

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  
}
```

Functions

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

Example

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

Reading and Writing Data

Also see the **readr** package.

Input	Output	Description
df <- read.table('file.txt')	write.table(df, 'file.txt')	Read and write a delimited text file.
df <- read.csv('file.csv')	write.csv(df, 'file.csv')	Read and write a comma separated value file. This is a special case of read.table/write.table.
load('file.RData')	save(df, file = 'file.Rdata')	Read and write an R data file, a file type special for R.

Conditions	a == b	Are equal	a > b	Greater than	a >= b	Greater than or equal to	is.na(a)	Is missing
	a != b	Not equal	a < b	Less than	a <= b	Less than or equal to	is.null(a)	Is null

Types

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

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.

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

Lists

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

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

Also see the `dplyr` package.

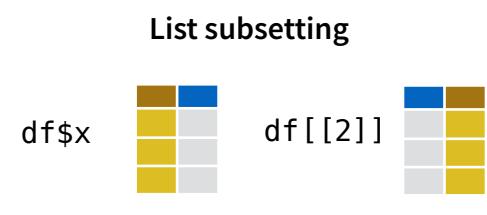
Data Frames

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

x	y
1	a
2	b
3	c

Matrix subsetting

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



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

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

`cbind` - Bind columns.

`rbind` - Bind rows.

Values of x in order.

Strings

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

Factors

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

Statistics

<code>lm(y ~ x, data=df)</code>	Linear model.
<code>glm(y ~ x, data=df)</code>	Generalised linear model.
<code>summary</code>	Get more detailed information out a model.
<code>pairwise.t.test</code>	Perform a t-test for paired data.

Distributions

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

Plotting

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

Dates

See the `lubridate` package.

caret Package

Cheat Sheet

Specifying the Model

Possible syntaxes for specifying the variables in the model:

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

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

Remember to:

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

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

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

Parallel Processing

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

