

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.

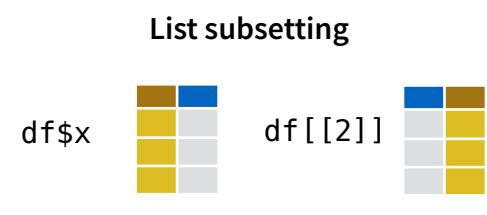
Data Frames

`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.

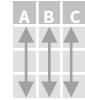
Dates

See the `lubridate` package.

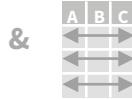
Data Transformation with dplyr :: CHEAT SHEET



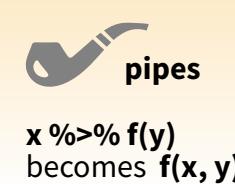
dplyr functions work with pipes and expect **tidy data**. In tidy data:



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. Summary functions take vectors as input and return one value (see back).

	summary function
	summarise(.data, ...) Compute table of summaries. Also summarise_() . <code>summarise(mtcars, avg = mean(mpg))</code>
	count(x, ..., wt = NULL, sort = FALSE) Count number of rows in each group defined by the variables in ... Also tally() . <code>count(iris, Species)</code>

VARIATIONS

- summarise_all()** - Apply funs to every column.
- summarise_at()** - Apply funs to specific columns.
- summarise_if()** - Apply funs to all cols of one type.

Group Cases

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

	<code>mtcars %>% group_by(cyl) %>% summarise(avg = mean(mpg))</code>
--	--

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

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

Manipulate Cases

EXTRACT CASES

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

	filter(.data, ...) Extract rows that meet logical criteria. Also filter_() . <code>filter(iris, Sepal.Length > 7)</code>
	distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values. Also distinct_() . <code>distinct(iris, Species)</code>
	sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select fraction of rows. <code>sample_frac(iris, 0.5, replace = TRUE)</code>
	sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows. <code>sample_n(iris, 10, replace = TRUE)</code>
	slice(.data, ...) Select rows by position. Also slice_() . <code>slice(iris, 10:15)</code>
	top_n(x, n, wt) Select and order top n entries (by group if grouped data). <code>top_n(iris, 5, Sepal.Width)</code>

Logical and boolean operators to use with filter()

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

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

ARRANGE CASES

	arrange(.data, ...) Order rows by values of a column (low to high), use with desc() to order from high to low. <code>arrange(mtcars, mpg)</code> <code>arrange(mtcars, desc(mpg))</code>
--	---

ADD CASES

	add_row(.data, ..., .before = NULL, .after = NULL) Add one or more rows to a table. <code>add_row(faithful, eruptions = 1, waiting = 1)</code>
--	---

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

	select(.data, ...) Extract columns by name. Also select_if() . <code>select(iris, Sepal.Length, Species)</code>
--	---

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

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

MAKE NEW VARIABLES

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

vectorized function
mutate(.data, ...) Compute new column(s). <code>mutate(mtcars, gpm = 1/mpg)</code>
transmute(.data, ...) Compute new column(s), drop others. <code>transmute(mtcars, gpm = 1/mpg)</code>
mutate_all(.tbl, .funs, ...) Apply funs to every column. Use with funs() . <code>mutate_all(faithful, funs(log(.), log2(.)))</code>
mutate_at(.tbl, .cols, .funs, ...) Apply funs to specific columns. Use with funs() , vars() and the helper functions for select() . <code>mutate_at(iris, vars(-Species), funs(log(.)))</code>
mutate_if(.tbl, .predicate, .funs, ...) Apply funs to all columns of one type. Use with funs() . <code>mutate_if(iris, is.numeric, funs(log(.)))</code>
add_column(.data, ..., .before = NULL, .after = NULL) Add new column(s). <code>add_column(mtcars, new = 1:32)</code>
rename(.data, ...) Rename columns. <code>rename(iris, Length = Sepal.Length)</code>



Vectorized Functions

TO USE WITH MUTATE ()

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

vectorized function →

OFFSETS

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

CUMULATIVE AGGREGATES

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

RANKINGS

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

MATH

+, -, *, /, ^, %/%, %% - arithmetic ops
log(), log2(), log10() - logs
<, <=, >, >=, !=, == - logical comparisons

