posit conf(2025)

CHEATSHEETS

Welcome to

posit conf(2025)

A polyglot community of innovators & transformers

Posit's mission is to create free and open-source software for data science, scientific research, and technical communication.

At posit::conf(2025), we celebrate Posit's enduring mission to enhance the production and consumption of knowledge for everyone, fostering collaboration and reproducible research critical to science, education, government, and industry. This year's theme, "A polyglot community of innovators and transformers," highlights our commitment to supporting data scientists who leverage diverse programming languages like R, Python, and Julia to drive data-driven decisions and transformative progress worldwide.

Posit drives data science forward through a virtuous cycle, where widespread adoption of our open-source tools like RStudio, Shiny, and Quarto creates demand for commercial software such as Posit Team, which allows us to then reinvest in open source. Posit Team, including Posit Workbench, Posit Connect, and Posit Package Manager, enables organizations to confidently develop and iterate quickly in R and Python, while ensuring compliance with security standards. Our solution–recognized by awards like the 2025 Databricks Developer Tools Partner of the Year–balances the needs of data scientists to develop quickly while working with the tools they want, with the security and governance needs of the organization.

A major highlight this year is the introduction of Positron, our new AI-driven Integrated Development Environment that enables data scientists to effectively work with Python, R, and other languages within a single, unified interface, streamlining development workflows and reducing context switching. For data scientists, by data scientists.

posit

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

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

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)**)** - 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)**)** - 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

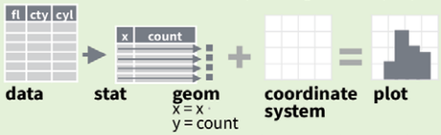
l + geom_contour_filled(aes(fill = z)**)**
x, y, alpha, color, fill, group, linetype, size, subgroup

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

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.

geom to use | stat function | geommappings

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

variable created by stat

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.

n <– d + geom_bar(aes(fill = fl))

scale | aesthetic to adjust | prepackaged scale to use | scale-specific arguments

n + scale_fill_manual(
values = c("skyblue", "royalblue", "blue", "navy"),
limits = c("d", "e", "p", "r"), breaks =c("d", "e", "p", "r"),
name = "fuel", labels = c("D", "E", "P", "R"))

range of values to include in mapping | title to use in legend/axis | labels to use in legend/axis | breaks to use in legend/axis

GENERAL PURPOSE SCALES

Use with most aesthetics

scale_*_continuous() - Map cont' values to visual ones.
scale_*_discrete() - Map discrete values to visual ones.
scale_*_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)**)**

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

p + scale_radius(range = c(1,6)**)**
p + scale_size_area(max_size = 6**)**

Coordinate Systems

r <– d + geom_bar()

r + coord_cartesian(xlim = c(0, 5)**)** - xlim, ylim
The default cartesian coordinate system.

r + coord_fixed(ratio = 1/2**)**
ratio, xlim, ylim - Cartesian coordinates with fixed aspect ratio between x and y units.

r + coord_flip()
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 3.5.2 • Updated: 2025-08

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.

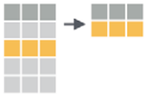
```
starwars |>
  rowwise() |>
  mutate(film_count = length(films))
```

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

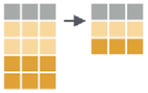
Manipulate Cases

EXTRACT CASES

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

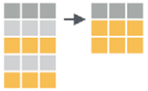


filter(.data, …, .preserve = FALSE**)** Extract rows that meet logical criteria.
mtcars |> filter(mpg > 20)

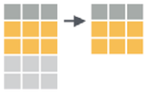


distinct(.data, …, .keep_all = FALSE**)** Remove rows with duplicate values.
mtcars |> distinct(gear)

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)

Vectorized Functions

TO USE WITH MUTATE ()

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

vectorized function

OFFSET

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

CUMULATIVE AGGREGATE

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

RANKING

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

MATH

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

MISCELLANEOUS

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

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

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

Summary Functions

TO USE WITH SUMMARIZE ()

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

summary function

COUNT

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

POSITION

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

LOGICAL

mean() - proportion of TRUEs
sum() - # of TRUEs

ORDER

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

RANK

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

SPREAD

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

Row Names

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

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

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

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

Combine Tables

COMBINE VARIABLES

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

COMBINE CASES

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

RELATIONAL DATA

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

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

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

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

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

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

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

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

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

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

COLUMN MATCHING FOR JOINS

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

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

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

SET OPERATIONS

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

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

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

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

Data 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

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

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

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

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

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 = "_")
)
```

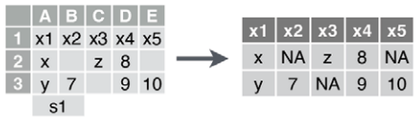
Use a single string of abbreviations

```
# col types: skip, guess, integer, logical, character
read_csv(
  file,
  col_type = "_?ilc"
)
```

Import Spreadsheets

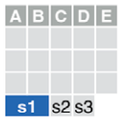
with readxl

READ EXCEL FILES



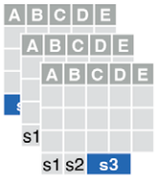
read_excel(path, sheet = NULL, range = NULL**)** Read a .xls or .xlsx file based on the file extension. See front page for more read arguments. Also **read_xls()** and **read_xlsx()**.
read_excel("excel_file.xlsx")

READ SHEETS



read_excel(path, **sheet = NULL)** Specify which sheet to read by position or name.
read_excel(path, sheet = 1)
read_excel(path, sheet = "s1")

excel_sheets(path**)** Get a vector of sheet names.
excel_sheets("excel_file.xlsx")



To **read multiple sheets:**

1. Get a vector of sheet names from the file path.
2. Set the vector names to be the sheet names.
3. Use purrr::map() and purrr::list_rbind() to read multiple files into one data frame.

```
path <- "your_file_path.xlsx"
path |>
  excel_sheets() |>
  set_names() |>
  map(read_excel, path = path) |>
  list_rbind()
```

OTHER USEFUL EXCEL PACKAGES

For functions to write data to Excel files, see:

- **openxlsx**
- **writexl**

For working with non-tabular Excel data, see:

- **tidyxl**

READXL COLUMN SPECIFICATION

Column specifications define what data type each column of a file will be imported as.

Use the **col_types** argument of **read_excel()** to set the column specification.

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

Set all columns to same type, e.g. character
read_excel(path, col_types = "text")

Set each column individually

```
read_excel(
  path,
  col_types = c("text", "guess", "guess","numeric")
)
```

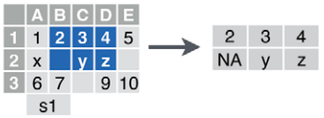
COLUMN TYPES

logical	numeric	text	date	list
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip
- guess
- logical
- numeric
- text
- date
- list

Use **list** for columns that include multiple data types. See **tidyr** and **purrr** for list-column data.

CELL SPECIFICATION FOR READXL AND GOOGLESHEETS4

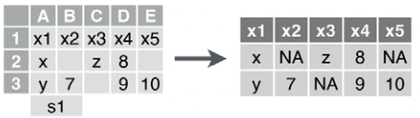


Use the **range** argument of **readxl::read_excel()** or **googlesheets4::read_sheet()** to read a subset of cells from a sheet.
read_excel(path, range = "Sheet1!B1:D2")
read_sheet(ss, range = "B1:D2")

Also use the range argument with cell specification functions **cell_limits()**, **cell_rows()**, **cell_cols()**, and **anchored()**.

with googlesheets4

READ SHEETS



read_sheet(ss, sheet = NULL, range = NULL**)** Read a sheet from a URL, a Sheet ID, or a dribble from the googledrive package. See front page for more read arguments. Same as **range_read()**.

SHEETS METADATA

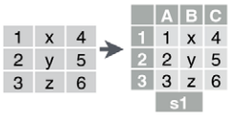
URLs are in the form:
https://docs.google.com/spreadsheets/d/**SPREADSHEET_ID**/edit#gid=**SHEET_ID**

gs4_get(ss**)** Get spreadsheet meta data.

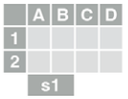
gs4_find(...) Get data on all spreadsheet files.

sheet_properties(ss**)** Get a tibble of properties for each worksheet. Also **sheet_names()**.

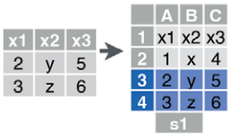
WRITE SHEETS



write_sheet(data, ss = NULL, sheet = NULL**)** Write a data frame into a new or existing Sheet.



gs4_create(name, ..., sheets = NULL**)** Create a new Sheet with a vector of names, a data frame, or a (named) list of data frames.



sheet_append(ss, data, sheet = 1**)** Add rows to the end of a worksheet.

GOOGLESHEETS4 COLUMN SPECIFICATION

Column specifications define what data type each column of a file will be imported as.

Use the **col_types** argument of **read_sheet()/ range_read()** to set the column specification.

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

Set all columns to same type, e.g. character
read_sheet(path, col_types = "c")

Set each column individually
# col types: skip, guess, integer, logical, character
read_sheets(ss, col_types = "_?ilc")

COLUMN TYPES

l	n	c	D	L
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

- skip - "_" or "-"
- guess - "?"
- logical - "l"
- integer - "i"
- double - "d"
- numeric - "n"
- date - "D"
- datetime - "T"
- character - "c"
- list-column - "L"
- cell - "C" Returns list of raw cell data.

Use list for columns that include multiple data types. See **tidyr** and **purrr** for list-column data.

FILE LEVEL OPERATIONS

googlesheets4 also offers ways to modify other aspects of Sheets (e.g. freeze rows, set column width, manage (work)sheets). Go to **googlesheets4.tidyverse.org** to read more.

For whole-file operations (e.g. renaming, sharing, placing within a folder), see the tidyverse package **googledrive** at **googledrive.tidyverse.org**.

CC BY SA Posit Software, PBC • info@posit.co • posit.co • **readr.tidyverse.org** • **readxl.tidyverse.org** • **googlesheets4.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • readxl 1.4.5 • googlesheets4 1.1.1 • Updated: 2025-08

Data tidying with tidyr : : CHEATSHEET

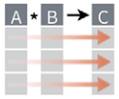
Tidy data is a way to organize tabular data in a consistent data structure across packages.
A table is tidy if:

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"))

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

Both make this tibble

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

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

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

is_tibble(x) Test whether x is a tibble.

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

table4a

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

→

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

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

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

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

table2

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

→

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

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

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

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

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

table5

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

→

country	year
A	1999
A	2000
B	1999
B	2000

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

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

table3

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

→

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

separate_wider_delim(data, cols, delim, ..., names = NULL, names_sep = NULL, names_repair = "check unique", too_few, too_many, cols_remove = TRUE) Separate each cell in a column into several columns. Also **separate_wider_regex()** and **separate_wider_position()**.

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

table3

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

→

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

separate_longer_delim(data, cols, delim, .., width, keep_eampty) Separate each cell in a column into several rows.

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

Expand Tables

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

x

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

→

x1	x2
A	1
A	2
B	1
B	2

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

x

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

→

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

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

Handle Missing Values

Drop or replace explicit missing values (NA).

x

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

→

x1	x2
A	1
D	3

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

x

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

→

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

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

x

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

→

x1	x2
A	1
B	2
C	2
D	3
E	2

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

Nested Data

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

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

CREATE NESTED DATA

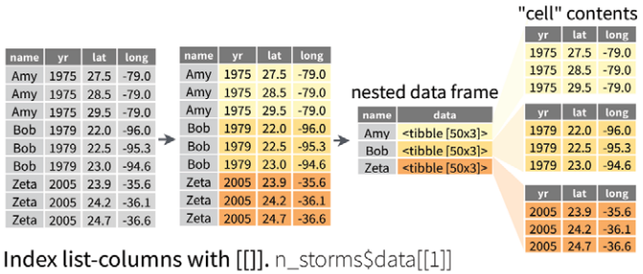
nest(data, …**)** Moves groups of cells into a list-column of a data frame. Use alone or with dplyr::**group_by()**:

1. Group the data frame with **group_by()** and use **nest()** to move the groups into a list-column.

```
n_storms <- storms |>
  group_by(name) |>
  nest()
```

2. Use **nest(new_col = c(x, y))** to specify the columns to group using dplyr::**select()** syntax.

```
n_storms <- storms |>
  nest(data = c(year:long))
```



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

CREATE TIBBLES WITH LIST-COLUMNS

tibble::**tribble(**…**)** Makes list-columns when needed.

```
tribble( ~max, ~seq,
         3,    1:3,
         4,    1:4,
         5,    1:5)
```

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

tibble::**tibble(**…**)** Saves list input as list-columns.

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

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

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

OUTPUT LIST-COLUMNS FROM OTHER FUNCTIONS

dplyr::**mutate(), transmute(),** and **summarise()** will output list-columns if they return a list.

```
mtcars |>
  group_by(cyl) |>
  summarise(q = list(quantile(mpg)))
```

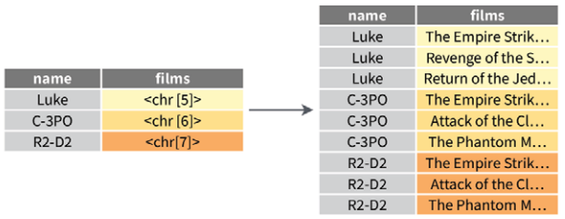
RESHAPE NESTED DATA

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

```
n_storms |> unnest(data)
```

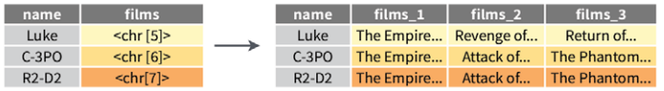
unnest_longer(data, col, values_to = NULL, indices_to = NULL**)** Turn each element of a list-column into a row.

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



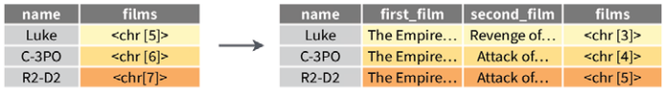
unnest_wider(data, col**)** Turn each element of a list-column into a regular column.

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



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

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



TRANSFORM NESTED DATA

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

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

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

```
n_storms |>
  rowwise() |>
  mutate(n = list(dim(data)))
```

dim() returns two values per row

wrap with list to tell mutate to create a list-column

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

```
n_storms |>
  rowwise() |>
  mutate(n = nrow(data))
```

nrow() returns one integer per row

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

```
starwars |>
  rowwise() |>
  mutate(transport = list(append(vehicles, starships)))
```

append() returns a list for each row, so col type must be list

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

