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 Libraries

`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.

`table(x)`
See counts of values.

`rev(x)`
Return x reversed.

`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

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

Learn more at **web page or vignette** • package version • Updated: 3/15

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'
```

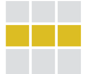
The Environment

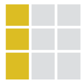
`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

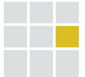
You can use the environment panel in RStudio to browse variables in your environment.

Matrixes

`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 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.

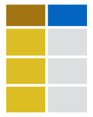
Also see the **dplyr** library.

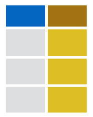
Data Frames

`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

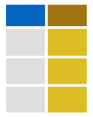
`df$x`


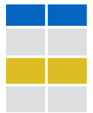
`df[[2]]`


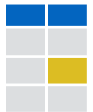
Understanding a data frame

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

Matrix subsetting

`df[ , 2]`


`df[2, ]`


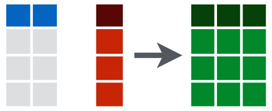
`df[2, 2]`


`nrow(df)`
Number of rows.

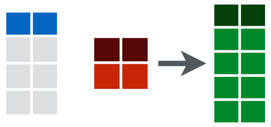
`ncol(df)`
Number of columns.

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

cbind - Bind columns.



rbind - Bind rows.



Strings

Also see the **stringr** library.

`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 but 'cutting' into sections.

Statistics

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

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

`summary`
Get more detailed information out a model.

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

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

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

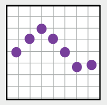
`aov`
Analysis of variance.

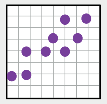
Distributions

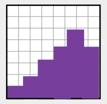
	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poison	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** library.


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


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


`hist(x)`
Histogram of x.

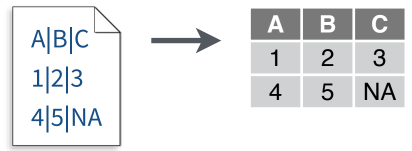
Dates

See the **lubridate** library.

Data import with the tidyverse : : CHEATSHEET

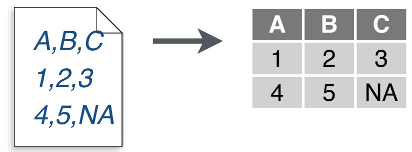
Read Tabular Data with readr

read_*(file, col_names = TRUE, col_types = NULL, col_select = NULL, id = NULL, locale, n_max = Inf, skip = 0, na = c("", "NA"), guess_max = min(1000, n_max), show_col_types = TRUE**)** See **?read_delim**