```
# on Mac OS or Linux      # on Windows
library(doMC)              library(doParallel)
registerDoMC(cores=4)       cl <- makeCluster(2)
                           registerDoParallel(cl)
```

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

Preprocessing

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

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

Methods include:

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

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

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

Adding Options

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

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

Resampling Options

`trainControl` is used to choose a resampling method:

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

Methods and options are:

- `"cv"` for K-fold cross-validation (`number` sets the # folds).
- `"repeatedcv"` for repeated cross-validation (`repeats` for # repeats).
- `"boot"` for bootstrap (`number` sets the iterations).
- `"LGOCV"` for leave-group-out (`number` and `p` are options).
- `"L0O"` for leave-one-out cross-validation.
- `"oob"` for out-of-bag resampling (only for some models).
- `"timeslice"` for time-series data (options are `initialWindow`, `horizon`, `fixedWindow`, and `skip`).

Performance Metrics

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

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

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

Grid Search

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

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

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

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

Random Search

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

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

Subsampling

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

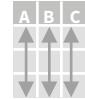
```
trainControl(sampling = "down")
```

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

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**



`x %>% f(y)` becomes `f(x, y)`

Summarise Cases

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

VARIATIONS

`summarise_all()` - Apply funs to every column.

`summarise_at()` - Apply funs to specific columns.

`summarise_if()` - Apply funs to all cols of one type.

Group Cases

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



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

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

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

Manipulate Cases

EXTRACT CASES

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



`filter(.data, ...)` Extract rows that meet logical criteria. `filter(iris, Sepal.Length > 7)`



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



`sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame())` Randomly select fraction of rows.
`sample_frac(iris, 0.5, replace = TRUE)`



`slice(.data, ...)` Select rows by position.
`slice(iris, 10:15)`



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

Manipulate Variables

EXTRACT VARIABLES

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



`pull(.data, var = -1)` Extract column values as a vector. Choose by name or index.
`pull(iris, Sepal.Length)`



`select(.data, ...)` Extract columns as a table. Also `select_if()`.
`select(iris, Sepal.Length, Species)`

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

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

MAKE NEW VARIABLES

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



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

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

`mutate_all(.tbl, .funs, ...)` Apply funs to every column. Use with `funs()`. Also `mutate_if()`.
`mutate_all(faithful, funs(log(.), log2(.)))`
`mutate_if(iris, is.numeric, funs(log(.)))`

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

`add_column(.data, ..., .before = NULL, .after = NULL)` Add new column(s). Also `add_count()`, `add_tally()`.
`add_column(mtcars, new = 1:32)`

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



Vector Functions

TO USE WITH MUTATE ()

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

vectorized function →

OFFSETS

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

CUMULATIVE AGGREGATES

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

RANKINGS

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

MATH

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

MISC

dplyr::case_when() - multi-case if_else()