```
starwars |>
  rowwise() |>
  mutate(n_transports = length(c(vehicles, starships)))
```

length() returns one integer per row

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

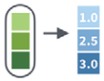
CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **tidyr.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • tidyr 1.3.1 • tibble 3.3.0 • Updated: 2025–08

Apply functions with purrr : : **CHEATSHEET**

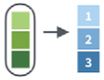
Map Functions

ONE LIST

map(.x, .f, …**)** Apply a function to each element of a list or vector, and return a list.
x <- list(a = 1:10, b = 11:20, c = 21:30)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
map(l1, sort, decreasing = TRUE)



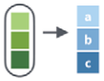
map_dbl(.x, .f, …**)**
Return a double vector.
map_dbl(x, mean)



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

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

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



map_vec(.x, .f, ...**)**
Return a vector that is of the simplest common type.
map_vec(l1, paste, collapse = "")

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

TWO LISTS

map2(.x, .y, .f, …**)** Apply a function to pairs of elements from two lists or vectors, return a list.
y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y,\(x, y) x*y)

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

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

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

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

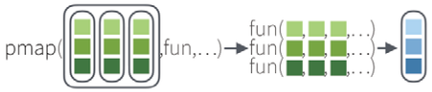
map2_vec(.x, .f, ...**)**
Return a vector that is of the simplest common type.
map2_vec(l1, l2, paste, collapse = ",", sep = ":")

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

imap(.x, .f, ...) is shorthand for **map2(.x, names(.x), .f)** or **map2(.x, seq_along(.x), .f)** depending on whether **.x** is named or not.

MANY LISTS

pmap(.l, .f, …**)** Apply a function to groups of elements from a list of lists or vectors, return a list.
pmap(
list(x, y, z),
function(first, second, third) first * (second + third)
)



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

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

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

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

pmap_vec(.l, .f, …**)**
Return a vector that is of the simplest common type.
pmap_vec(list(l1, l2), paste, collapse = ",", sep = ":")

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

Function Shortcuts

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

map(l, \(x) x + 2)
becomes
map(l, function(x) x + 2)

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

map2(l, p, \(x, y) x + y)
becomes
map2(l, p, function(l, p) l + p)

Use **\(x, y, z)** etc with functions like **pmap()** that have many arguments.

pmap(list(x, y, z), \(x, y, z) x + y / z)
becomes
pmap(list(x, y, z), function(x, y, z) x * (y + z))

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

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

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

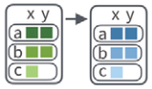
purrr

Modify

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

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

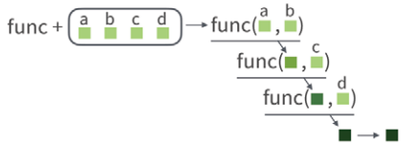
modify_if(.x, .p, .f, ...**)** Apply a function to elements that pass a test. Also **map_if()**.
modify_if(x, is.numeric,~.+2)



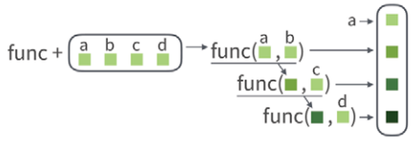
modify_depth(.x, .depth, .f, ...**)** Apply function to each element at a given level of a list. Also **map_depth()**.
modify_depth(x, 1, ~.+ 2)

Reduce

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



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



Vectors

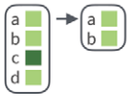
compact(.x, .p = identity**)**
Discard empty elements.
compact(x)

keep_at(x, at**)**
Keep/discard elements based by name or position. Conversely, **discard_at()**.
keep_at(x, "a")

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)

Predicate functions

keep(.x, .p, ...**)**
Keep elements that pass a logical test. Conversely, **discard()**.
keep(x, is.numeric)



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

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

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

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

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

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

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

Pluck

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

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

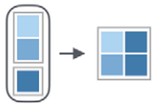
modify_in(.x, .where, .f**)** Apply a function to a value at a selected location.
modify_in(x, "b", abs)
x |> modify_in("b", abs)

Concatenate

```
x1 <- list(a = 1, b = 2, c = 3)
x2 <- list(
  a = data.frame(x = 1:2),
  b = data.frame(y = "a")
)
```

list_c(x**)** Combines elements into a vector by concatenating them together.
list_c(x1)

list_rbind(x**)** Combines elements into a data frame by row-binding them together.
list_rbind(x2)



list_cbind(x**)** Combines elements into a data frame by column-binding them together.
list_cbind(x2)

Reshape

list_flatten(.x**)** Remove a level of indexes from a list.
list_flatten(x)

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

List-Columns

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

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

WORK WITH LIST-COLUMNS

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

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

```
starwars |>
  transmute(ships = map2(vehicles,
                         starships,
                         append))
```

list function, return list — list-columns — column function

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

```
starwars |>
  mutate(n_films = map_int(films, length))
```

list function, return int — column function — list-column

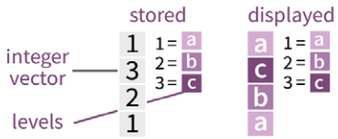
CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **purrr.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • purrr 1.1.0 • Updated: 2025-08

Factors with forcats : : CHEATSHEET

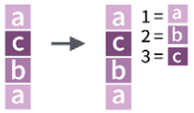
The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors

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

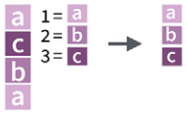


Create a factor with factor()



factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA) Convert a vector to a factor. Also **as_factor()**.

```
f <- factor(c("a", "c", "b", "a"),
            levels = c("a", "b", "c"))
```

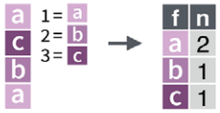


Return its levels with levels()

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

Use unclass() to see its structure

Inspect Factors

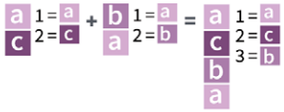


fct_count(f, sort = FALSE, prop = FALSE) Count the number of values with each level. fct_count(f)

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

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

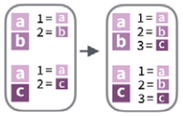
Combine Factors



fct_c(…) Combine factors with different levels. Also **fct_cross()**.

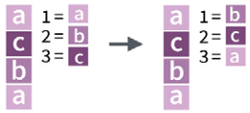
```
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
```

fct_c(f1, f2)



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

Change the order of levels



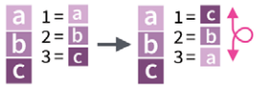
fct_relevel(.f, ..., after = 0L) Manually reorder factor levels. fct_relevel(f, c("b", "c", "a"))

fct_infreq(f, ordered = NA) Reorder levels by the frequency in which they appear in the data (highest frequency first). Also **fct_inseq()**.

```
f3 <- factor(c("c", "c", "a"))
```

fct_infreq(f3)

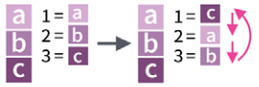
fct_inorder(f, ordered = NA) Reorder levels by order in which they appear in the data. fct_inorder(f2)



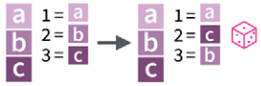
fct_rev(f) Reverse level order.

```
f4 <- factor(c("a","b","c"))
```

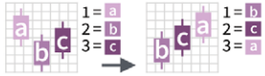
fct_rev(f4)



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

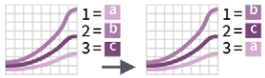


fct_shuffle(f, n = 1L) Randomly permute order of factor levels. fct_shuffle(f4)



fct_reorder(.f, .x, .fun = median, ..., .desc = FALSE) Reorder levels by their relationship with another variable.

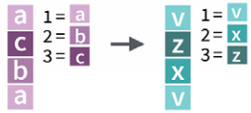
```
boxplot(
  PlantGrowth,
  weight ~ fct_reorder(group, weight)
)
```



fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE) Reorder levels by their final values when plotted with two other variables.

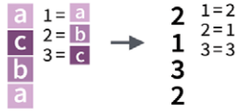
```
ggplot(
  diamonds,
  aes(
    carat, price,
    color = fct_reorder2(color, carat, price)
  )) +
  geom_smooth()
```

Change the value of levels



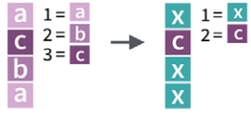
fct_recode(.f, ...) Manually change levels. Also **fct_relabel()** which obeys purrr::map syntax to apply a function or expression to each level.

fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))



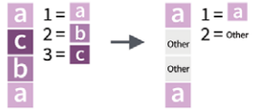
fct_anon(f, prefix = "") Anonymize levels with random integers.

fct_anon(f)



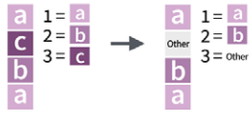
fct_collapse(.f, …, other_level = NULL) Collapse levels into manually defined groups.

fct_collapse(f, x = c("a", "b"))



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

fct_lump_min(f, min = 2)



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

fct_other(f, keep = c("a", "b"))

Add or drop levels

fct_drop(f, only) Drop unused levels.

```
f5 <- factor(c("a","b"),c("a","b","x"))
f6 <- fct_drop(f5)
```

fct_expand(f, …) Add levels to a factor.

fct_expand(f6, "x")

fct_na_value_to_level(f, level = "(Missing)") Assigns a level to NAs to ensure they appear in plots, etc.

f7 <- factor(c("a", "b", NA))
fct_na_value_to_level(f7, level = "(Missing)")

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **forcats.tidyverse.org** • Diagrams inspired by **@LVaudor** on Twitter • HTML cheatsheets at **pos.it/cheatsheets** • forcats 1.0.0 • Updated: 2025-08

Positron

The data science code editor

Data scientists work differently. So does Positron.

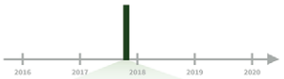
Explore the code editor made for data science and analysis with Python, R, and you.

pos.it/positron

Creators of RStudio

Dates and times with lubridate : : **CHEATSHEET**

Date-times



2017-11-28 12:00:00

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2:01 — hms::**hms()** Also lubridate::**hms(), hm()** and **ms()**, which return periods.* hms::hms(seconds = 0, minutes = 1, hours = 2)

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

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

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

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

parse_date_time() Easier strptime. parse_date_time("09-01-01", "ymd")

GET AND SET COMPONENTS

Use an accessor function to get a component. — d ## "2017-11-28" / day(d) ## 28

Assign into an accessor function to change a component in place. — day(d) <- 1 / d ## "2017-11-01"

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

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

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

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

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

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

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

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

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

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

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

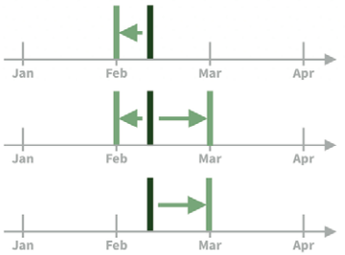
am(x**)** Is it in the am? am(dt) **pm(**x**)** Is it in the pm? pm(dt)

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

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

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

Round Date-times



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

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

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

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

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

Stamp Date-times

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

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

Tip: use a date with day > 12

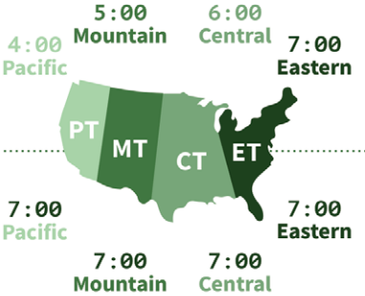
Time Zones

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

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

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

Sys.timezone() Gets current time zone.



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

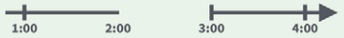
force_tz(time, tzone = ""**)** Get the **same clock time** in a new time zone (a new date-time). Also **force_tzs()**. force_tz(dt, "US/Pacific")

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

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

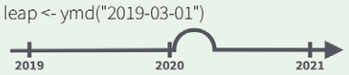
A normal day
nor <- ymd_hms("2018-01-01 01:30:00",tz="US/Eastern")

The start of daylight savings (spring forward)
gap <- ymd_hms("2018-03-11 01:30:00",tz="US/Eastern")



The end of daylight savings (fall back)
lap <- ymd_hms("2018-11-04 00:30:00",tz="US/Eastern")

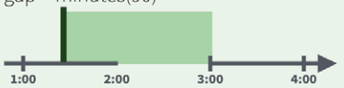
Leap years and leap seconds
leap <- ymd("2019-03-01")



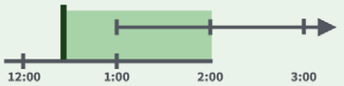
Periods track changes in clock times, which ignore time line irregularities.

nor + minutes(90)

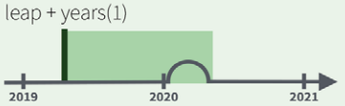
gap + minutes(90)



lap + minutes(90)

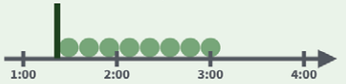


leap + years(1)



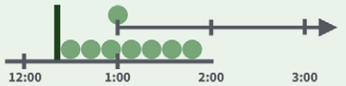
Durations track the passage of physical time, which deviates from clock time when irregularities occur.

nor + dminutes(90)

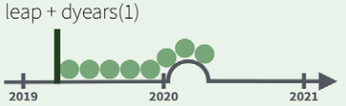


gap + dminutes(90)

lap + dminutes(90)

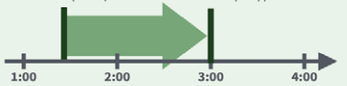


leap + dyears(1)



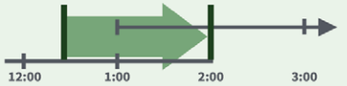
Intervals represent specific intervals of the timeline, bounded by start and end date-times.

interval(nor, nor + minutes(90))

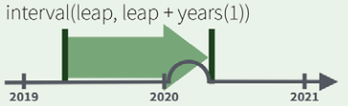


interval(gap, gap + minutes(90))

interval(lap, lap + minutes(90))



interval(leap, leap + years(1))



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

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

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

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

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

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

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

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

PERIODS

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

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

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

Number of months | Number of days | etc.

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

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

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

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

DURATIONS

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

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

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

Exact length in seconds | Equivalent in common units

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

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

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

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

INTERVALS

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

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

Start Date | End Date

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

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

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

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

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

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

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

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

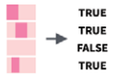
as.interval(x, start, …**)** Coerce a timespan to an interval with the start date-time. Also **is.interval()**. as.interval(days(1), start = now())

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **lubridate.tidyverse.org** • HTML cheatsheets at **pos.it/cheatsheets** • lubridate 1.9.4 • Updated: 2025-08

String manipulation with stringr : : **CHEATSHEET**

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

Detect Matches

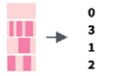


str_detect(string, **pattern**, negate = FALSE**)** Detect the presence of a pattern match in a string. Also **str_like()**. str_detect(fruit, "a")

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

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

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



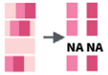
str_count(string, **pattern**) Count the number of matches in a string. str_count(fruit, "a")

Subset Strings

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

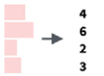
str_subset(string, **pattern**, negate = FALSE**)** Return only the strings that contain a pattern match. str_subset(fruit, "p")

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



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

Manage Lengths



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

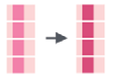
str_pad(string, width, side = c("left", "right", "both"), pad = " "**)** Pad strings to constant width. str_pad(fruit, 17)

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

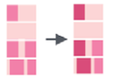
str_trim(string, side = c("both", "left", "right"))** Trim whitespace from the start and/or end of a string. str_trim(str_pad(fruit, 17))

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

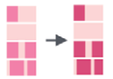
Mutate Strings



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



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



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

A STRING
↓
a string

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

a string
↓
A STRING

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

a string
↓
A String

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

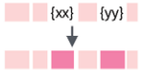
Join and Split

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

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

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

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



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

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

Order Strings

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

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

Helpers

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