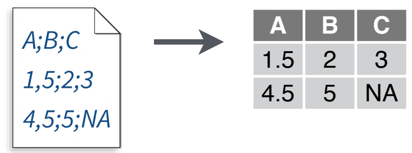
A	B	C
1	2	3
4	5	NA

read_delim("file.txt", delim = "|"**)** Read files with any delimiter. If no delimiter is specified, it will automatically guess.
To make file.txt, run: write_file("A|B|C\n1|2|3\n4|5|NA", file = "file.txt")



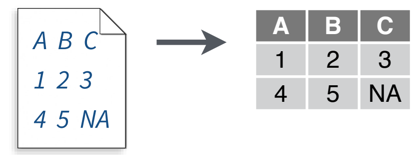
A	B	C
1	2	3
4	5	NA

read_csv("file.csv"**)** Read a comma delimited file with period decimal marks.
write_file("A,B,C\n1,2,3\n4,5,NA", file = "file.csv")



A	B	C
1.5	2	3
4.5	5	NA

read_csv2("file2.csv"**)** Read semicolon delimited files with comma decimal marks.
write_file("A;B;C\n1,5;2;3\n4,5;5;NA", file = "file2.csv")



A	B	C
1	2	3
4	5	NA

read_tsv("file.tsv"**)** Read a tab delimited file. Also **read_table()**.
read_fwf("file.tsv", fwf_widths(c(2, 2, NA))**)** Read a fixed width file.
write_file("A\tB\tC\n1\t2\t3\n4\t5\tNA\n", file = "file.tsv")

USEFUL READ ARGUMENTS

A	B	C
1	2	3
4	5	NA

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

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

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

Read multiple files into a single table
read_csv(c("f1.csv", "f2.csv", "f3.csv"),
  id = "origin_file")

1	2	3
4	5	NA

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

A	B	C
1	2	3

Read a subset of lines
read_csv("file.csv", n_max = 1)

A	B	C
NA	2	3
4	5	NA

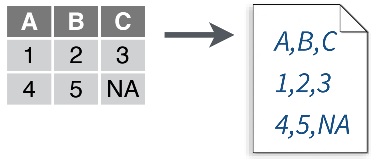
Read values as missing
read_csv("file.csv", na = c("1"))

A;B;C
1,5;2;3,0

Specify decimal marks
read_delim("file2.csv", locale =
  locale(decimal_mark = ","))

Save Data with readr

write_*(x, file, na = "NA", append, col_names, quote, escape, eol, num_threads, progress**)**



A	B	C
1	2	3
4	5	NA

write_delim(x, file, delim = " "**)** Write files with any delimiter.

write_csv(x, file**)** Write a comma delimited file.

write_csv2(x, file**)** Write a semicolon delimited file.

write_tsv(x, file**)** Write a tab delimited file.

One of the first steps of a project is to import outside data into R. Data is often stored in tabular formats, like csv files or spreadsheets.

The front page of this sheet shows how to import and save text files into R using **readr**.

The back page shows how to import spreadsheet data from Excel files using **readxl** or Google Sheets using **googlesheets4**.

OTHER TYPES OF DATA

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

- **haven** - SPSS, Stata, and SAS files
- **DBI** - databases
- **jsonlite** - json
- **xml2** - XML
- **httr** - Web APIs
- **rvest** - HTML (Web Scraping)
- **readr::read_lines()** - text data

Column Specification with readr

Column specifications define what data type each column of a file will be imported as. By default readr will generate a column spec when a file is read and output a summary.

spec(x**)** Extract the full column specification for the given imported data frame.