dplyr::coalesce() - first non-NA values by element across a set of vectors
dplyr::if_else() - element-wise if() + else()
dplyr::na_if() - replace specific values with NA
 pmax() - element-wise max()
 pmin() - element-wise min()
dplyr::recode() - Vectorized switch()
dplyr::recode_factor() - Vectorized switch() for factors

Summary Functions

TO USE WITH SUMMARISE ()

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

summary function →

COUNTS

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

LOCATION

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

LOGICALS

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

POSITION/ORDER

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

RANK

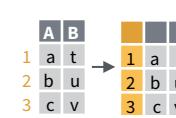
quantile() - nth quantile
min() - minimum value
max() - maximum value

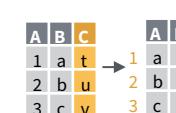
SPREAD

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

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

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

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

Combine Tables

COMBINE VARIABLES

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

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

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

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

A B C D	left_join(x, y, by = NULL,
a t 1 3 b u 2 2 c v 3 NA	copy=FALSE, suffix=c("x","y"),...)
	Join matching values from y to x.

A B C D	right_join(x, y, by = NULL, copy =
a t 1 3 b u 2 2 d w NA 1	FALSE, suffix=c("x","y"),...)
	Join matching values from x to y.

A B C D	inner_join(x, y, by = NULL, copy =
a t 1 3 b u 2 2	FALSE, suffix=c("x","y"),...)
	Join data. Retain only rows with matches.

A B C D	full_join(x, y, by = NULL,
a t 1 3 b u 2 2 d w NA 1	copy=FALSE, suffix=c("x","y"),...)
	Join data. Retain all values, all rows.

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"))

COMBINE CASES

X	y	=
A B C a t 1 b u 2 c v 3	A B C C v 3 d w 4	

Use **bind_rows()** to paste tables below each other as they are.

df A B C	bind_rows(..., .id = NULL)
x a t 1 x b u 2 x c v 3 z c v 3 z d w 4	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)

A B C	intersect(x, y, ...)
c v 3	Rows that appear in both x and y.

A B C	setdiff(x, y, ...)
a t 1 b u 2	Rows that appear in x but not y.

A B C	union(x, y, ...)
a t 1 b u 2 c v 3 d w 4	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).

EXTRACT ROWS

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

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

A B C	semi_join(x, y, by = NULL, ...)
a t 1 b u 2	Return rows of x that have a match in y.

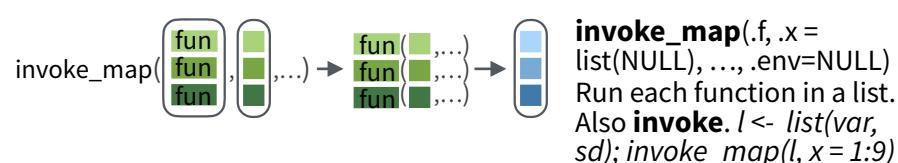
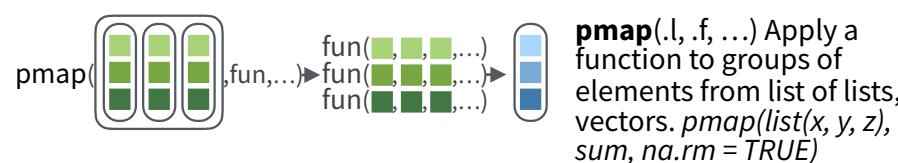
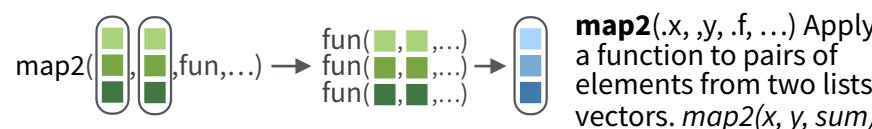
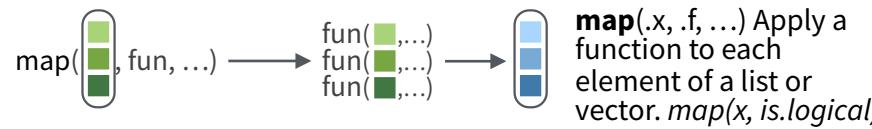
A B C	anti_join(x, y, by = NULL, ...)
c v 3	Return rows of x that do not have a match in y. USEFUL TO SEE WHAT WILL NOT BE JOINED.

Apply functions with purrr :: CHEAT SHEET



Apply Functions

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



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

OUTPUT

map(), **map2()**, **pmap()**, **imap** and **invoke_map** each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. **map2_chr**, **pmap_lgl**, etc.

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

SHORTCUTS - within a purrr function:

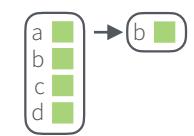
"**name**" becomes **function(x) x\$name**. e.g. `map(l, "a")` extracts \$a from each element of l

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

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

Work with Lists

FILTER LISTS



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



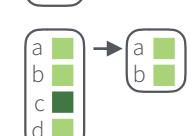
keep(.x, .p, ...) Select elements that pass a logical test. `keep(x, is.na)`



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

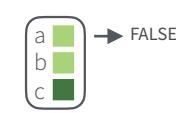


compact(.x, .p = identity) Drop empty elements. `compact(x)`

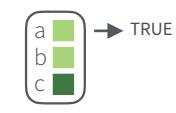


head_while(.x, .p, ...) Return head elements until one does not pass. Also **tail_while**. `head_while(x, is.character)`

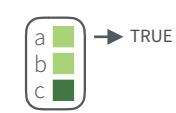
SUMMARISE LISTS



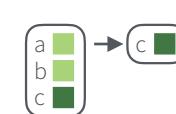
every(.x, .p, ...) Do all element pass a test? `every(x, is.character)`



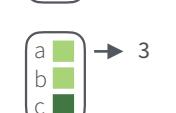
some(.x, .p, ...) Do some elements pass a test? `some(x, is.character)`



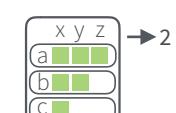
has_element(.x, .y) Does a list contain an element? `has_element(x, "foo")`



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

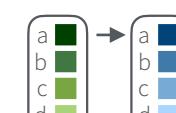