```
appl<e>
banana
p<e>ar
```

str_view(string, **pattern**, match = NA**)** View HTML rendering of all regex matches. str_view(sentences, "[aeiou]")

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

This is a long sentence.
↓
This is a long
sentence.

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

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

Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed.*

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes (**""**) or single quotes(**''**).

Some characters cannot be represented directly in an R string . These must be represented as **special characters**, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\	\
\"	"
\n	new line

Run **?"'"** to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

Use **writeLines()** to see how R views your string after all special characters have been parsed.

writeLines("\\.")
 # \.

writeLines("\\ is a backslash")
 # \ is a backslash

INTERPRETATION

Patterns in stringr are interpreted as regexs. To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...**)** Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's , and/or to have . match everything including \n.
str_detect("I", regex("i", TRUE))

fixed() Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). str_detect("\u0130", fixed("i"))

coll() Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow).
str_detect("\u0130", coll("i", TRUE, locale = "tr"))

boundary() Matches boundaries between characters, line_breaks, sentences, or words.
str_split(sentences, boundary("word"))

Regular Expressions -

Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view("abc ABC 123\t.!?\\(){}\n", rx)

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

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

[:space:]
↵ new line

[:blank:]
space
tab

.

[:graph:]

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

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

[:alnum:]

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

[:alpha:]

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

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

ALTERNATES

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

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

ANCHORS

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

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

LOOK AROUNDS

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

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

QUANTIFIERS

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

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

GROUPS

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

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

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

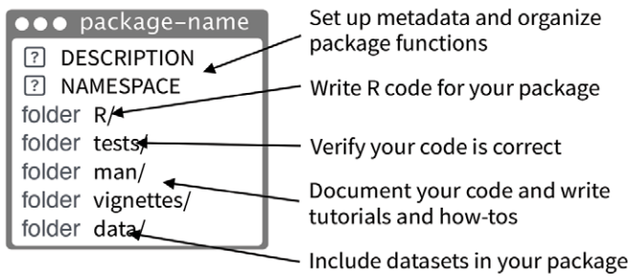
Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

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

Package Development : : **CHEATSHEET**

Package Structure

A package is a convention for organizing files into directories. This cheat sheet shows how to work with the 7 most common parts of an R package:



There are multiple packages useful to package development, including **usethis** which handily automates many of the more repetitive tasks. Install and load **devtools,** which wraps together several of these packages to access everything in one step.

Getting Started

Once per machine:

- Get set up with **use_devtools()** so **devtools** is always loaded in interactive R sessions

```
if (interactive()) {
   require("devtools", quietly = TRUE)
   # automatically attaches usethis
}
```

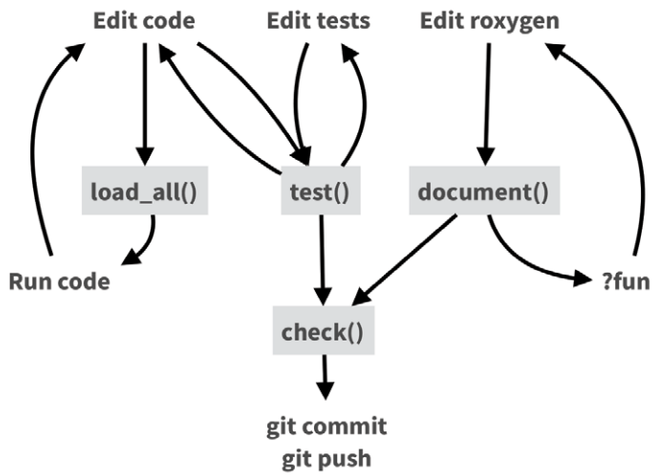
- **create_github_token()** — Set up GitHub credentials
- **git_vaccinate()** — Ignores common special files

Once per package:

- **create_package()** — Create a project with package scaffolding
- **use_git()** — Activate git
- **use_github()** — Connect to GitHub
- **use_github_action()** — Set up automated package checks

Having problems with git? Get a situation report with **git_sitrep().**

Workflow



- **load_all()** (Ctrl/Cmd + Shift + L) — Load code
- **document()** (Ctrl/Cmd + Shift + D) — Rebuild docs and NAMESPACE
- **test()** (Ctrl/Cmd + Shift + T) — Run tests
- **check()** (Ctrl/Cmd + Shift + E) — Check complete package

folder R/

All of the R code in your package goes in folder R/. A package with just an R/ directory is still a very useful package.

☑ Create a new package project with **create_package("path/to/name").**

☑ Create R files with **use_r("file-name").**

- Follow the tidyverse style guide at **style.tidyverse.org**
- Click on a function and press **F2** to go to its definition
- Find a function or file with **Ctrl + .**

? DESCRIPTION

The ? DESCRIPTION file describes your work, sets up how your package will work with other packages, and applies a license.

☑ Pick a license with **use_mit_license(), use_gpl3_license(), use_proprietary_license().**

☑ Add packages that you need with **use_package().**

Import packages that your package requires to work. R will install them when it installs your package.

use_package(x, type = "imports")

Suggest packages that developers of your package need. Users can install or not, as they like.

use_package(x, type = "suggests")

? NAMESPACE

The ? NAMESPACE file helps you make your package self-contained: it won't interfere with other packages, and other packages won't interfere with it.

☑ Export functions for users by placing **@export** in their roxygen comments.

☑ Use objects from other packages with **package::object** or **@importFrom package object** (recommended) or **@import package** (use with caution).

☑ Call **document()** to generate NAMESPACE and **load_all()** to reload.

DESCRIPTION	NAMESPACE
Makes **packages** available	Makes **function** available
Mandatory	Optional (can use :: instead)
use_package()	**use_import_from()**

folder man/

The documentation will become the help pages in your package.

- ☑ Document each function with a roxygen block above its definition in R/. In RStudio, Code > Insert Roxygen Skeleton helps (Ctrl/Cmd + Alt + Shift + R).
- ☑ Document each dataset with roxygen block above the name of the dataset in quotes.
- ☑ Document the package with **use_package_doc().**
- ☑ Build documentation in folder man/ from Roxygen blocks with **document().**

folder vignettes/

- ☑ Create a vignette that is included with your package with **use_vignette().**
- ☑ Create an article that only appears on the website with **use_article().**
- ☑ Write the body of your vignettes in R Markdown.

Websites with pkgdown

- ☑ Use GitHub and **use_pkgdown_github_pages()** to set up pkgdown and configures an automated workflow using GitHub Actions and Pages.
- ☑ If you're not using GitHub, call **use_pkgdown()** to configure pkgdown. Then build locally with **pkgdown::build_site().**

folder tests/

- ☑ Set up test infrastructure with **use_testthat().**
- ☑ Create a test file with **use_test().**
- ☑ Write tests with **test_that()** and **expect_()**.
- ☑ Run all tests with **test()** and run tests for current file with **test_active_file().**
- ☑ See coverage of all files with **test_coverage()** and see coverage of current file with **test_coverage_active_file().**

ROXYGEN2

The **roxygen2** package lets you write documentation inline in your .R files with shorthand syntax.

- Add roxygen documentation as comments beginning with **#'**.
- Place a roxygen **@** tag (right) after **#'** to supply a specific section of documentation.
- Untagged paragraphs will be used to generate a title, description, and details section (in that order).

```
#' Add together two numbers
#'
#' @param x A number.
#' @param y A number.
#' @returns The sum of `x` and `y`.
#' @export
#' @examples
#' add(1, 1)
add <- function(x, y) {
  x + y
}
```

COMMON ROXYGEN TAGS

@description	@family	**@returns**
@examples	@inheritParams	@seealso
@examplesIf	**@param**	
@export	@rdname	

README.Rmd + NEWS.md

- ☑ Create a README and NEWS markdown files with **use_readme_rmd()** and **use_news_md().**

Expect statement	Tests
expect_equal()	Is equal? (within numerical tolerance)
expect_error()	Throws specified error?
expect_snapshot()	Output is unchanged?