```
spec(x)
# cols(
#   age = col_integer(),
#   edu = col_character(),
#   earn = col_double()
# )
```

age is an integer

edu is a character

earn is a double (numeric)

COLUMN TYPES

Each column type has a function and corresponding string abbreviation.

- **col_logical()** - "l"
- **col_integer()** - "i"
- **col_double()** - "d"
- **col_number()** - "n"
- **col_character()** - "c"
- **col_factor(**levels, ordered = FALSE**)** - "f"
- **col_datetime(**format = ""**)** - "T"
- **col_date(**format = ""**)** - "D"
- **col_time(**format = ""**)** - "t"
- **col_skip()** - "-", "_"
- **col_guess()** - "?"

USEFUL COLUMN ARGUMENTS

Hide col spec message
read_*(file, show_col_types = FALSE)

Select columns to import
Use names, position, or selection helpers.
read_*(file, col_select = c(age, earn))

Guess column types
To guess a column type, read_ *() looks at the first 1000 rows of data. Increase with **guess_max**.
read_*(file, guess_max = Inf)

DEFINE COLUMN SPECIFICATION

Set a default type
```
read_csv(
  file,
  col_type = list(.default = col_double())
)
```

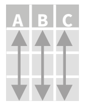
Use column type or string abbreviation
```
read_csv(
  file,
  col_type = list(x = col_double(), y = "l", z = "_")
)
```

Use a single string of abbreviations
```
# col types: skip, guess, integer, logical, character
read_csv(
  file,
  col_type = "_?ilc"
)
```

• info@posit.co • posit.co • **readr.tidyverse.org • readxl.tidyverse.org • googlesheets4.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • readxl 1.4.3 • googlesheets4 1.1.1 • Updated: 2024-05

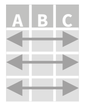
Data transformation with dplyr : : **CHEATSHEET**

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)**

Summarize 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).

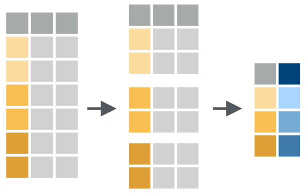
summary function

summarize(.data, ...)
Compute table of summaries.
mtcars |> summarize(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()**, **add_count()**, **add_tally()**.
mtcars |> count(cyl)

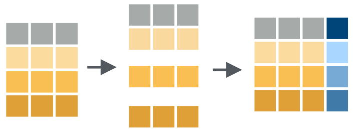
Group Cases

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) |>
  summarize(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.



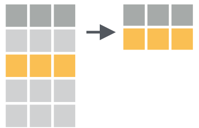
```
starwars |>
  rowwise() |>
  mutate(film_count = length(films))
```

ungroup(x, ...) Returns ungrouped copy of table.
g_mtcars <- mtcars |> group_by(cyl)
ungroup(g_mtcars)

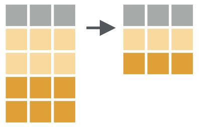
Manipulate Cases

EXTRACT CASES

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

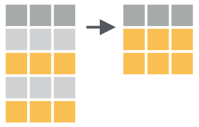


filter(.data, ..., .preserve = FALSE) Extract rows that meet logical criteria.
mtcars |> filter(mpg > 20)

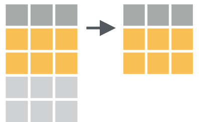


distinct(.data, ..., .keep_all = FALSE) Remove rows with duplicate values.
mtcars |> distinct(gear)

slice(.data, ..., .preserve = FALSE) Select rows by position.
mtcars |> slice(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.
mtcars |> slice_sample(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.
mtcars |> slice_min(mpg, prop = 0.25)

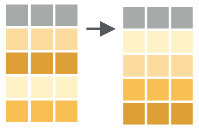
slice_head(.data, ..., n, prop) and **slice_tail()** Select the first or last rows.
mtcars |> slice_head(n = 5)

Logical and boolean operators to use with filter()

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

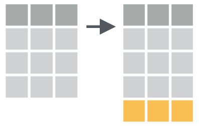
See **?base::Logic** and **?Comparison** for help.

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.
mtcars |> arrange(mpg)
mtcars |> arrange(desc(mpg))

ADD CASES

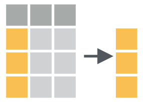


add_row(.data, ..., .before = NULL, .after = NULL) Add one or more rows to a table.
cars |> add_row(speed = 1, dist = 1)

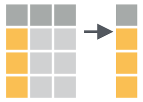
Manipulate Variables

EXTRACT VARIABLES

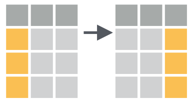
Column functions return a set of columns as a new vector or table.



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



select(.data, ...) Extract columns as a table.
mtcars |> select(mpg, wt)



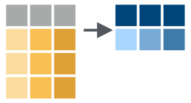
relocate(.data, ..., .before = NULL, .after = NULL) Move columns to new position.
mtcars |> relocate(mpg, cyl, .after = last_col())

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

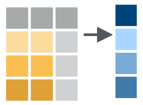
contains(match)	**num_range(**prefix, range)	**:**, e.g., mpg:cyl
ends_with(match)	**all_of(**x)/**any_of(**x, ..., vars)	**!**, e.g., !gear
starts_with(match)	**matches(**match)	**everything()**

MANIPULATE MULTIPLE VARIABLES AT ONCE

df <- tibble(x_1 = c(1, 2), x_2 = c(3, 4), y = c(4, 5))



across(.cols, .funs, ..., .names = NULL) Summarize or mutate multiple columns in the same way.
df |> summarize(across(everything(), mean))



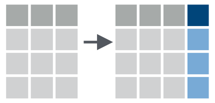
c_across(.cols) Compute across columns in row-wise data.