MISC

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

Summary Functions

TO USE WITH SUMMARISE ()

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

summary function →

COUNTS

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

LOCATION

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

LOGICALS

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

POSITION/ORDER

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

RANK

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

SPREAD

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

Row Names

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

A B
1 a t → 1 a t
2 b u → 2 b u
3 c v → 3 c v

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

A B C
1 a t → 1 a t
2 b u → 2 b u
3 c v → 3 c v

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

Also has `_rownames()`, `remove_rownames()`

Summary Functions

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 the column(s) to match on.
`left_join(x, y, by = "A")`

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

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

COMBINE CASES

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.

bind_rows(..., .id = NULL)
Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured)

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

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

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

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

EXTRACT ROWS

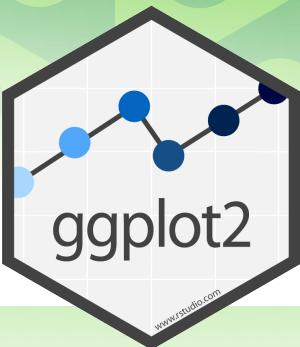
x	y	=
A B C a t 1 b u 2 c v 3	A B D a t 3 b u 2 d w 1	

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

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

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

Data Visualization with ggplot2 :: CHEAT SHEET



Basics

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



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



Complete the template below to build a graph.

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

required

Not required, sensible defaults supplied

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

aesthetic mappings **data** **geom**

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

last_plot() Returns the last plot

ggsave("plot.png", width = 5, height = 5) Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Geoms

Use a geom function to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.

GRAPHICAL PRIMITIVES

- a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))
- a + geom_blank()**
(Useful for expanding limits)
- b + geom_curve(aes(yend = lat + 1, xend=long+1, curvature=z))** - x, xend, y, yend, alpha, angle, color, curvature, linetype, size
- a + geom_path(lineend="butt", linejoin="round", linemitre=1)**
x, y, alpha, color, group, linetype, size
- a + geom_polygon(aes(group = group))**
x, y, alpha, color, fill, group, linetype, size
- b + geom_rect(aes(xmin = long, ymin=lat, xmax=long + 1, ymax = lat + 1))** - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size
- a + geom_ribbon(aes(ymin=unemploy - 900, ymax=unemploy + 900))** - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

- common aesthetics: x, y, alpha, color, linetype, size
- b + geom_abline(aes(intercept=0, slope=1))**
 - b + geom_hline(aes(yintercept = lat))**
 - b + geom_vline(aes(xintercept = long))**

- b + geom_segment(aes(yend=lat+1, xend=long+1))**
- b + geom_spoke(aes(angle = 1:1155, radius = 1))**

ONE VARIABLE continuous

- c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)
- c + geom_area(stat = "bin")**
x, y, alpha, color, fill, linetype, size
- c + geom_density(kernel = "gaussian")**
x, y, alpha, color, fill, group, linetype, size, weight
- c + geom_dotplot()**
x, y, alpha, color, fill
- c + geom_freqpoly()**
x, y, alpha, color, group, linetype, size
- c + geom_histogram(binwidth = 5)**
x, y, alpha, color, fill, linetype, size, weight
- c2 + geom_qq(aes(sample = hwy))**
x, y, alpha, color, fill, linetype, size, weight

discrete

- d <- ggplot(mpg, aes(f1))
- d + geom_bar()**
x, alpha, color, fill, linetype, size, weight

TWO VARIABLES

continuous x , continuous y

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

- e + geom_jitter(height = 2, width = 2)**
x, y, alpha, color, fill, shape, size

- e + geom_point()**, x, y, alpha, color, fill, shape, size, stroke

- e + geom_quantile()**, x, y, alpha, color, group, linetype, size, weight

- e + geom_rug(sides = "bl")**, x, y, alpha, color, linetype, size

- e + geom_smooth(method = lm)**, x, y, alpha, color, fill, group, linetype, size, weight

- e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1, check_overlap = TRUE)**, x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

discrete x , continuous y

- f <- ggplot(mpg, aes(class, hwy))

- f + geom_col()**, x, y, alpha, color, fill, group, linetype, size

- f + geom_boxplot()**, x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight

- f + geom_dotplot(binaxis = "y", stackdir = "center")**, x, y, alpha, color, fill, group

- f + geom_violin(scale = "area")**, x, y, alpha, color, fill, group, linetype, size, weight

discrete x , discrete y

- g <- ggplot(diamonds, aes(cut, color))

- g + geom_count()**, x, y, alpha, color, fill, shape, size, stroke

THREE VARIABLES

- seals\$z <- with(seals, sqrt(delta_long^2 + delta_lat^2))
l <- ggplot(seals, aes(long, lat))

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

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

Data Import :: CHEAT SHEET

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

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

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

OTHER TYPES OF DATA

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

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

Save Data

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

Comma delimited file

```
write_csv(x, path, na = "NA", append = FALSE,  
          col_names = !append)
```

File with arbitrary delimiter

```
write_delim(x, path, delim = " ", na = "NA",  
            append = FALSE, col_names = !append)
```

CSV for excel

```
write_excel_csv(x, path, na = "NA", append =  
                FALSE, col_names = !append)
```

String to file

```
write_file(x, path, append = FALSE)
```

String vector to file, one element per line

```
write_lines(x, path, na = "NA", append = FALSE)
```

Object to RDS file

```
write_rds(x, path, compress = c("none", "gz",  
                                "bz2", "xz"), ...)
```

Tab delimited files

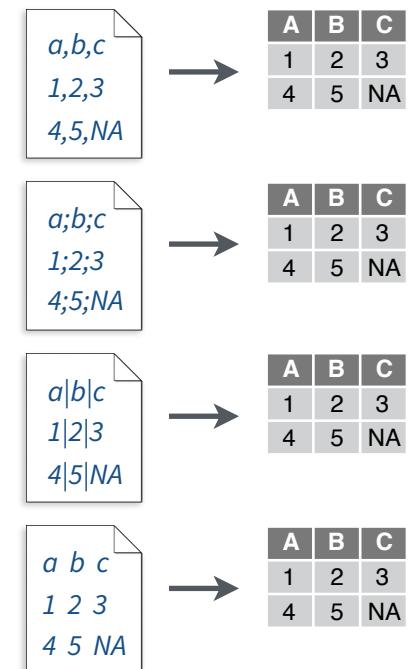
```
write_tsv(x, path, na = "NA", append = FALSE,  
          col_names = !append)
```



Read Tabular Data

- These functions share the common arguments:

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



Comma Delimited Files

```
read_csv("file.csv")
```

To make file.csv run:

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

Semi-colon Delimited Files

```
read_csv2("file2.csv")
```

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

Files with Any Delimiter

```
read_delim("file.txt", delim = "|")
```

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

Fixed Width Files

```
read_fwf("file.fwf", col_positions = c(1, 3, 5))
```

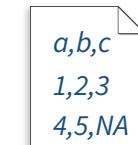
```
write_file(x = "a b c\n1 2 3\n4 5 NA", path = "file.fwf")
```

Tab Delimited Files

```
read_tsv("file.tsv") Also read_table().
```

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

USEFUL ARGUMENTS



Example file

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

1	2	3
4	5	NA

Skip lines

```
read_csv(f, skip = 1)
```

A	B	C
1	2	3
4	5	NA

No header

```
read_csv(f, col_names = FALSE)
```

A	B	C
1	2	3

Read in a subset

```
read_csv(f, n_max = 1)
```

x	y	z
A	B	C
1	2	3

Provide header

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

A	B	C
NA	2	3
4	5	NA

Missing Values

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

Read Non-Tabular Data

Read a file into a single string

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

Read each line into its own string

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

Read Apache style log files

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

Read a file into a raw vector

```
read_file_raw(file)
```

Read each line into a raw vector

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



Data types

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

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

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

age is an integer

sex is a character

1. Use **problems()** to diagnose problems.
`x <- read_csv("file.csv"); problems(x)`

2. Use a **col_** function to guide parsing.

- **col_guess()** - the default
 - **col_character()**
 - **col_double()**, **col_euro_double()**
 - **col_datetime(format = "")** Also **col_date(format = "")**, **col_time(format = "")**
 - **col_factor(levels, ordered = FALSE)**
 - **col_integer()**
 - **col_logical()**
 - **col_number()**, **col_numeric()**
 - **col_skip()**
- `x <- read_csv("file.csv", col_types = cols(
 A = col_double(),
 B = col_logical(),
 C = col_factor()))`