```
test_that("Math works", {
  expect_equal(1 + 1, 2)
  expect_equal(1 + 2, 3)
  expect_equal(1 + 3, 4)
})
```

folder data/

- ☑ Record how a data set was prepared as an R script and save that script to folder data-raw/ with **use_data_raw().**
- ☑ Save a prepared data object to folder data/ with **use_data().**

Package States

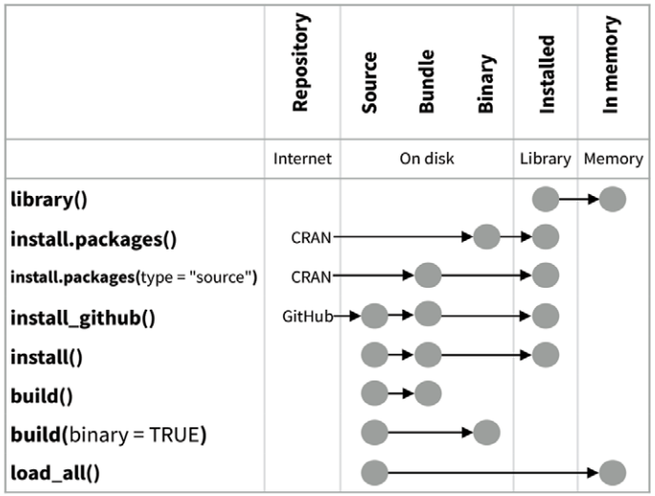
The contents of a package can be stored on disk as a:

- **source** - a directory with sub-directories (as shown in Package structure)
- **bundle** - a single compressed file (*.tar.gz*)
- **binary** - a single compressed file optimized for a specific OS

Packages exist in those states locally or remotely, e.g. on CRAN or on GitHub.

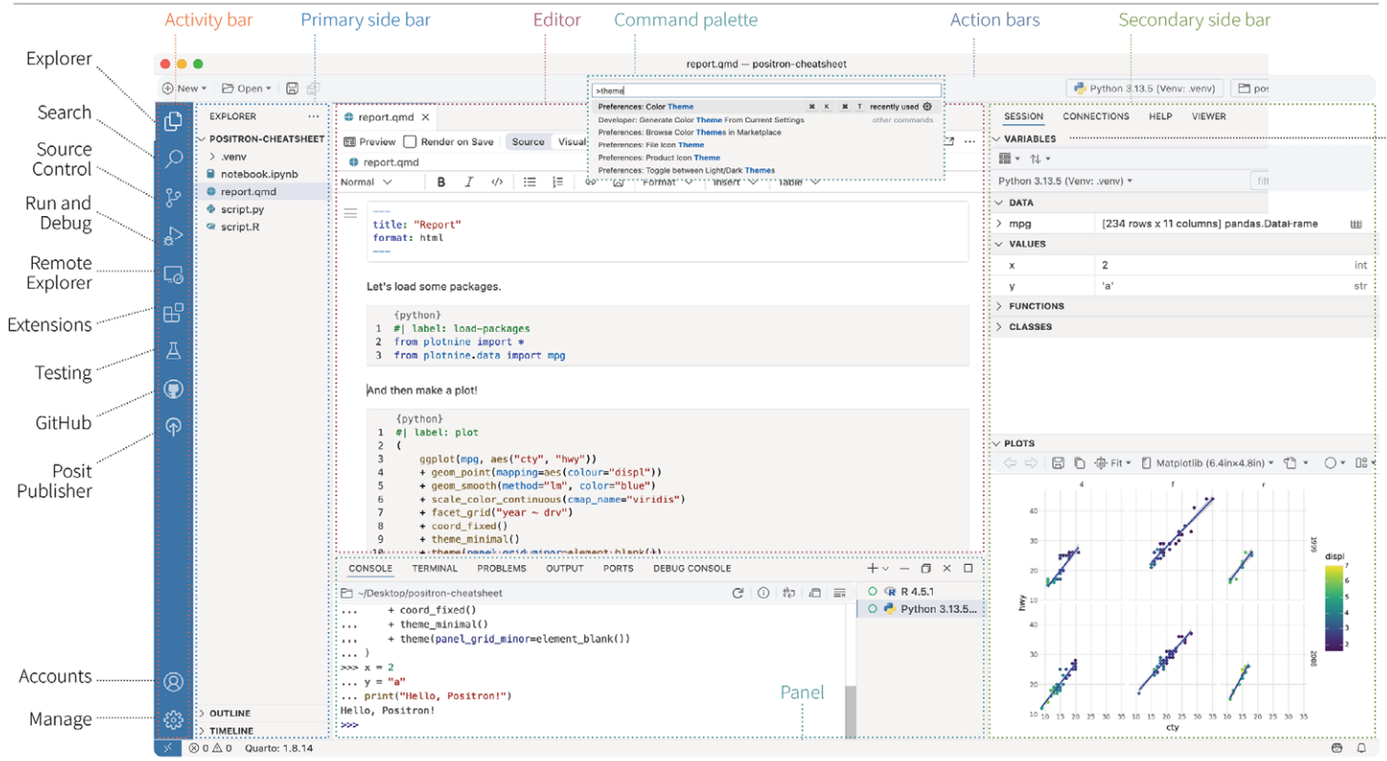
From those states, a package can be installed into an R library and then loaded into memory for use during an R session.

Use the functions below to move between these states.

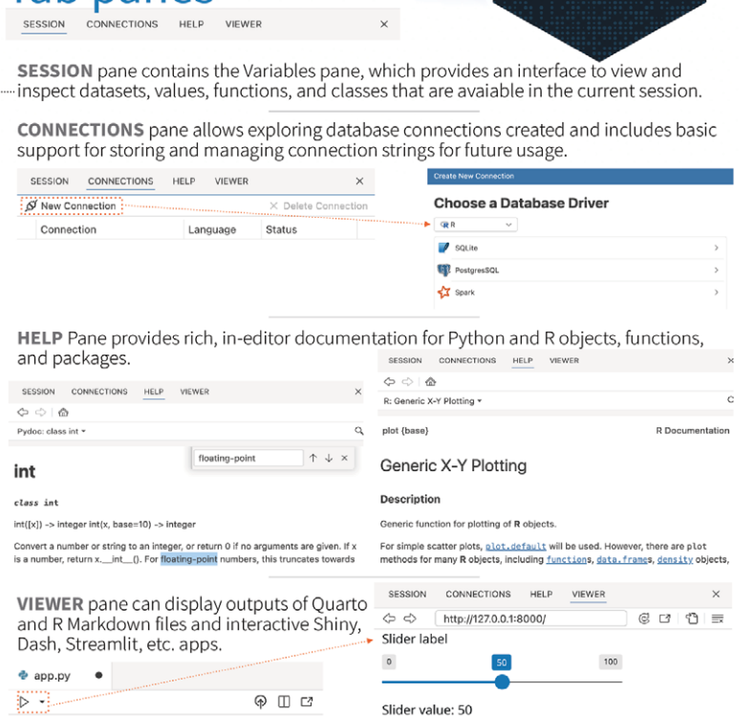


Visit **r-pkgs.org** to learn much more about writing and publishing packages for R.

Positron : : CHEATSHEET



Tab panes



SESSION pane contains the Variables pane, which provides an interface to view and inspect datasets, values, functions, and classes that are avaiable in the current session.

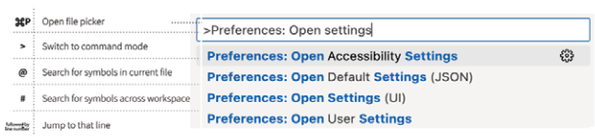
CONNECTIONS pane allows exploring database connections created and includes basic support for storing and managing connection strings for future usage.

HELP Pane provides rich, in-editor documentation for Python and R objects, functions, and packages.

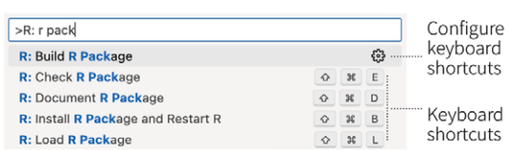
VIEWER pane can display outputs of Quarto and R Markdown files and interactive Shiny, Dash, Streamlit, etc. apps.

Command palette

Fast and powerful way to interact with Positron without taking your hands off the keyboard, press **Cmd/Ctrl⇧P** to open.

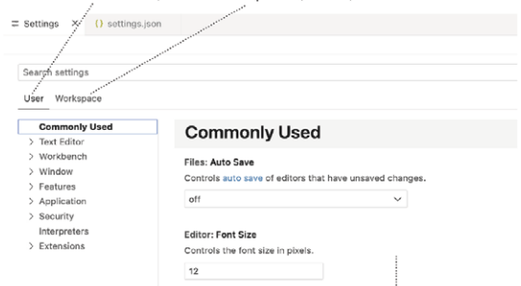


Certain commands are enabled in related contexts, e.g., package development commands when in a package folder.

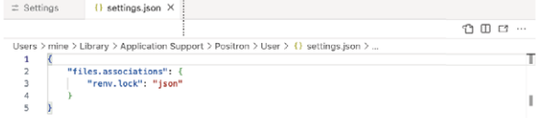


Settings

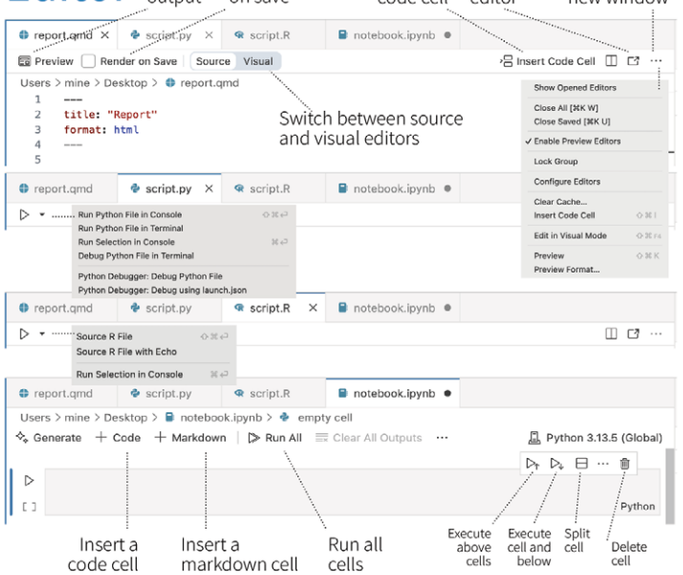
Nearly every part of Positron's editor, UI, and functional behavior has customizable options that can be set at user (globally, for all Positron instances) or at workspace (folder) level.



Settings can be changed by using the Settings editor or editing the settings JSON file.

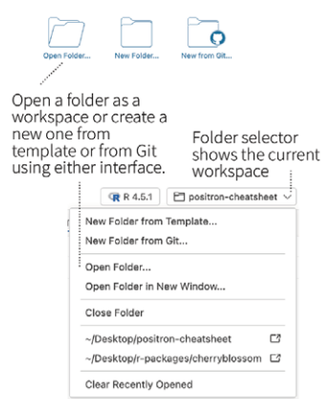


Editor



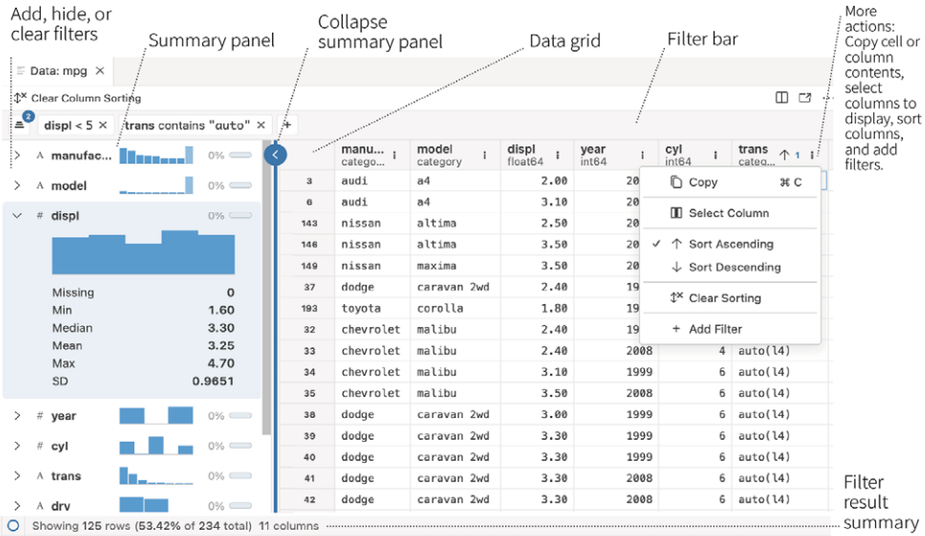
Workspaces and folders

A Positron workspace is a project folder that has been opened in its own window.



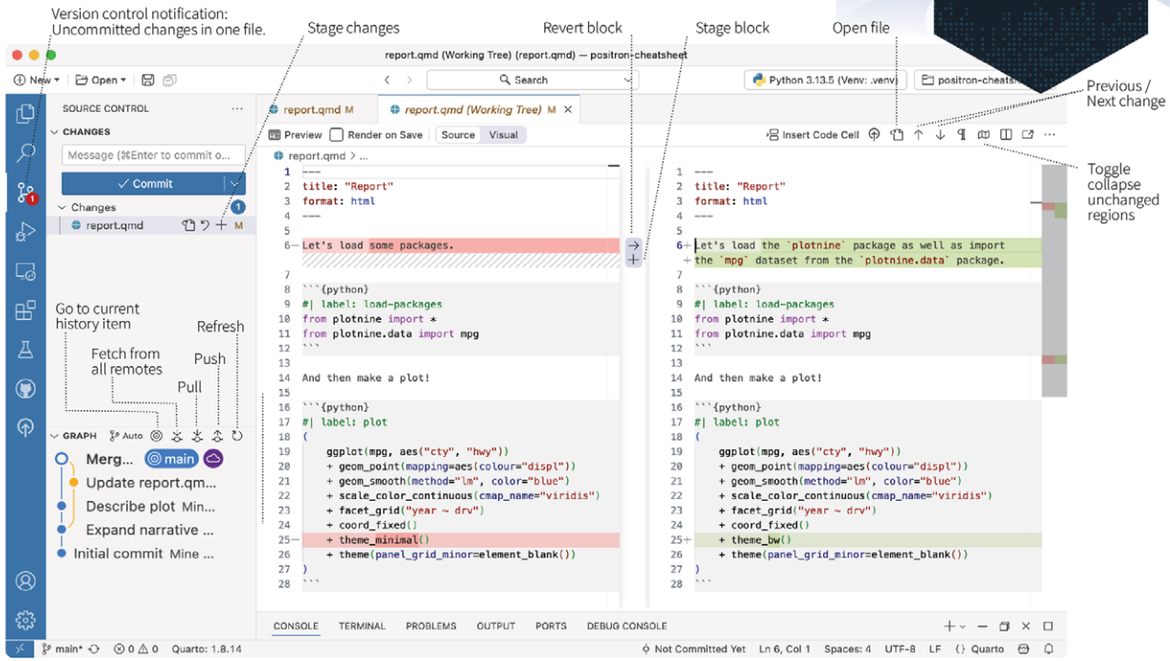
Data explorer

View or explore raw data files (CSV, Parquet, etc.) and dataframes in your Python or R sessions. Open by clicking on the table icon in Variables pane or with **%view** (Python) and **View()** (R).



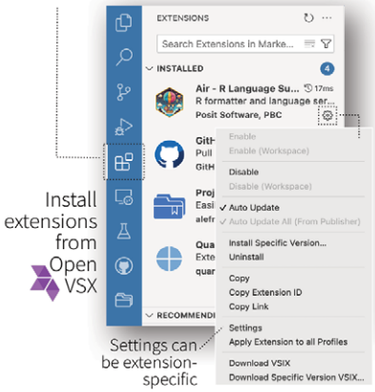
Version control

Built-in Git integration supports the basic Git workflow (make, stage, commit, push changes) as well as working with branches and resolving merge conflicts via its 3-way merge editor.



Extensions

Extensions let you add languages, debuggers, tools, themes, etc. to Positron to support your development workflow.



Built-in extensions (e.g., R, Positron Python, and Markdown Language Basics): installed with Positron, can only be updated by updating Positron.

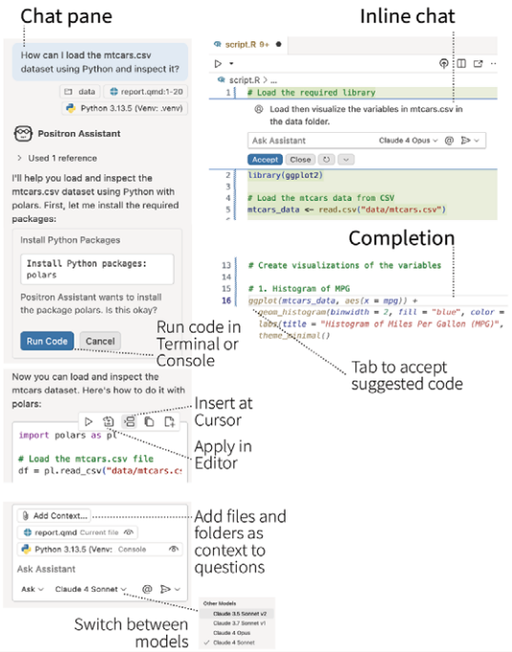
Bootstrapped extensions (e.g., Quarto, Air, GitHub Pull Requests): installed by Positron, act like you installed them yourself, can be updated independently of updating Positron.

Positron Assistant

AI client that provides LLM integration within Positron, both for chat and for inline completions.

Positron › Assistant: Enable *Preview* — Preview feature in Positron 2025.07.0-204 and beyond

☑ Enable Positron Assistant, an AI assistant for Positron.



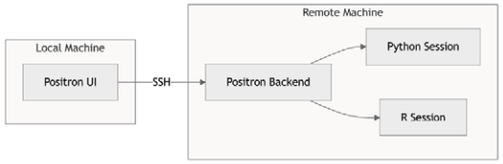
Workbench

Extend Positron with commercial license in Posit Workbench, with key benefits including:

- Centralized management: Centrally manage, configure, and update Positron for all users.
- Enhanced security: User authentication and secure authorization to enterprise data sources.
- Resource management: Run sessions on powerful servers with more CPU, RAM, GPUs, etc. than a laptop.
- Background jobs: Long-running computations executed in the background with detailed audit logs.
- Commercial support: Access professional support from Posit for troubleshooting and guidance.

Run remote jobs

Run Positron IDE's front end on one machine with back end (files, projects, Python and R sessions, etc.) on another machine via ordinary secure shell (SSH) connection.



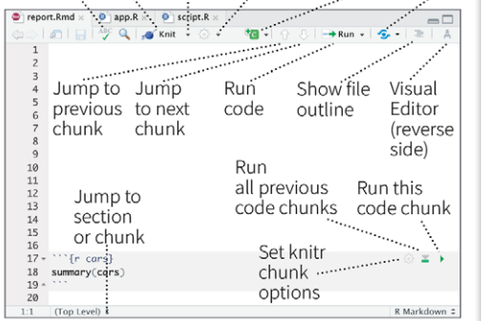
Keyboard shortcuts

Global shortcuts	Windows/Linux	Mac
Run the selected code in the editor; if no code is selected, run the current statement	Ctrl+**Enter**	Cmd+Enter
Restart the interpreter currently open in the Console	Ctrl+Shift+0	Cmd+Shift+0
Run the file open in the editor (using e.g. `source()` or `%run`)	Ctrl+Shift+Enter	Cmd+Shift+Enter
Show contextual help for the topic under the cursor	Ctrl+K or F1	Cmd+R or F1
Focus the Console	Ctrl+K, F	Cmd+K, F
Focus the Variables pane	Ctrl+K, V	Cmd+K, V
Clear the Console	Ctrl+L	Ctrl+L
R shortcuts		
Insert the pipe operator (`|>` or `%>%`)	Ctrl+Shift+M	Cmd+Shift+M
Insert the assignment operator (`<-`)	Alt+-	Alt+-

RStudio IDE : : CHEATSHEET

Documents and Apps

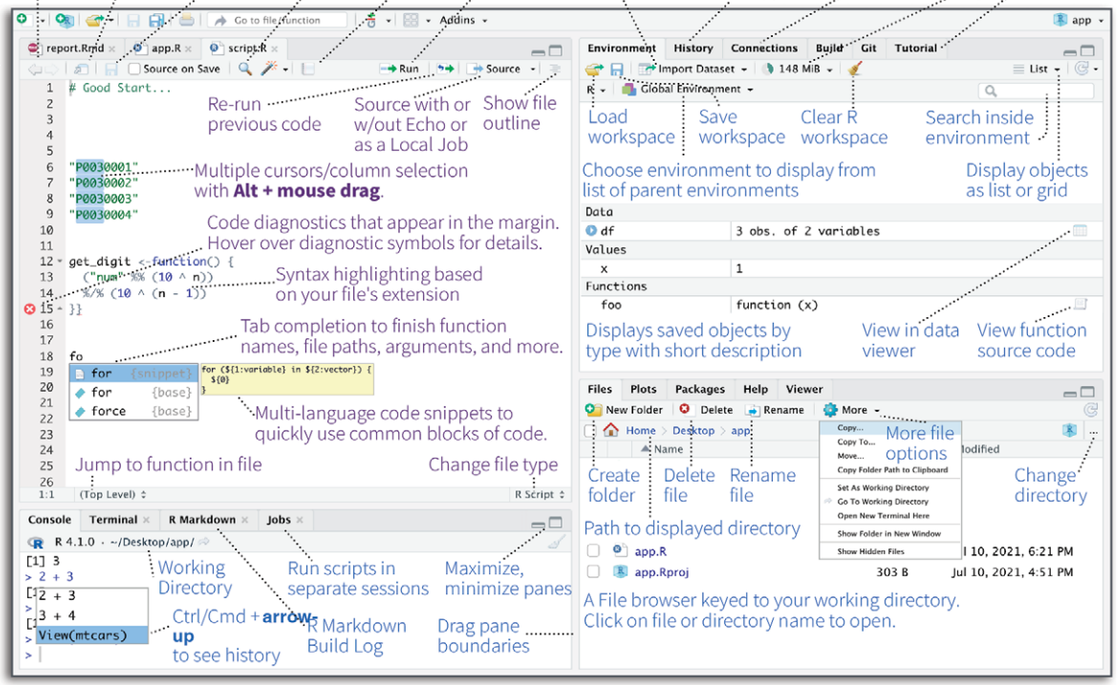
Open Shiny, R Markdown, knitr, Sweave, LaTeX, .Rd files and more in Source Pane



Access markdown guide at
Help > Markdown Quick Reference
See reverse side for more on **Visual Editor**

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

Source Editor

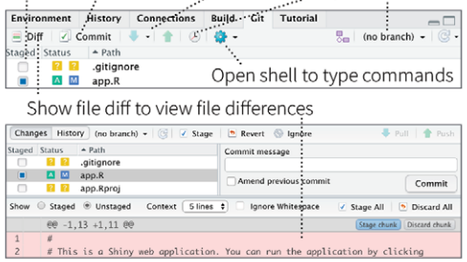


Tab Panes

Version Control

Turn on at **Tools > Project Options > Git/SVN**

Added · Modified · Untracked · Deleted · Renamed

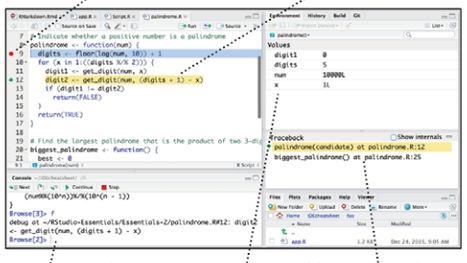


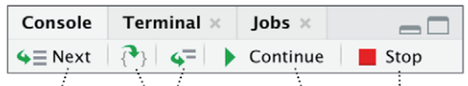
Debug Mode

Use **debug(), browser(),** or a breakpoint and execute your code to open the debugger mode.

Click next to line number to add/remove a breakpoint.

Highlighted line shows where execution has paused





Step through code one line at a time · Step into and out of functions to run · Resume execution · Quit debug mode

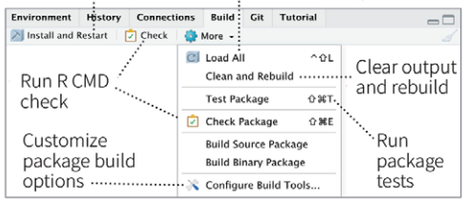
Package Development

Create a new package with
File > New Project > New Directory > R Package

Enable roxygen documentation with
Tools > Project Options > Build Tools

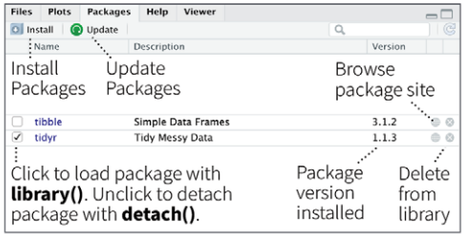
Roxygen guide at **Help > Roxygen Quick Reference**

See package information in the **Build Tab**

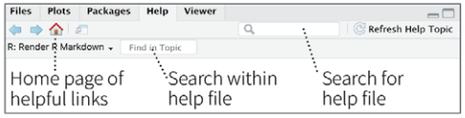


RStudio opens plots in a dedicated **Plots** pane

GUI **Package** manager lists every installed package

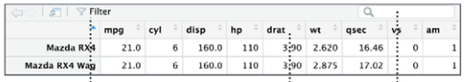


RStudio opens documentation in a dedicated **Help** pane



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

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



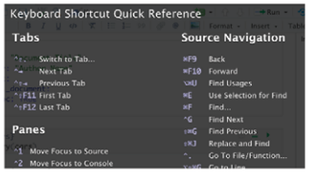
Keyboard Shortcuts

RUN CODE	Windows/Linux	Mac
Search command history	Ctrl+**arrow-up**	Cmd+**arrow-up**
Interrupt current command	Esc	Esc
Clear console	Ctrl+L	Ctrl+L
NAVIGATE CODE		
Go to File/Function	Ctrl+.	Ctrl+.
WRITE CODE		
Attempt completion	Tab or Ctrl+Space	Tab or Ctrl+Space
Insert <– (assignment operator)	Alt+-	Option+-
Insert |> or %>% (pipe operator)	Ctrl+Shift+M	Cmd+Shift+M
(Un)Comment selection	Ctrl+Shift+C	Cmd+Shift+C

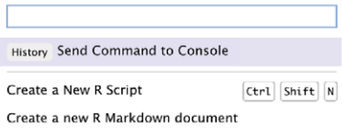
MAKE PACKAGES	Windows/Linux	Mac
Load All (devtools)	Ctrl+Shift+L	Cmd+Shift+L
Test Package (Desktop)	Ctrl+Shift+T	Cmd+Shift+T
Document Package	Ctrl+Shift+D	Cmd+Shift+D

DOCUMENTS AND APPS	Windows/Linux	Mac
Knit/Render Document (knitr)	Ctrl+Shift+K	Cmd+Shift+K
Insert chunk (Sweave & Knitr)	Ctrl+Alt+I	Cmd+Option+I
Run from start to current line	Ctrl+Alt+B	Cmd+Option+B
MORE KEYBOARD SHORTCUTS		
Keyboard Shortcuts Help	Alt+Shift+K	Option+Shift+K
Show Command Palette	Ctrl+Shift+P	Cmd+Shift+P

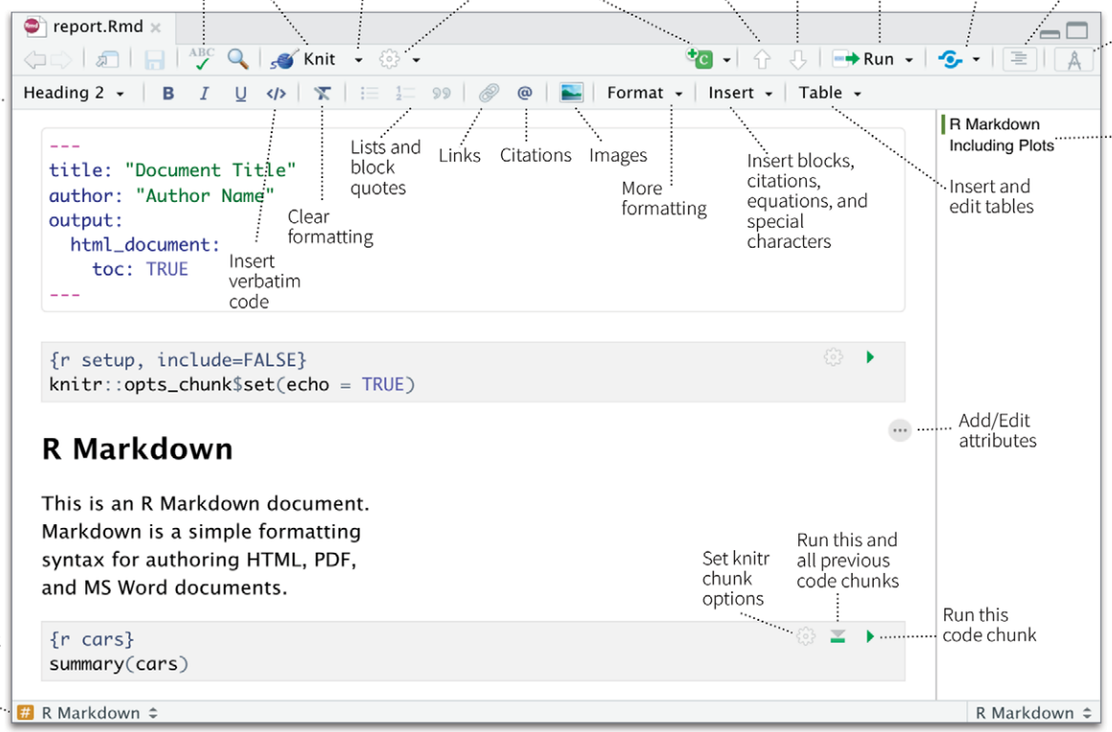
View the Keyboard Shortcut Quick Reference with **Tools > Keyboard Shortcuts** or **Alt/Option + Shift + K**



Search for keyboard shortcuts with **Tools > Show Command Palette** or **Ctrl/Cmd + Shift + P**.



Visual Editor



Posit Workbench

WHY POSIT WORKBENCH?

Extend the open source server with a commercial license, support, and more:

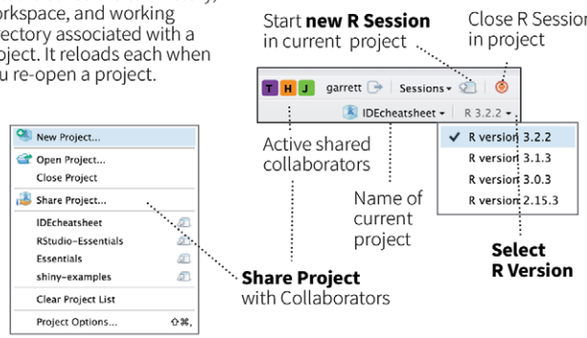
- open and run multiple R sessions at once
- tune your resources to improve performance
- administrative tools for managing user sessions
- collaborate real-time with others in shared projects
- switch easily from one version of R to a different version
- integrate with your authentication, authorization, and audit practices
- work in the RStudio IDE, JupyterLab, Jupyter Notebooks, or VS Code

Download a free 45 day evaluation at **posit.co/products/enterprise/workbench/**

Share Projects

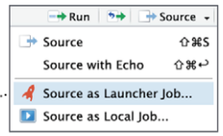
File > New Project

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



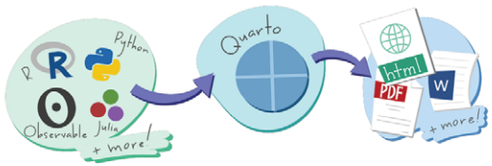
Run Remote Jobs

Run R on remote clusters (Kubernetes/Slurm) via the Job Launcher



CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **posit.co/products/open-source/rstudio** • HTML cheatsheets at **pos.it/cheatsheets** • RStudio IDE 2025.05.1+513 • Updated: 2025-07

Publish and Share with Quarto : : CHEATSHEET



Author

WRITE AND CODE IN PLAIN TEXT

Author documents as .qmd files or Jupyter notebooks. Write in a rich Markdown syntax.

Render

GENERATE DOCUMENTS, PRESENTATIONS AND MORE

Produce HTML, PDF, MS Word reveal.js, MS Powerpoint, Beamer Websites, blogs, books...

Publish

SHARE YOUR WORK WITH THE WORLD

Quickly deploy to GitHub Pages, Netlify, Quarto Pub, Posit Cloud, or Posit Connect

GET QUARTO

https://quarto.org/docs/download/

Or use version **bundled with Positron or RStudio**

GET STARTED

https://quarto.org/docs/get-started/

Author

SOURCE FILE: hello.qmd