detect_index(.x, .f, ..., .right = FALSE, .p) Find index of first element to pass. `detect_index(x, is.character)`

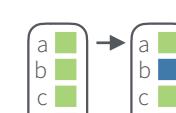


vec_depth(x) Return depth (number of levels of indexes). `vec_depth(x)`

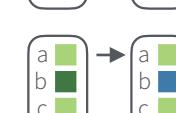
TRANSFORM LISTS



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



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

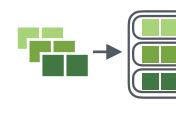


modify_if(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. `modify_if(x, is.numeric, ~.+2)`



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

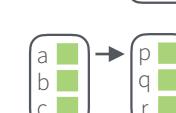
WORK WITH LISTS



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

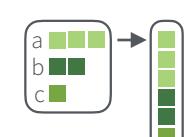


cross2(.x, .y, .filter = NULL) All combinations of .x and .y. Also **cross**, **cross3**, **cross_df**. `cross2(1:3, 4:6)`

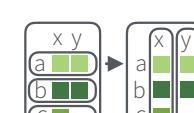


set_names(x, nm = x) Set the names of a vector/list directly or with a function. `set_names(x, c("p", "q", "r"))`
`set_names(x, tolower)`

RESHAPE LISTS

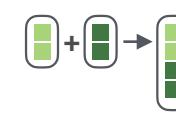


flatten(.x) Remove a level of indexes from a list. Also **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. `flatten(x)`

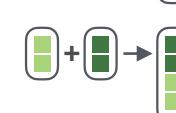


transpose(.l, .names = NULL) Transposes the index order in a multi-level list. `transpose(x)`

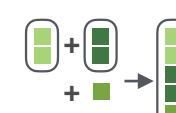
JOIN (TO) LISTS



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

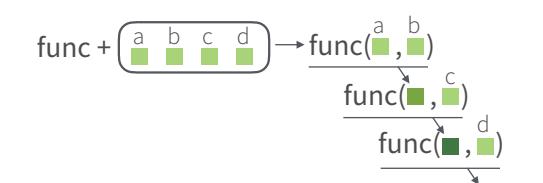


prepend(x, values, before = 1) Add to start of list. `prepend(x, list(d = 1))`

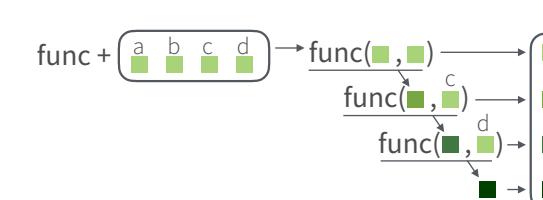


splice(...) Combine objects into a list, storing S3 objects as sub-lists. `splice(x, y, "foo")`

Reduce Lists



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



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

compose() Compose multiple functions.

lift() Change the type of input a function takes. Also **lift_dl**, **lift_lv**, **lift_vd**, **lift_vl**.

rerun() Rerun expression n times.

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

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

safely() Modify func to return list of results whenever an error occurs (instead of error).

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

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



Nested Data

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

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

`n_iris$data[[1]]`

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

`n_iris`

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

`n_iris$data[[2]]`

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

`n_iris$data[[3]]`

Use a nested data frame to:

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

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

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

Species	S.L	S.W	P.L	P.W	Species	S.L	S.W	P.L	P.W
setosa	5.1	3.5	1.4	0.2	setosa	5.1	3.5	1.4	0.2
setosa	4.9	3.0	1.4	0.2	setosa	4.9	3.0	1.4	0.2
setosa	4.7	3.2	1.3	0.2	setosa	4.7	3.2	1.3	0.2
setosa	4.6	3.1	1.5	0.2	setosa	4.6	3.1	1.5	0.2
setosa	5.0	3.6	1.4	0.2	setosa	5.0	3.6	1.4	0.2
versi	7.0	3.2	4.7	1.4	versi	7.0	3.2	4.7	1.4
versi	6.4	3.2	4.5	1.5	versi	6.4	3.2	4.5	1.5
versi	6.9	3.1	4.9	1.5	versi	6.9	3.1	4.9	1.5
versi	5.5	2.3	4.0	1.3	versi	5.5	2.3	4.0	1.3
versi	6.5	2.8	4.6	1.5	versi	6.5	2.8	4.6	1.5
virgini	6.3	3.3	6.0	2.5	virgini	6.3	3.3	6.0	2.5
virgini	5.8	2.7	5.1	1.9	virgini	5.8	2.7	5.1	1.9
virgini	7.1	3.0	5.9	2.1	virgini	7.1	3.0	5.9	2.1
virgini	6.3	2.9	5.6	1.8	virgini	6.3	2.9	5.6	1.8
virgini	6.5	3.0	5.8	2.2	virgini	6.5	3.0	5.8	2.2

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

`tidy::nest(data, ..., .key = data)`

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

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

Species	data	Species	S.L	S.W	P.L	P.W
setos	<tibble [50x4]>	setosa	5.1	3.5	1.4	0.2
versi	<tibble [50x4]>	setosa	4.9	3.0	1.4	0.2
virgini	<tibble [50x4]>	setosa	4.7	3.2	1.3	0.2

`n_iris %>% unnest()`

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

Unnests a nested data frame.

List Column Workflow

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

1 Make a list column

Species	S.L	S.W	P.L	P.W
setosa	5.1	3.5	1.4	0.2
setosa	4.9	3.0	1.4	0.2
setosa	4.7	3.2	1.3	0.2
setosa	4.6	3.1	1.5	0.2

Species	data
setosa	<tibble [50x4]>
versi	<tibble [50x4]>
virgini	<tibble [50x4]>

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

2 Work with list columns

Species	data	model
setosa	<tibble [50x4]>	<code>Call: lm(S.L ~ ., df)</code> <code>Coefs: (Int) S.W P.L P.W</code> 2.3 0.6 0.2 0.2