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

- **parse_guess()**
 - **parse_character()**
 - **parse_datetime()** Also **parse_date()** and **parse_time()**
 - **parse_double()**
 - **parse_factor()**
 - **parse_integer()**
 - **parse_logical()**
 - **parse_number()**
- `x$A <- parse_number(x$A)`

Tibbles - an enhanced data frame

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

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

# A tibble: 234 × 6	manufacturer	model	displ	cyl	trans
1 audi	a4	1.80	4	auto(l4)	
2 audi	a4	1.80	4	auto(l4)	
3 audi	a4	2.00	4	auto(l4)	
4 audi	a4	2.00	4	auto(l4)	
5 audi	a4	2.00	4	auto(l4)	
6 audi	a4	2.00	4	auto(l4)	
7 audi	a4	3.10	6	quattro	
8 audi	a4	3.10	6	quattro	
9 audi	a4	3.10	6	quattro	
10 audi	a4	3.10	6	quattro	
... with 224 more rows, and 3 more variables: year <int>, cyl <int>, trans <chr>					

tibble display

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

A large table to display

data frame display

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

CONSTRUCT A TIBBLE IN TWO WAYS

tibble(...)	Construct by columns. <code>tibble(x = 1:3, y = c("a", "b", "c"))</code>	Both make this tibble
tribble(...)	Construct by rows. <code>tribble(~x, ~y, 1, "a", 2, "b", 3, "c")</code>	<code>A tibble: 3 × 2</code> <code>x <int> y <chr></code> 1 1 a 2 2 b 3 3 c

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

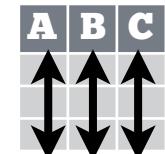
enframe(x, name = "name", value = "value")
Convert named vector to a tibble

is_tibble(x) Test whether x is a tibble.

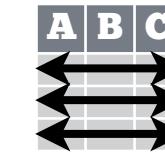
Tidy Data with tidyverse

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

A table is tidy if:



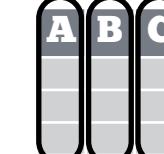
&



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

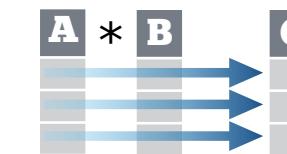
Each **observation**, or **case**, is in its own **row**

Tidy data:



Makes variables easy to access as vectors

$A * B \rightarrow C$



Preserves cases during vectorized operations

Reshape Data - change the layout of values in a table

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

gather(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor_key = FALSE)

gather() moves column names into a **key** column, gathering the column values into a single **value** column.

table4a

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



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

key value

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

table2

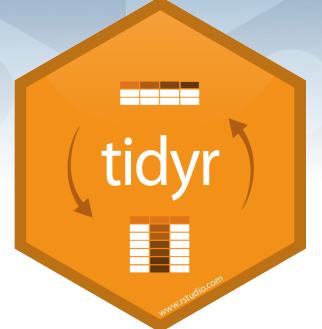
country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
B	1999	cases	37K
B	1999	pop	172M
C	1999	cases	212K
C	1999	pop	1T
A	2000	cases	2K
A	2000	pop	80K
B	2000	cases	80K
B	2000	pop	174M
C	2000	cases	213K
C	2000	pop	1T

key value

`spread(table2, type, count)`

Split Cells

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



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

Separate each cell in a column to make several columns.

table3

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

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

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

separate_rows(data, ..., sep = "[^[:alnum:]].+", convert = FALSE)

Separate each cell in a column to make several rows. Also **separate_rows_()**.

table3

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

separate_rows(table3, rate)

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

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

Collapse cells across several columns to make a single column.

table5

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

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

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

Handle Missing Values

drop_na(data, ...)