```
df |>
  rowwise() |>
  mutate(x_total = sum(c_across(1:2)))
```

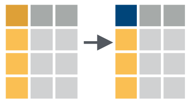
MAKE NEW VARIABLES

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

vectorized function



mutate(.data, ..., .keep = "all", .before = NULL, .after = NULL) Compute new column(s). Also **add_column()**.
mtcars |> mutate(gpm = 1 / mpg)
mtcars |> mutate(gpm = 1 / mpg, .keep = "none")



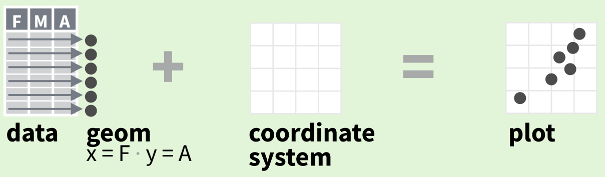
rename(.data, ...) Rename columns. Use **rename_with()** to rename with a function.
mtcars |> rename(miles_per_gallon = mpg)

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **dplyr.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • dplyr 1.1.4 • Updated: 2024-05

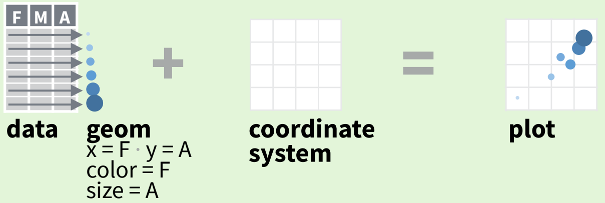
Data visualization with ggplot2 : : CHEATSHEET

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.

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.

Aes

Common aesthetic values.

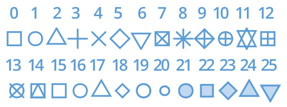
color and **fill** - string ("red", "#RRGGBB")

linetype - integer or string (0 = "blank", 1 = "solid", 2 = "dashed", 3 = "dotted", 4 = "dotdash", 5 = "longdash", 6 = "twodash")

size - integer (in mm for size of points and text)

linewidth - integer (in mm for widths of lines)

shape - integer/shape name or a single character ("a")



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() and **a + expand_limits()**
Ensure limits include values across all plots.

b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1**)** - 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(alpha = 50)**)** - x, y, alpha, color, fill, group, subgroup, 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

both continuous

e <- ggplot(mpg, aes(cty, hwy))

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

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) - x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

one discrete, one continuous

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

both discrete

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

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

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

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_density_2d()
x, y, alpha, color, group, linetype, size

h + geom_hex()
x, y, alpha, color, 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

Draw the appropriate geometric object depending on the simple features present in the data. aes() arguments: map_id, alpha, color, fill, linetype, linewidth.

nc <- **sf::st_read(**system.file("shape/nc.shp", package = "sf")**)**

ggplot(nc) +
geom_sf(aes(fill = AREA)**)**

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, color, group, linetype, size, weight

l + geom_contour_filled(aes(fill = z)**)**
x, y, alpha, color, fill, group, linetype, size, subgroup

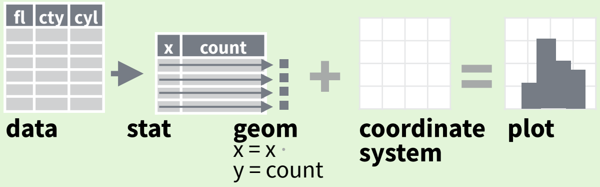
l + geom_raster(aes(fill = z), hjust = 0.5, vjust = 0.5, interpolate = FALSE**)**
x, y, alpha, fill

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

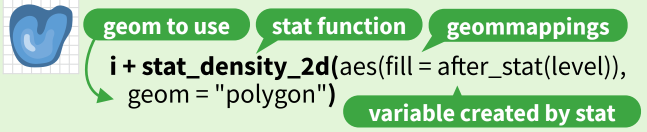
CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **ggplot2.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • ggplot2 3.5.1 • Updated: 2024-05

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 **after_stat(name)** syntax to map the stat variable **name** to an aesthetic.



i + stat_density_2d(aes(fill = after_stat(level)),
geom = "polygon"**)**

c + stat_bin(binwidth = 1, boundary = 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() + xlim(-5, 5**) + stat_function(**fun = dnorm,
n = 20, geom = "point"**) x** | x, y

ggplot() + stat_qq(aes(sample = 1:100)**)**
x, y, sample | sample, theoretical