```
---
title: "Hello, Penguins"
format: html
execute:
  echo: false
---

## Meet the penguins

The `penguins` data contains size m
the Palmer Archipelago, Antarctica.

The three species of penguins have
dimensions (@fig-penguins).

```{r}
#| label: fig-penguins
#| fig-cap: "Dimensions of penguins across three
#| warning: false
library(tidyverse, quietly = TRUE)
library(palmerpenguins)
penguins |>
  ggplot(aes(x = flipper_length_mm, y = bill_length_mm)) +
  geom_point(aes(color = species)) +
  scale_color_manual(
    values = c("darkorange", "purple", "cyan4")) +
```
```

Set format(s) and options
Use YAML Syntax

Write with **Markdown**
RStudio: Help > Markdown Quick Reference
Use **Visual Editor**

Include code
R, Python, Julia, Observable, or any language with a Jupyter kernel

USE A TOOL WITH A RICH EDITING EXPERIENCE

RStudio, Positron, VS Code + Quarto extension

Run code cells as you write

Render with a button or keyboard shortcut

Edit Quarto documents with a **Visual Editor**

Apply formatting in Visual Editor. Saved as Markdown in source.

Insert elements like code cells, cross references, and more.

Normal | B | I | Format | Insert | Table

OR ANY TEXT EDITOR

Quarto documents (.qmd) can be edited in any tool that edits text.

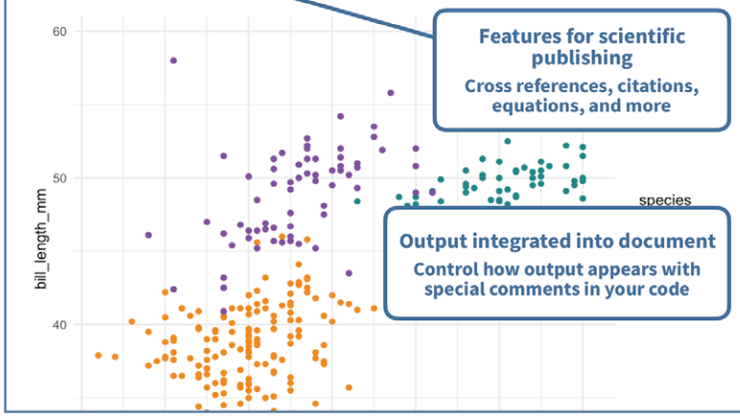
Render

RENDERED OUTPUT: hello.html

Hello, Penguins

Meet the penguins

The three species of penguins have quite distinct distributions of physical dimensions (Figure 1).



Features for scientific publishing
Cross references, citations, equations, and more

Output integrated into document
Control how output appears with special comments in your code

Save, then render to **preview** the document output.

Terminal
```
quarto preview hello.qmd
```

Use **Render** button

Use **Preview** button

The resulting HTML/PDF/MS Word/etc. document will be created and saved in the same directory as the source .qmd file.

BEHIND THE SCENES

When you render a document, Quarto:

1. Runs the code and embeds results and text into an .md file with: **Knitr**, if any {r} cells or, **Jupyter**, if any other cells.
2. Converts the .md file into the output format with Pandoc.

Publish

Terminal
```
quarto publish {venue} hello.qmd
```

{venue}: quarto-pub, connect, gh-pages, netlify, confluence

Quarto Pub — Free publishing service for Quarto content.

posit Connect — Org-hosted, control access, schedule updates.

Use **Publish** button

Use **Posit Publisher** extension

Quarto Projects

CREATE WEBSITES, BOOKS, AND MORE

A directory of Quarto documents + a configuration file (_quarto.yml)

See examples at https://quarto.org/docs/gallery/

Get started from the command line:

Terminal
```
quarto create project {type}
```

{type}: default, website, blog, book, confluence, manuscript

Use **File > New Project**

Use **Quarto: Create Project** command

Artwork from "Hello, Quarto" keynote by Julia Lowndes and Mine Çetinkaya-Rundel, presented at RStudio Conference 2022. Illustrated by Allison Horst.

Include Code

CODE CELLS

Code cells start with ```{language} and end with ```.

Use **Insert Code Chunk/Cell**

```{r}
#| label: chunk-id
library(tidyverse)
```

```{python}
#| label: chunk-id
import pandas as pd
```

Other languages: {julia}, {ojs}

Add code cell options with **#|** comments.

Cell options control **execution**, figures, tables, layout and more. See them all at: https://quarto.org/docs/reference/cells

EXECUTION OPTIONS

OPTION	DEFAULT	EFFECTS
echo	true	false: hide code fenced: include code cell syntax
eval	true	false: don't run code
include	true	false: don't include code or results
output	true	false: don't include results asis: treat results as raw markdown
warning	true	false: don't include warnings in output
error	false	true: include error in output and continue with render

Set execution options at the **cell level**:

```{r}
#| echo: false
```

```{python}
#| echo: false
```

Or, **globally** in the YAML header with the **execute** option:

```
---
execute:
  echo: false
---
```

Set options in code cells with #| comments and YAML syntax:
key: value

INLINE CODE

Use computed values directly in text sections. Code is evaluated at render and results appear as text.

KNITR	JUPYTER	OUTPUT
Value is `r 2 + 2`.	Value is `{python} 2 + 2`.	Value is 4.

Set Format and Options

SET FORMAT OPTIONS

```
---
title: "My Document"
format:
  html:
    code-fold: true
    toc: true
---
```

Indent options 4 spaces

Indent format 2 spaces

MULTIPLE FORMATS