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

x1	x2</

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

	<code>str_detect(string, pattern)</code> Detect the presence of a pattern match in a string. <code>str_detect(fruit, "a")</code>
	<code>str_which(string, pattern)</code> Find the indexes of strings that contain a pattern match. <code>str_which(fruit, "a")</code>
	<code>str_count(string, pattern)</code> Count the number of matches in a string. <code>str_count(fruit, "a")</code>
	<code>str_locate(string, pattern)</code> Locate the positions of pattern matches in a string. Also <code>str_locate_all</code> . <code>str_locate(fruit, "a")</code>

Subset Strings

	<code>str_sub(string, start = 1L, end = -1L)</code> Extract substrings from a character vector. <code>str_sub(fruit, 1, 3); str_sub(fruit, -2)</code>
	<code>str_subset(string, pattern)</code> Return only the strings that contain a pattern match. <code>str_subset(fruit, "b")</code>
	<code>str_extract(string, pattern)</code> Return the first pattern match found in each string, as a vector. Also <code>str_extract_all</code> to return every pattern match. <code>str_extract(fruit, "[aeiou]")</code>
	<code>str_match(string, pattern)</code> Return the first pattern match found in each string, as a matrix with a column for each () group in pattern. Also <code>str_match_all</code> . <code>str_match(sentences, "(a the) ([^]+)")</code>

Manage Lengths

	<code>str_length(string)</code> The width of strings (i.e. number of code points, which generally equals the number of characters). <code>str_length(fruit)</code>
	<code>str_pad(string, width, side = c("left", "right", "both"), pad = " ")</code> Pad strings to constant width. <code>str_pad(fruit, 17)</code>
	<code>str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...")</code> Truncate the width of strings, replacing content with ellipsis. <code>str_trunc(fruit, 3)</code>
	<code>str_trim(string, side = c("both", "left", "right"))</code> Trim whitespace from the start and/or end of a string. <code>str_trim(fruit)</code>

Mutate Strings

	<code>str_sub()</code> <- value. Replace substrings by identifying the substrings with <code>str_sub()</code> and assigning into the results. <code>str_sub(fruit, 1, 3) <- "str"</code>
	<code>str_replace(string, pattern, replacement)</code> Replace the first matched pattern in each string. <code>str_replace(fruit, "a", "-")</code>
	<code>str_replace_all(string, pattern, replacement)</code> Replace all matched patterns in each string. <code>str_replace_all(fruit, "a", "-")</code>
	<code>str_to_lower(string, locale = "en")¹</code> Convert strings to lower case. <code>str_to_lower(sentences)</code>
	<code>str_to_upper(string, locale = "en")¹</code> Convert strings to upper case. <code>str_to_upper(sentences)</code>
	<code>str_to_title(string, locale = "en")¹</code> Convert strings to title case. <code>str_to_title(sentences)</code>

Join and Split

	<code>str_c(..., sep = "", collapse = NULL)</code> Join multiple strings into a single string. <code>str_c(letters, LETTERS)</code>
	<code>str_c(..., sep = "", collapse = NULL)</code> Collapse a vector of strings into a single string. <code>str_c(letters, collapse = "")</code>
	<code>str_dup(string, times)</code> Repeat strings times times. <code>str_dup(fruit, times = 2)</code>
	<code>str_split_fixed(string, pattern, n)</code> Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also <code>str_split</code> to return a list of substrings. <code>str_split_fixed(fruit, " ", n=2)</code>
	<code>str_glue(..., .sep = "", .envir = parent.frame())</code> Create a string from strings and {expressions} to evaluate. <code>str_glue("Pi is {pi}")</code>
	<code>str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA")</code> Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. <code>str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")</code>

Order Strings

	<code>str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)¹</code> Return the vector of indexes that sorts a character vector. <code>x[str_order(x)]</code>
	<code>str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)¹</code> Sort a character vector. <code>str_sort(x)</code>

Helpers

	<code>str_conv(string, encoding)</code> Override the encoding of a string. <code>str_conv(fruit, "ISO-8859-1")</code>
	<code>str_view(string, pattern, match = NA)</code> View HTML rendering of first regex match in each string. <code>str_view(fruit, "[aeiou]")</code>
	<code>str_view_all(string, pattern, match = NA)</code> View HTML rendering of all regex matches. <code>str_view_all(fruit, "[aeiou]")</code>
	<code>str_wrap(string, width = 80, indent = 0, exdent = 0)</code> Wrap strings into nicely formatted paragraphs. <code>str_wrap(sentences, 20)</code>

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