versi	<tibble [50x4]>	<code>Call: lm(S.L ~ ., df)</code> <code>Coefs: (Int) S.W P.L P.W</code> 1.8 0.3 0.9 -0.6
virgini	<tibble [50x4]>	<code>Call: lm(S.L ~ ., df)</code> <code>Coefs: (Int) S.W P.L P.W</code> 0.6 0.3 0.9 -0.1

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

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

3 Simplify the list column

Species	beta
setos	2.35
versi	1.89
virgini	0.69

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

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

1. MAKE A LIST COLUMN

- You can create list columns with functions in the **tibble** and **dplyr** packages, as well as **tidyR**'s `nest()`

`tibble::tribble(...)`

Makes list column when needed

max	seq
3	<code>c(1:3)</code>
4	<code>c(1:4)</code>
5	<code>c(1:5)</code>

`tibble::tibble(...)`

Saves list input as list columns

`tibble(max = c(3, 4, 5), seq = list(1:3, 1:4, 1:5))`

`tibble::enframe(x, name="name", value="value")`

Converts multi-level list to tibble with list cols

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

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

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

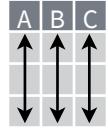
`n_iris %>% mutate(n = map(data, dim))`

Data tidying with `tidyr` :: CHEATSHEET

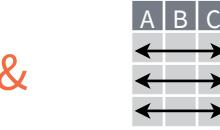


Tidy data is a way to organize tabular data in a consistent data structure across packages.

A table is tidy if:



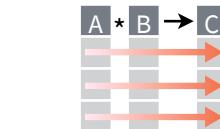
Each **variable** is in its own **column**



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



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



Preserve **cases** in vectorized operations

Tibbles

AN ENHANCED DATA FRAME

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

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

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

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

CONSTRUCT A TIBBLE

tibble(...) Construct by columns.

`tibble(x = 1:3, y = c("a", "b", "c"))`

Both make this tibble

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

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

enframe(x, name = "name", value = "value")

Convert a named vector to a tibble. Also **deframe()**.

is_tibble(x) Test whether x is a tibble.



Reshape Data

- Pivot data to reorganize values into a new layout.

table4a

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



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

table2

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

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

Split Cells

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

table5

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

country	year
A	1999
A	2000
B	1999
B	2000

table3

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

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

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

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

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

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

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

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

Expand Tables

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

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

expand(data, ...) Create a new tibble with all possible combinations of the values listed in ... Drop other variables.

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

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

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

```
drop_na(x, x2)
```

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

```
fill(x, x2)
```

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))
```

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

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

name	data
Luke	<tibble [50x3]>
C-3PO	<tibble [50x3]>
R2-D2	<tibble [50x3]>

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

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)
```

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

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

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

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

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

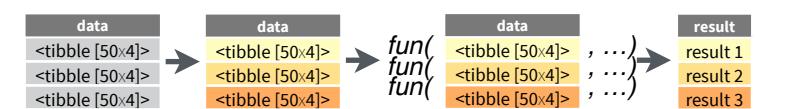
name	films
Luke	<chr [5]>
C-3PO	<chr [6]>
R2-D2	<chr[7]>

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

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.