e + stat_sum() x, y, size | n, prop

e + stat_summary(fun.data = "mean_cl_boot"**)**

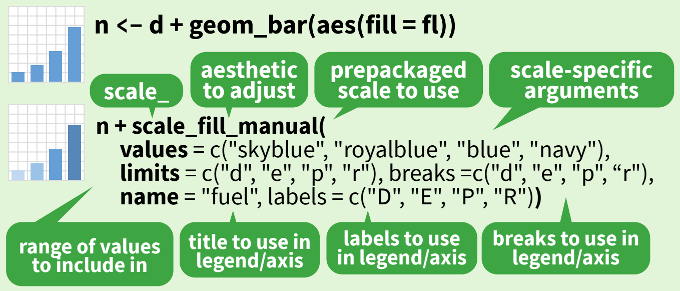
h + stat_summary_bin(fun = "mean", geom = "bar"**)**

e + stat_identity()

e + stat_unique()

Scales Override defaults with **scales** package.

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


scale_
aesthetic to adjust
prepackaged scale to use
scale-specific arguments
range of values to include in
title to use in legend/axis
labels to use in legend/axis
breaks to use in legend/axis

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

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")**)**

GENERAL PURPOSE SCALES

Use with most aesthetics

scale_*_continuous() - Map cont’ values to visual ones.

scale_*_discrete() - Map discrete values to visual ones.

scale_*_binned() - Map continuous values to discrete bins.

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 values as date times.
Same as scale_*_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 the direction of the x axis.
scale_x_sqrt() - Plot x on square root scale.

COLOR AND FILL SCALES (DISCRETE)

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"**)**

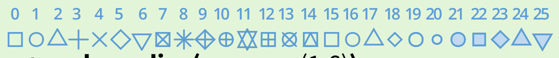
o + scale_fill_gradient2(low = "red", high = “blue”,
mid = "white", midpoint = 25**)**

o + scale_fill_gradientn(colors = 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)**)**



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()
Flip cartesian coordinates by switching x and y aesthetic mappings.

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

r + coord_trans(y = “sqrt")**)** - x, y, xlim, ylim
Transformed cartesian coordinates. Set xtrans and ytrans to the name of a window function.

π + coord_sf() - xlim, ylim, crs. Ensures all layers use a common Coordinate Reference System.

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 theme.

r + theme_void()
Empty theme.

r + theme() Customize aspects of the theme such as axis, legend, panel, and facet properties.
r + labs(title = “Title”) + theme(plot.title.position = “plot”)
r + theme(panel.background = element_rect(fill = “blue”))

Faceting

Facets divide a plot into subplots based on the values of one or more discrete variables.

t <- ggplot(mpg, aes(cty, hwy)) + geom_point()

t + facet_grid(. ~ fl)
Facet into columns based on fl.

t + facet_grid(year ~ .)
Facet into rows based on year.

t + facet_grid(year ~ fl)
Facet into both rows and columns.

t + facet_wrap(~ fl)
Wrap facets into a rectangular layout.

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

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

Set **labeller** to adjust facet label:

t + facet_grid(. ~ fl, labeller = label_both)

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

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

α^c	α^d	α^e	α^p	α^r

Labels and Legends

Use **labs()** to label the elements of your plot.

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",
alt = "Add alt text to the plot",
<AES> = "New <AES> legend title"**)**

t + annotate(geom = "text", x = 8, y = 9, label = “A"**)**
Places a geom with manually selected aesthetics.

p + guides(x = guide_axis(n.dodge = 2)**)** Avoid crowded or overlapping labels with guide_axis(n.dodge or angle).

n + guides(fill = “none"**)** Set legend type for each aesthetic: colorbar, legend, or none (no legend).

n + theme(legend.position = "bottom"**)**
Place legend at "bottom", "top", "left", or “right”.

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)**)**

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **ggplot2.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • ggplot2 3.5.1 • Updated: 2024-05