```
---
title: "My Document"
toc: true
format:
  html:
    code-fold: true
  pdf: default
---
```

Top-level options apply to all formats

Common formats: **html, pdf, docx, odt, rtf, gfm, pptx, revealjs, beamer**

Render **all** formats:

Terminal
```
quarto render hello.qmd
```

Render a **specific** format:

Terminal
```
quarto render hello.qmd --to pdf
```

	OPTION	html/revealjs	pdf/beamer	docx/pptx	DESCRIPTION
Nav	**toc**	X	X	X	Add a table of contents (true or false)
Nav	**toc-depth**	X	X	X	Lowest level of headings to add to table of contents (e.g. 2, 3)
Nav	**anchor-sections**	X			Show section anchors on mouse hover (true or false)
Style	**highlight-style**	X	X	X	Syntax highlighting theme (e.g. arrow, pygments, kate, zenburn)
Style	**mainfont, monofont**	X	X		Font name. HTML: sets CSS font-family; LaTeX: via fontspec package
Style	**theme**	X			Bootswatch theme name (e.g. cosmo, darkly, solar etc.)
Style	**css**	X			CSS or SCSS file to use to style the document (e.g. "style.css")
	reference-doc			X	docx/pptx file containing template styles (e.g. file.docx, file.pptx)
LaTeX	**include-in-header**	X	X		Files of content to include in header of output document, also **include-before-body, include-after-body**
LaTeX	**keep-md**	X	X	X	Keep intermediate markdown (true or false), also **keep-ipynb**, **keep-tex**
LaTeX	**documentclass**		X		LaTeX document class, set document class options with **classoption**
LaTeX	**pdf-engine**		X		LaTeX engine to produce PDF output (xelatex, pdflatex, lualatex)
LaTeX	**cite-method**		X		Method used to format citations (citeproc, natbib, biblatex)
Code	**code-fold**	X			Let readers toggle the display of R code (false, true, or show)
Code	**code-tools**	X			Add menu for hiding, showing, and downloading code (true or false)
Code	**code-overflow**	X			Display of wide code (scroll, or wrap)
Figures	**fig-align**	X	X	/	Alignment of figures (default, left, right, or center)
Figures	**fig-width, fig-height**	X	X	X	Default width and height for figures in inches
Figures	**fig-format**	X	X	X	Format for Matplotlib or R figures (retina, png, jpeg, svg, or pdf)

Also use in code cells

Knitr

Visit https://quarto.org/docs/reference/ to see **all options** by format

Add Content

FIGURES

MARKDOWN

```
![CAP](image.png){#fig-LABEL fig-alt="ALT"}
```

COMPUTATION

```{python}
#| label: fig-LABEL
#| fig-cap: CAP
#| fig-alt: ALT
{{ plot code here }}
```

Or {r}

CROSS REFERENCES

1. **Add labels**
   Code cell: add option label: prefix-LABEL
   Markdown: add attribute #prefix-LABEL
2. **Add references** @prefix-LABEL, e.g.

You can see in **@fig-scatterplot**, that...

Prefix	Renders	Prefix	Renders
fig-	Figure 1	eq-	Equation 1
tbl-	Table 1	sec-	Section 1

TABLES

MARKDOWN

```
|object | radius|
|:------|------:|
|Sun    | 696000|
|Earth  |   6371|

: CAPTION {#tbl-LABEL}
```

Insert Table in **Visual Editor**

COMPUTATION Output a Markdown table or an HTML table from your code

KNITR

Use knitr::kable() to produce Markdown:

```{r}
#| label: tbl-LABEL
#| tbl-cap: CAPTION
knitr::kable(head(cars))
```

Also see the R packages: gt, flextable, kableExtra.

JUPYTER Add Markdown()to Markdown output:

```{python}
#| label: tbl-LABEL
#| tbl-cap: CAPTION
import pandas as pd, tabulate
from IPython.display import Markdown
df = pd.DataFrame({"A": [1, 2],
                   "B": [1, 2]})
Markdown(df.to_markdown(index=False))
```

CITATIONS

1. Add a bibliography **file** to the YAML header:

```
---
bibliography: references.bib
---
```

2. Add citations: [@citation], or @citation

Use **Insert Citations** dialog in the **Visual Editor**

Build your bibliography file from your Zotero library, DOI, Crossref, DataCite, or PubMed

CALLOUTS

tip

```
::: {.callout-tip}
## Title

Text
:::
```

Instead of **tip** use one of: note, caution, warning, or important.

note | warning | caution | important

SHORTCODES

```
{{< include _file.qmd >}}
{{< embed file.ipynb#id >}}
{{< video video.mp4 >}}
```

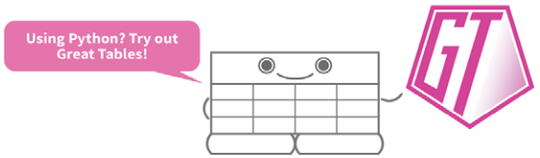
gt : : CHEAT SHEET

Introduction

The gt package allows you to make beautiful, informative table displays from R data frames. The package provides functions for you to **structure**, **format**, and **style** the table to your heart's content.

The resulting tables can be easily displayed in Shiny apps and Quarto documents. And there are quite a few output formats for your tables, including HTML, LaTeX, RTF, and Word.

INSTALLATION `install.packages("gt")`



Adding Structure

ADDING A HEADER AND FOOTER

`tab_header()` → add title and optional subtitle, can use `md()` or `html()` helpers

`tab_source_note()` → add text to table footer (above helpers can be used here too)

`tab_footnote()` → add footnote to footer; needs a *location* (see *Styling...* for this)

ADDING A STUB AND ROW GROUPS

`gt(rowname_col = "column")` → assign a column with row names to the stub

`gt(groupname_col = "column")` → use column with categoricals to group rows

ADDING COLUMN SPANNERS

`tab_spanner()` → add spanner over cols

`tab_spanner_delim()` → use delimited column names to set up multiple spanners

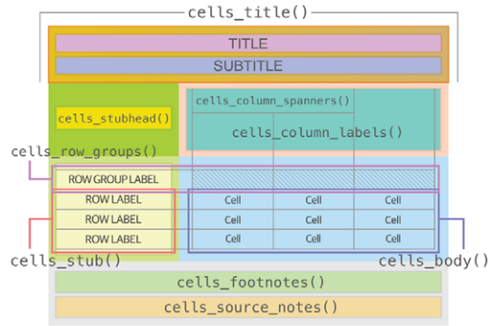
Styling the Table

ONE FUNCTION FOR STYLING EVERYWHERE

`tab_style()` → apply styles to various locations

```
gt_tbl |> tab_style(style = list(styles), locations = list(locations))
```

THE DIFFERENT LOCATIONS OF A TABLE



THREE WAYS TO STYLE

`cell_fill()` → fill location with color

`cell_text()` → style location's text

`cell_borders()` → set up borders and define styles for them

YOU ALSO HAVE OTHER OPTIONS

`tab_options()` → many, many options for styling that applies to whole table

`opt_stylize()` → set a theme, 36 looks

`opt_table_font()` → set a font to use

Creating a gt Table

BASIC EXAMPLE WITH SAMPLE DATA

```
library(gt)

gt_tbl = gt(exibble)
gt_tbl
```

create GT object

display HTML table

OUTPUT

column labels

num	char	fctr	date	time	datetime	currency	row	group
1.111e-01	apricot	one	2015-01-15	13:35	2018-01-01 02:22	49.950	row_1	grp_a
2.222e+00	banana	two	2015-02-15	14:40	2018-02-02 14:33	17.950	row_2	grp_a
3.333e+01	coconut	three	2015-03-15	15:45	2018-03-03 03:44	1.390	row_3	grp_a
4.444e+02	durian	four	2015-04-15	16:50	2018-04-04 15:55	65100.000	row_4	grp_a
5.550e+03	NA	five	2015-05-15	17:55	2018-05-05 04:00	1325.810	row_5	grp_b
NA	fig	six	2015-06-15	NA	2018-06-06 16:11	13.255	row_6	grp_b
7.770e+05	grapefruit	seven	NA	19:10	2018-07-07 05:22	NA	row_7	grp_b
8.880e+06	honeydew	eight	2015-08-15	20:20	NA	0.440	row_8	grp_b

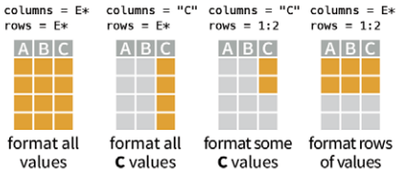
table body

SAMPLE DATASETS `countrypops`, `sza`, `gtcars`, `sp500`, `films`, `pizzaplace`, `exibble`, `towny`, `peeps` + more

Formatting Data Values

FORMATTING BASICS

Formatting functions have the `columns` and `rows` options to specify which body values should be formatted.



`E*` means `everything()`

NUMERIC FORMATTING

1 `fmt_number()` → decimal values
2 `fmt_integer()` → integer values
3 `fmt_percent()` → percentage values
4 `fmt_scientific()` → scientific not'n
5 `fmt_engineering()` → engineering not'n

common options: `decimals` in 1,3,4,5 for fixed # of decimals / `scale_by` in 1,2,4,5 to multiply number before formatting / `accounting` in 1,2,3 to display in that notation / `suffixing` in 1,2 to condense large numbers / `locale` in 1–5 to localize formatted values to language/region.

DATE/TIME FORMATTING

`fmt_date()` → format with a date style preset

```
iso: "2000-02-29" (the default)
wday_month_day_year: "Tuesday, February 29, 2000"
wd_m_day_year: "Tue, Feb 29, 2000"
wday_day_month_year: "Tuesday 29 February 2000"
month_day_year: "February 29, 2000"
```

`fmt_time()` → format with a time style preset

```
iso: "14:35:00" (the default)
h_m_s_p: "2:35:00 PM"
h_m_p: "2:35 PM"
```

`fmt_datetime()` → format with date/time style presets or with `format` option for more control

FORMATTING INVOLVING GRAPHICS

`fmt_image()` → insert images to cells with filenames; `path` / `file_pattern` help resolve

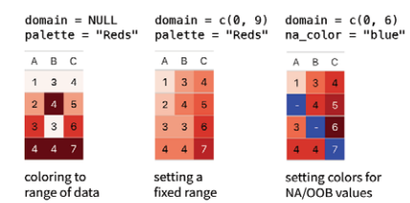
`fmt_icon()` → add icons to cells containing FontAwesome short icon names (e.g., `"cat"`)

`fmt_flag()` → add flag icons to cells w/ 2- or 3-letter country codes (e.g., `"LU"` or `"LUX"`)

common to all: cells may have multiple instances of replaceable text for these formatters, they must be comma-separated though.

COLORIZING BODY CELLS

`data_color()` → apply color to a series of cells according to their value



ADDING NANOPLOTS

`cols_nanoplot()` → convert series of values (strings w/ vals or a list column) to tiny interactive plots in a new column

	vals	nanoplots
1	20 23 6 7 37 23 21 4 7 16	
2	2.3 6.8 9.2 2.42 3.5 12.1 5.3 3.6	

CC BY SA Richard Iannone • rich@posit.co • Learn more at the Great Tables project website • package version 0.18.0 • Updated: 2025-07

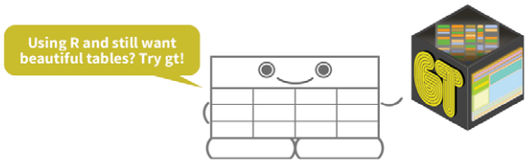
Great Tables : : CHEAT SHEET

Introduction

Great Tables allows you to make beautiful and presentable table displays from Python DataFrames (Polars, Pandas, and Arrow). The package provides methods for you to **structure**, **format**, and **style** the table to your heart's content.

The resulting tables can be easily displayed in notebooks and Quarto documents. And there are options for saving images of a table or getting HTML or LaTeX table code.

INSTALLATION `pip install great_tables`



Adding Structure

ADDING A HEADER AND FOOTER

`tab_header()` → add title and optional subtitle, can use `md()` or `html()` helpers

`tab_source_note()` → add text to table footer (helpers can be used here too)

ADDING A STUB AND ROW GROUPS

`GT(rowname_col="column")` → assign a column with row names to the stub

`GT(groupname_col="column")` → column with categoricals groups rows

ADDING COLUMN SPANNERS

`tab_spanner()` → add spanner over cols

`tab_spanner_delim()` → use delimited col names to set up multiple spanners

Styling the Table

```
from great_tables import loc, style
```

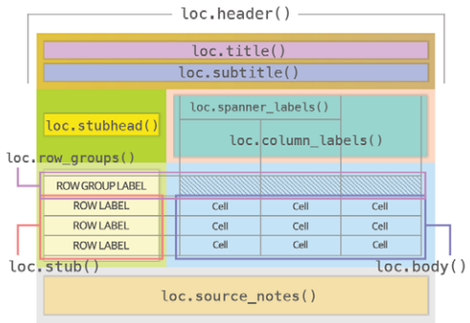
import these two modules

ONE METHOD FOR STYLING EVERYWHERE

`tab_style()` → apply styles to various locations

```
gt_tbl.tab_style(style=[styles], locations=[locations])
```

THE DIFFERENT LOCATIONS OF A TABLE



THREE WAYS TO STYLE

`style.fill()` → fill location with color

`style.text()` → style location's text

`style.borders()` → set up borders and define styles for them

YOU ALSO HAVE OTHER OPTIONS

`tab_options()` → many, many options for styling that applies to whole table

`opt_stylize()` → set a theme, 36 looks

`opt_table_font()` → set a font to use

Creating a GT Table

BASIC EXAMPLE WITH SAMPLE DATA

```
from great_tables import GT
from great_tables.data import islands
```

import GT class

dataset

```
gt_tbl = GT(islands.head(5))
gt_tbl
```

create GT object

display HTML table

OUTPUT

column labels

name	size
Africa	11506
Antarctica	5500
Asia	16988
Australia	2968
Axel Heiberg	16

table body

SAMPLE DATASETS

`countrypops`, `sza`, `gtcars`, `sp500`, `pizzaplace`, `exibble`, `towny` + more

SHOWING IN BROWSER

```
gt_tbl.show("browser")
```

POLARS STYLE PROPERTY

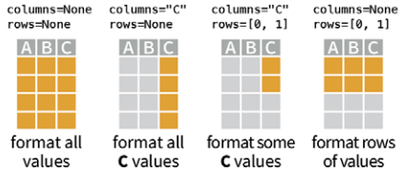
```
df_polars.style
```

create GT object with Polars API

Formatting Data Values

FORMATTING BASICS

Formatting methods have the `columns` and `rows` parameters to specify which body values should be formatted.



NUMERIC FORMATTING

1 `fmt_number()` → decimal values
2 `fmt_integer()` → integer values
3 `fmt_percent()` → percentage values
4 `fmt_scientific()` → scientific not'n

common params: `decimals` in 1,3,4 for fixed # of decimals / `scale_by` in 1,2,4 to multiply number before formatting / `accounting` in 1,2,3 to display in that notation / `compact` in 1,2 to condense large numbers / `locale` in 1–4 to localize formatted values to language/region.

DATE/TIME FORMATTING

`fmt_date()` → format with a date style

```
iso: "2000-02-29" (the default)
wday_month_day_year: "Tuesday, February 29, 2000"
wd_m_day_year: "Tue, Feb 29, 2000"
wday_day_month_year: "Tuesday 29 February 2000"
month_day_year: "February 29, 2000"
```

`fmt_time()` → format with a time style

```
iso: "14:35:00" (the default)
h_m_s_p: "2:35:00 PM"
h_m_p: "2:35 PM"
```

`fmt_datetime()` → format with date/time styles or with the `format_str` parameter

FORMATTING INVOLVING GRAPHICS

`fmt_image()` → insert images to cells with filenames; `path` / `file_pattern` help resolve

`fmt_icon()` → add icons to cells containing FontAwesome short icon names (e.g., `"cat"`)

`fmt_flag()` → add flag icons to cells w/ 2- or 3-letter country codes (e.g., `"LU"` or `"LUX"`)

common to all: cells may have multiple instances of replaceable text for these formatters, they must be comma-separated though.

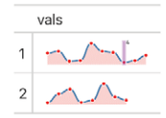
COLORIZING BODY CELLS

`data_color()` → apply color to a series of cells according to their value

ADDING NANOPLOTS

`fmt_nanoplot()` → convert series of values (strings w/ vals or a list column) to tiny interactive plots

	vals
1	20 23 6 7 37 23 21 4 7 16
2	2.3 6.8 9.2 2.42 3.5 12.1 5.3 3.6



CC BY SA Richard Iannone • rich@posit.co • Learn more at the Great Tables project website • package version 0.18.0 • Updated: 2025-07

Data visualization with Plotnine # *cheatsheet*

Basics

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

To display values, map columns 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 plot:

