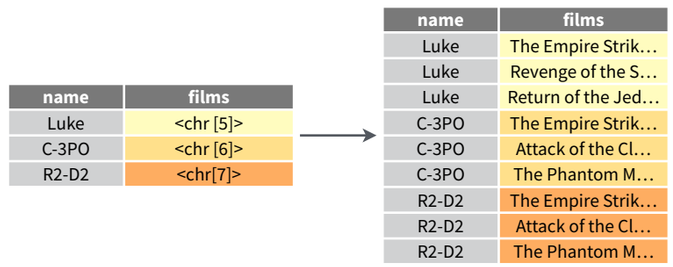
dplyr::**mutate()**, **transmute()**, and **summarise()** will output list-columns if they return a list.
```
mtcars %>%
  group_by(cyl) %>%
  summarise(q = list(quantile(mpg)))
```

RESHAPE NESTED DATA

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

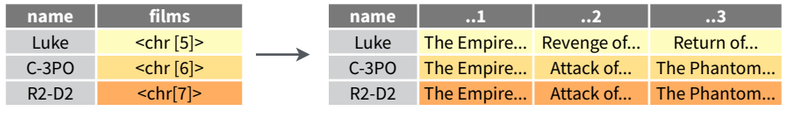
unnest_longer(data, col, values_to = NULL, indices_to = NULL**)** Turn each element of a list-column into a row.

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



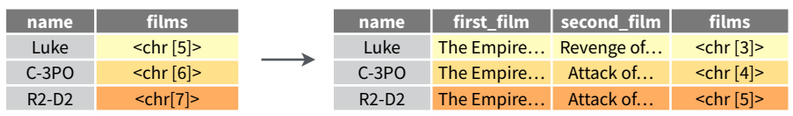
unnest_wider(data, col**)** Turn each element of a list-column into a regular column.

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



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

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



TRANSFORM NESTED DATA

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

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

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

```
n_storms %>%
  rowwise() %>%
  mutate(n = list(dim(data)))
```
dim() returns two values per row
wrap with list to tell mutate to create a list-column

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

```
n_storms %>%
  rowwise() %>%
  mutate(n = nrow(data))
```
nrow() returns one integer per row

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

```
starwars %>%
  rowwise() %>%
  mutate(transport = list(append(vehicles, starships)))
```
append() returns a list for each row, so col type must be list

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

```
starwars %>%
  rowwise() %>%
  mutate(n_transports = length(c(vehicles, starships)))
```
length() returns one integer per row

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