```
from plotnine import *
from plotnine.data import *
                                          required
                                          optional
(
    ggplot(data=data_frame)
    + geom_function(
        mapping=aes(**mappings),
        stat=stat,
        position=position)
    + coord_function()
    + facet_function()
    + scale_function()
    + theme_function()
    + theme(**settings)
)
```

For example:

```
p = ggplot(mpg, aes("cty", "hwy")) + geom_point()

p.save("plot.png", width=6, height=4, dpi=200)
```

Aesthetics

Common properties and values

color and **fill**: color name or "#RRGGBB"
linetype: str one of ["solid", "dashed", "dashdot", "dotted"] or tuple of the form (offset, (on, off, on, off, ...))
size: int (in points)
linewidth: int (in points)
shape: character or int

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()
# Ensure limits include values across all plots
```

```
a + geom_path(lineend="butt", linejoin="round", linemitre=1)  # x, y, alpha, color, group, linetype, size
```

```
a + geom_polygon(aes(alpha=50))  # color, fill, group, subgroup, linetype, size
```

```
b + geom_rect(aes(xmin="long", ymin="lat", xmax="long+1", ymax="lat+1"))  # xmin, xmax, ymin, ymax, alpha, color, fill, linetype, size
```

```
a + geom_ribbon(aes(ymin="unemploy-900", ymax="unemploy+900"))  # x, ymin, ymax, 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 continuous variable

```
c = ggplot(mpg, aes(x="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
```

One discrete variable

```
d = ggplot(mpg, aes("fl"))
```

```
d + geom_bar()  # x, alpha, color, fill, linetype, size, weight
```

Two continuous variables

```
e = ggplot(mpg, aes(x="cty", y="hwy"))
```

```
e + geom_label(aes(label="cty"), nudge_x=1, nudge_y=1)  # 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)  # alpha, angle, color, family, fontface, hjust, lineheight, size, vjust
```

One discrete, one continuous variable

```
f = ggplot(mpg, aes(x="class", y="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
```

Two discrete variables

```
g = ggplot(diamonds, aes(x="cut", y="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
```

Three continuous variables

```
import polars as pl

seals = pl.DataFrame(seals).with_columns(z=(pl.col("delta_long")**2 + pl.col("delta_lat")**2).sqrt())
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
```

Continuous function

```
h = ggplot(economics, aes("date", "unemploy"))
```

```
h + geom_area()  # x, y, alpha, color, fill, linetype, size
```

```
h + geom_line()  # x, y, alpha, color, group, linetype, size
```

```
h + geom_step(direction="hv")  # x, y, alpha, color, group, linetype, size
```

Continuous bivariate distribution

```
i = ggplot(diamonds, aes("carat", "price"))
```

```
i + geom_bin_2d(binwidth=[0.25, 500])  # x, y, alpha, color, fill, linetype, size, weight
```

```
i + geom_density_2d()  # x, y, alpha, color, group, linetype, size
```

Visualizing error

```
df = pl.DataFrame({"grp": ["A", "B", "C"], "fit": [4, 5, 6], "se": [1, 2, 1]}); 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(); j + geom_errorbarh()  # x, ymax, ymin, alpha, color, group, linetype, size, width
```

```
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

```
import geopandas as gp
import geodatasets as gd

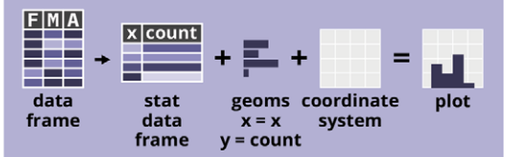
np = gp.read_file(gd.get_path("geoda.nepal"))
```

```
ggplot(np) + geom_map(aes(fill="population"))
```

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 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=True)  # 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
e + stat_quantile(quantiles=(0.1, 0.9),
    formula="y ~ np.log(x)")  # x, y | quantile
e + stat_smooth(method="lm", formula="y ~ x",
    se=True, level=0.95)  # x, y | se, ymin, ymax
```

```
import scipy.stats as stats

ggplot() + lims(x=(-5, 5)) +
    stat_function(fun=stats.norm.pdf, n=20,
    geom="point")  # x | y
ggplot() + stat_qq(aes(sample=range(100)))  # x,
    y | sample, theoretical
e + stat_sum()  # x, y, size | n, prop
e + stat_summary(fun_data="mean_cl_boot")
i + stat_summary_bin(fun="mean", geom="bar")
e + stat_identity()
e + stat_unique()
```

Scales

Override default mappings.

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

```
n = d + geom_bar(aes(fill="fl"))
```

aesthetic to adjust | scale to use | scale-specific arguments

```
n + scale_fill_manual(values=["palegreen",
    "green", "forestgreen", "darkgreen"],
    limits=["d", "e", "p", "r"],
    breaks=["d", "e", "p", "r"],
    labels=["D", "E", "P", "R"],
    name="fuel")
```

labels to use in legend/axis | title to use in legend/axis | breaks to use in legend/axis

General purpose scales

Use with most aesthetics

```
scale_*_continuous()  # Map continuous values
scale_*_discrete()  # Map discrete values
scale_*_binned()  # Map continuous values to bins
scale_*_identity()  # Use data values literally
scale_*_manual(values=[])  # Map discrete values
    to manually chosen visual ones
scale_*_date(date_labels="%d/%m"), date_breaks=
    "2 weeks")  # Treat data values as dates
scale_*_datetime()  # Treat values as date times
```

X and Y location scales

Use with x or y aesthetics (x shown here)

```
scale_x_log10()  # Plot on log10 scale
scale_x_reverse()  # Reverse direction of axis
scale_x_sqrt()  # Plot on square root scale
```

Color and fill scales, discrete

```
n + scale_fill_brewer(palette="Blues")
```

```
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=["green",
    "purple", "papayawhip"])
```

Shape and size scales

```
p = e + geom_point(aes(shape="fl", size="cyl"))
```

```
p + scale_shape() + scale_size()
p + scale_shape_manual(values=["o", "^",
    "s", "D", "v"])  # see page 1 for shapes
```

```
p + scale_radius(range=range(1, 7))
p + scale_size_area(max_size=6)
```

Coordinate Systems

```
r = d + geom_bar()
```

```
r + coord_cartesian(xlim=[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 coordi+
    nates by switching x and y aesthetic
    mappings.
```

```
r + coord_trans(y="sqrt")  # x, y, xlim,
    ylim. Transformed cartesian coordinates.
```

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
```

```
r + theme_gray()  # Grey background (default)
```

```
r + theme_dark()  # Dark for contrast
```

```
r + theme_classic()
r + theme_light()
```

```
r + theme_linedraw()
r + theme_minimal()
```

```
r + theme_void()  # Empty theme
```

```
r + theme()  # Customize aspects of the theme such
as axis, legend, panel, and facet properties.
r + 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(cols="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", ncol=4)
# 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(cols="fl", labeller="label_both")
```

fl: c	fl: d	fl: e	fl: p	fl: 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",
    tag="Add a tag to the plot",
    ="New  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 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=["A",
"B", "C", "D", "E"])  # Set legend title and labels
with a scale function
```

Zooming

Without clipping (preferred):

```
t + coord_cartesian(xlim=[10, 100],
ylim=[0, 20])
```

With clipping (removes unseen data points):

```
t + lims(x=(10, 100), y=(0, 20))  # option 1
t + scale_x_continuous(limits=[10, 100]) +
scale_y_continuous(limits=[0, 20])  # option 2
```


CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at plotnine.org • HTML cheatsheets at pos.it/cheatsheets • plotnine 0.15 • Updated: 2025-08

Shiny : : CHEATSHEET

Build an app

A **Shiny** app is a web page (**ui**) connected to a computer running a live R session (**server**).

Users can manipulate the UI, which will cause the server to update the UI's displays (by running R code).

🤖 Build with AI assistance:
gallery.shinyapps.io/assistant

💡 Get inspiration & examples:

- shiny.posit.co/r/gallery
- shinylive.io/r/examples
- `runExample()` in R console

The **UI** is a collection of input, output, and layout elements

The **server** determines how to render outputs given inputs

An **app** is a combination of UI and server logic

```
# app.R
library(shiny)

ui <- bslib::page_fluid(
  sliderInput(
    "n", "Sample size", 0, 100, 25
  ),
  plotOutput("hist")
)

server <- function(input, output) {

  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

shinyApp(ui, server)
```

Create `*Input()` UI and read with `input$<id>`

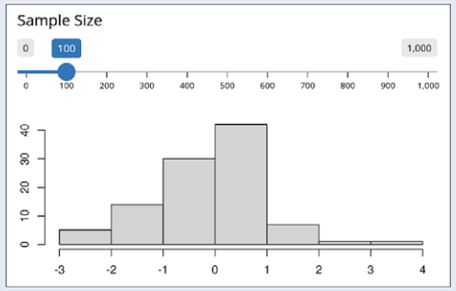
Match `*Output()` with `render*()` to add R output

Save `shinyApp()` to **app.R**

Optionally include supporting code, images, etc. in R/ and www/ folders

app-name
- app.R
- R/
- www/

Launch an **app.R** with `runApp("path/to/app-name")`.



Inputs

Collect values from the user.

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

Action — `actionButton(id, label, …)`

Action — `actionLink(id, label, …)`

Choice 1, Choice 2, Choice 3 — `checkboxGroupInput(id, label, choices, selected, …)`

Check me — `checkboxInput(id, label, value, …)`

`dateInput(id, label, value, …)`
`dateRangeInput(id, label, start, end, …)`

Choose File — `fileInput(id, label, …)`

1 — `numericInput(id, label, value, …)`

Option 1, Option 2, Option 3 — `radioButtons(id, label, choices, selected, …)`

Choice 1A, Choice 1B — `selectInput(id, label, choices, selected, multiple, …)`
Also `selectizeInput()`

0 50 100 — `sliderInput(id, label, min, max, value, …)`

Enter text... — `textInput(id, label, value, …)`
Also `textAreaInput()`

More from the **bslib** package:

`input_dark_mode(id, mode)`

`input_switch(id, label, value, …)`

Processing... — `input_task_button(id, label, value, …)`

See input gallery at **shiny.posit.co/r/components**

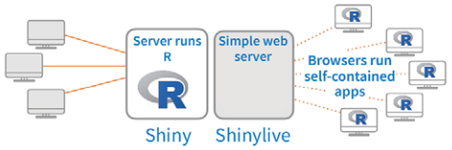
Share

Share your app in four ways:

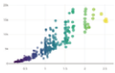
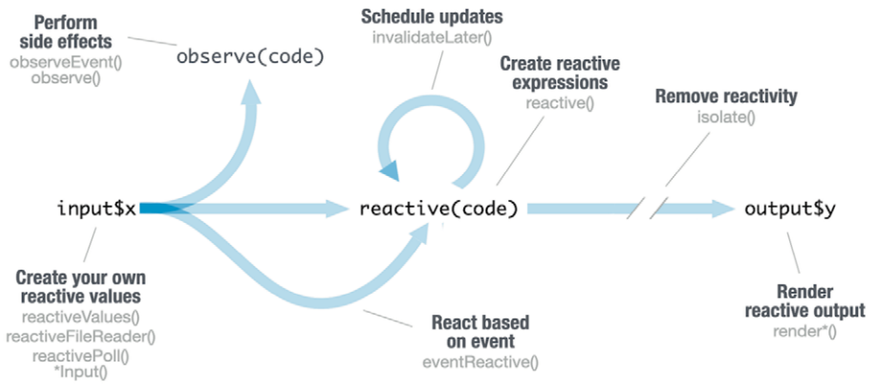
1. **Host it on shinyapps.io**, a cloud based service from Posit. To deploy Shiny apps:
   - Create a free or professional account at **shinyapps.io**
   - Click the Publish icon in RStudio IDE, or run: `rsconnect::deployApp("path/to/app-name")`
2. **Purchase Posit Connect**, a publishing platform for R and Python. **posit.co/connect**
3. **Host your own Shiny Server** **posit.co/products/open-source/shinyserver**
4. **Export to shinylive**, a technology for running apps entirely in the browser. **posit-dev.github.io/r-shinylive**

Shinylive

Shinylive apps use WebAssembly to run entirely in a browser–no need for a server to run R.



- Edit and/or host apps at **shinylive.io/r**
- Export an app to Shinylive with `shinylive::export("app-name", "site")` Then deploy to a hosting site like Github or Netlify
- Embed Shinylive apps in Quarto sites, blogs, etc

```
---
filters:
 - shinylive
---
An embedded Shinylive app:
```{shinylive-r}
#| standalone: true
# [App.py code here…]
```
```

To embed a Shinylive app in a Quarto doc, include the bold syntax.

Outputs

Reactively render R outputs

`plotOutput(id, width, height,…)`
`renderPlot(expr, …)`

Species	Island	Bill Length (mm)	Body Mass (g)
Chinstrap	Dream	45.70	3650
Chinstrap	Dream	55.80	4000
Chinstrap	Dream	43.50	3400
Chinstrap	Dream	49.60	3775

`tableOutput(id, …)`
`renderTable(expr, striped, …)`

	area	peri	shape	perm
1	4990	2791.90	0.0903296	6.3
2	7002	3892.60	0.1486220	6.3
3	7558	3930.66	0.1833120	6.3
4	7352	3869.32	0.1170630	6.3

`verbatimTextOutput(id, …)`
`renderPrint(expr, …)`

Current value: 30

`textOutput(id, …)`
`renderText(expr, …)`

Current *value*: **30**

`uiOutput(id, …)`
`renderUI(expr, …)`

`imageOutput(id, …)`
`renderImage(expr, …)`

More from **htmlwidgets.org** ecosystem

`leafletOutput(id, …)`
`renderLeaflet(expr, …)`



`plotlyOutput(id, …)`
`renderPlotly(expr, …)`

See output gallery at **shiny.posit.co/r/components**

Reactivity

Reactive values work together with reactive functions. Call a reactive value from within the arguments of one of these functions to avoid the error **Operation not allowed without an active reactive context.**



CREATE REACTIVE VALUES

```
ui <- bslib::page_fluid(
  textInput("a", "", "A")
)

server <- \(input, output){
  print(isolate(input$a))

  rv <- reactiveVal(NULL)
  print(isolate(rv()))
}

shinyApp(ui, server)
```

`*Input()` functions

Create a reactive value **`input$<id>`** from user input.

`reactiveVal(``value`**`)`**

Create a reactive value from a given value. Useful for managing state.

RENDER REACTIVE OUTPUT

```
ui <- bslib::page_fluid(
  textInput("a", "", "A"),
  textOutput("b")
)

server <- \(input, output){
 output$b <- renderText({
   input$a
  })
}

shinyApp(ui, server)
```

`render*()` functions

Produces results for a corresponding `*Output()` UI container. Re-render occurs when reactive dependencies change.

Save the results to **`output$<id>`**.

CREATE REACTIVE EXPRESSIONS

```
ui <- bslib::page_fluid(
 textInput("a", "", "A"),
 textInput("z", "", "Z"),
 textOutput("b"))

server <- \(input, output){
 re <- reactive({
  paste(input$a, input$z)
 })
 output$b <- renderText({
  re()
 })
}

shinyApp(ui, server)
```

`reactive(``x`**`)`**

Calculate a (reactive) value based on other reactive values.

Useful for encapsulating reactive logic needed across multiple outputs.

PERFORM SIDE EFFECTS

```
ui <- bslib::page_fluid(
  textInput("a", "", "A"),
  actionButton("go", "Go")
)

server <- \(input, output){
  observe(print(input$a))

  observeEvent(input$go, {
   print(input$a)
 })
}

shinyApp(ui, server)
```

`observe(``x`**`)`**

Observe changes to reactive values

`observeEvent(`
`eventExpr, handlerExpr`
`)`

Runs code in 2nd argument when 1st argument changes.

REACT BASED ON EVENT

```
ui <- bslib::page_fluid(
 textInput("a", "", "A"),
 actionButton("go", "Go"),
 textOutput("b")
)

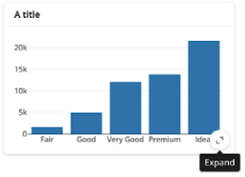
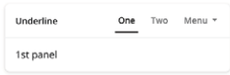
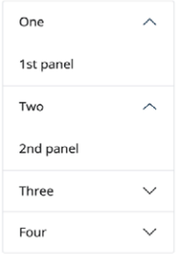
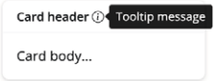
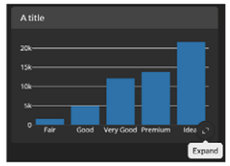
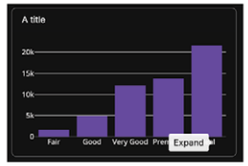
server <- \(input, output){
 re <- eventReactive(
  input$go, {input$a}
 )
 output$b <- renderText({
  re()
 })
}

shinyApp(ui, server)
```

`eventReactive(`
`eventExpr, valueExpr`
`)`

Creates reactive expression with code in 2nd argument that only invalidates when reactive values in 1st argument change.

REMOVE REACTIVE DEPENDENCIES

```
ui <- bslib::page_fluid(
  textInput("a", "", "A"),
  actionButton("go", "Go"),
  textOutput("b")
)

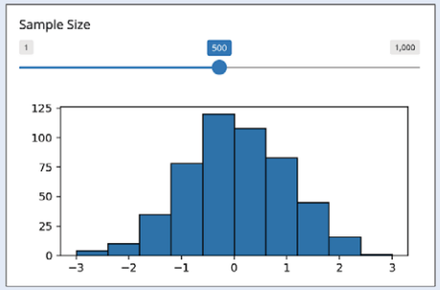
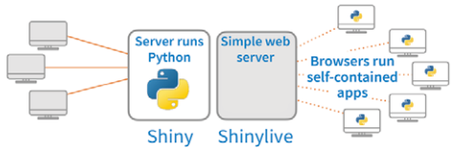
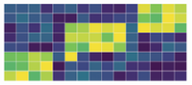
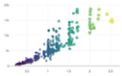
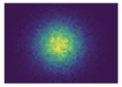
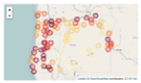
server <- \(input, output){

 output$b <- renderText({
   input$go
   isolate(input$a)
  })

}

shinyApp(ui, server)
```

`isolate(``expr`**`)`**

Prevent reactive values from invalidating a reactive expression.

User Interfaces (UI)

Design delightful UI with the **bslib** package. It provides layouts, components, themes, & more.

PAGE LAYOUTS

`page_sidebar()`	Screen-filling sidebar layout
`page_fillable()`	Screen-filling page layout
`page_fixed()`	Constrained width page
`page_fluid()`	Basic full-width page
`page_navbar()`	Multi-page app with a top nav bar

CARDS

Visually group UI elements together with the **`card()`** component.

```
card(full_screen = T,
  card_header("A title"),
  plotOutput("my_output"),
  card_footer("A footer")
)
```



UI LAYOUTS

Multiple columns

`layout_columns()`	Bootstrap's 12-column grid
`layout_column_wrap()`	Equal-width columns
`layout_sidebar()`	Resizable 2-column layout

Multiple panels

```
navset_card_underline(
  nav_panel(
    "One", "1st panel"),
  nav_panel(
    "Two", "2nd panel"),
  nav_menu("Menu",
    nav_panel("3", "3rd"))
)
```

Navigate a set of **`nav_panel()`**s in various ways with **`navset_card_*`**



ACCORDIONS

```
accordion(
 open = c("One", "Two"),
 accordion_panel(
   "One", "1st panel"),
 accordion_panel(
   "Two", "2nd panel"),
 accordion_panel(
   "Three", "3rd panel"),
)
```



Tip: place within **`sidebar()`** to group similar inputs

TOOLTIPS

```
tooltip(
  icon("info-circle"),
  "Tooltip message"
)
```



VALUE BOXES

```
value_box(
  "Title",
  "Value",
  showcase=icon()
)
```

Custom UI

Custom how your app looks and behaves.

THEMES

Bootswatch

Choose from over a dozen pre-packaged themes

```
library(bslib)
theme <- bs_theme(
  bootswatch="darkly"
)
ui <- page_fluid(
  theme=theme, ...
)
```



Custom themes

Quickly change main colors and fonts. Change in real-time by adding **`bs_themer()`** to your UI.

```
bs_theme(
  bg = "#222",
  fg = "white",
  primary = "purple",
  base_font =
font_google("Inter")
)
```



CUSTOM HTML

Shiny UI is powered by HTML, CSS, and JS:

```
page_fluid(class = "pt-3")
#> <div class="container-fluid pt-3"></div>
```

If you know these web technologies, you can customize UI to your heart's content. Start small by modifying/authoring HTML and including CSS/JS snippets. Or, go fully custom with **`htmlTemplate()`**

HTML

Add HTML elements with **`tags`**, a list of functions that parallel common HTML tags, e.g. **`tags$a()`**. Unnamed arguments are treated as children and named arguments become HTML attributes.

CSS

To include a CSS file, use **`includeCSS()`**, or
1. Place the file in the **`www`** subdirectory
2. Link to it with:

```
tags$head(tags$link(href="<file name>",
rel="stylesheet"))
```

JS

To include JS, use **`includeScript()`** or
1. Place the file in the **`www`** subdirectory
2. Link to it with:

```
tags$head(tags$script(src="<file name>"))
```

IMAGE

To include an image:
1. Place the file in the **`www`** subdirectory
2. Link to it with `img(src="<file name>")`

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **shiny.posit.co** • HTML cheatsheets at **pos.it/cheatsheets** • shiny 1.11.1 • Updated: 2025-07

Shiny : : CHEAT SHEET

Build an App

Shiny makes it easy to create truly reactive data & AI apps in pure Python

Shiny apps easily scale in complexity and sophistication thanks to its reactivity model and other opinionated design choices.

🤖 Build with AI assistance:
gallery.shinyapps.io/assistant

💡 Get inspiration & templates:

- Run `shiny create` in terminal
- shiny.posit.co/py/templates
- shiny.posit.co/py/gallery

Supports output from many popular packages

Collect user input with `input_*()`

Reactively render a plot output (re-execute when relevant `input` changes)

```
# app.py
import matplotlib.pyplot as plt
import numpy as np
from shiny.express import (
  input, render, ui
)

ui.input_slider(
  "n", "Sample Size", 0, 1000, 50
)

@render.plot
def dist():
  x = np.random.randn(input.n())
  plt.hist(x, range=[-3, 3])
```

Reactively reads the slider's value

Save as **app.py**

Include supporting code, images, etc. in same directory.

```
app-name
  app.py
  www/
  <other>/
```

Launch apps via the VSCode extension or with the `shiny run` CLI



Inputs

Collect values from the user.

Reactively read input values with `input.<id>()`

Action — `ui.input_action_button(id, label, …)`

Action — `ui.input_action_link(id, label, …)`

Processing... — `ui.input_task_button(id, label, …)`

Check me — `ui.input_checkbox(id, label, value, …)`

Choice 1, Choice 2, Choice 3 — `ui.input_checkbox_group(id, label, choices, selected, …)`

`ui.input_dark_mode(id, mode)`

`ui.input_date(id, label, value, …)`

`ui.input_date_range(id, label, start, end, …)`

Choose File — `ui.input_file(id, label, …)`

1 — `ui.input_numeric(id, label, value, …)`

Option 1, Option 2, Option 3 — `ui.input_radio_buttons(id, label, choices, selected, …)`

Choice 1A, Choice 1A, Choice 1B — `ui.input_select(id, label, choices, selected, …)`
Also `ui.input_selectize()`

0, 50, 100 — `ui.input_slider(id, label, min, max, value, …)`

`ui.input_switch(id, label, value, …)`

Enter text... — `ui.input_text(id, label, value, …)`
Also `ui.input_text_area()`

Share

Share your app in three ways:

1. **Host it on shinyapps.io**, a cloud based service from Posit. To deploy Shiny apps:
   - Create a free or professional account at **shinyapps.io**
   - Use the reconnect-python package to publish with `rsconnect deploy shiny <path to directory>`
2. **Purchase Posit Connect**, a publishing platform for R and Python. **posit.co/connect**
3. **Use open source deployment options** **shiny.posit.co/py/docs/deploy.html**

Shinylive

Shinylive apps use WebAssembly to run entirely in a browser–no need for a special server to run Python.



- Edit and/or host Shinylive apps at **shinylive.io**
- Create a Shinylive version of an app to deploy with `shinylive export myapp site`
  Then deploy to a hosting site like Github or Netlify
- Embed Shinylive apps in Quarto sites, blogs, etc.

```
---
filters:
 - shinylive
---
An embedded Shinylive app:
```{shinylive-python}
#| standalone: true
# [App.py code here…]
```
```

To embed a Shinylive app in a Quarto doc, include the bold syntax.

Outputs

Decorate a function with **@render.*** to reactively render Python outputs

`@render.data_frame`

Species	Island	Bill Length (mm)	Body Mass (g)
Chinstrap	Dream	45.70	3650
Chinstrap	Dream	55.80	4000
Chinstrap	Dream	43.50	3400
Chinstrap	Dream	49.60	3775

`@render.plot`



`@render.code`

```
   area    peri     shape perm
1 4990 2791.90 0.0903296  6.3
2 7002 3892.60 0.1486220  6.3
3 7558 3930.66 0.1833120  6.3
4 7352 3869.32 0.1170630  6.3
```

`@render.text`

Current value: 30

`@render.ui`

Current *value*: **30**

`@render.download`

Download

`@render.image`

posit

And many more via the **shinywidgets** project

`@render_altair`

`@render_plotly`



`@render_bokeh`



`@render_widget`



Reactivity

Reactive values work together with reactive functions. A reactive value must be read from within a reactive functions to avoid the error `No current reactive context`

Module located at `shiny.reactive`

Create a reactive value from other (reactive) values. Helps avoid redundant logic and computation.

Create a reactive UI output.

Perform side effects like logging, updating a database, etc.

```
from shiny import reactive
from shiny.express import (
  input, render, ui
)

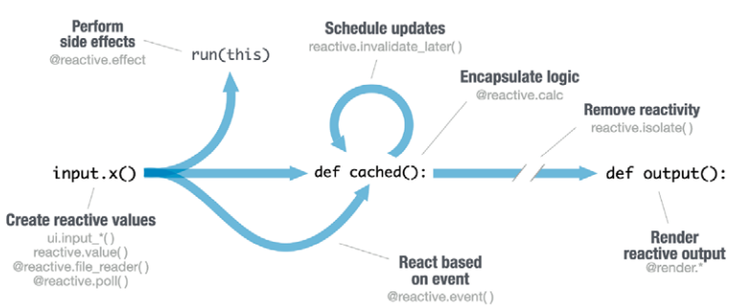
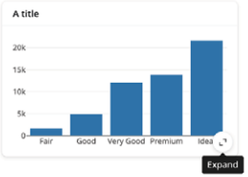
ui.input_text("text", "Enter text")

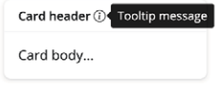
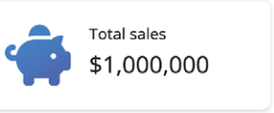
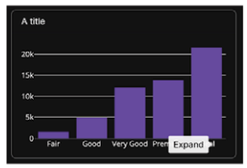
@reactive.calc
def length():
    return len(input.text())

@render.text
def length_output():
    return f"{length()} characters"

@reactive.effect
def length_log():
    print(f"{length()} characters")
```

Reactive functions re-execute when any of their reactive dependencies (i.e., values) change. However, sometimes you want to ignore all but one (i.e., event):

Don't execute until `input.submit` is truthy (i.e. button is clicked)

```
ui.input_text("name", "Enter name")
ui.input_action_button("submit", "Submit")

@render.text
@reactive.event(input.submit)
def greeting():
  return f"Hello {input.name()}!"

@reactive.effect
@reactive.event(input.submit)
def log_name():
  print(f"Name submitted {input.name()}")
```

A `reactive.value()` can be useful for programmatically setting/reading a reactive value. This is often useful when the value can't be derived from `input` values alone.



User Interfaces (UI)

Design delightful UI with a collection of layouts, components, themes, & more.

PAGE LAYOUTS

`with ui.sidebar():`	Sidebar
`with ui.nav_panel():`	Multi-page
`ui.page_opts(`	
`  fillable=True,`	Filling (vertical) layout
`  full_width=True)`	Full-width page

CARDS

Visually group UI elements together with the `card()` component.

```
with ui.card():
  ui.card_header("Title")
  @render.plot
  def plot():
    ...
```



UI LAYOUTS

Multiple columns

`with ui.layout_columns()`	12-col grid
`with ui.layout_column_wrap()`	Equal-width cols
`with ui.layout_sidebar()`	Resizable 2-cols

Multiple panels

```
with ui.navset_card_underline():
  with ui.nav_panel("One"):
    "1st panel"
  with ui.nav_panel("Two"):
    "2nd panel"
  with ui.nav_menu("Menu"):
    with ui.nav_panel("3"):
      "3rd panel"
```

Navigate a set of `nav_panel()`s in various ways with `navset_card_*`

ACCORDIONS

```
with ui.accordion():
 with ui.accordion_panel("One"):
   "1st panel"
 with ui.accordion_panel("Two"):
   "2nd panel"
 with ui.accordion_panel("Three"):
  "3rd panel"
```

Tip: place within `ui.sidebar()` to group similar inputs

TOOLTIPS & ICONS

```
from faicons import icon_svg

with ui.tooltip():
  icon_svg("info-circle")
  "Tooltip message"
```



VALUE BOXES

```
with ui.value_box(
  showcase=icon
):
  "Title"
  "Value"
```



Custom UI

Make the app behave and look exactly how you want it with web tooling and theming

UI as HTML

Shiny UI is powered by HTML (plus JS/CSS):

```
ui.page_fluid(class = "pt-3")
#> <div class="container-fluid pt-3"></div>
```

- Create bespoke experiences with custom HTML (`ui.tags`) and CSS/JS snippets: `ui.include_css()` / `ui.include_js()`.
- Can also interface with popular frameworks like React, Vue, Svelte, etc.

LOCAL FILES

Statically serve any file (image, CSS, JS, etc) by placing them in `www/` dir (next to `app.py`)

THEMES

Choose from set of pre-packaged themes via **shinyswatch** or change main colors / fonts via **brand-yml**

```
color:
  foreground: '#222'
  background: white
  primary: purple
typography:
  fonts:
    - family: Inter
      source: google
```



Gen AI

Build streaming Gen AI interfaces like chatbots and more with the `Chat` and `MarkdownStream` components.

Use `Chat` to implement a streaming chat interface. Provide a callback to generate a response to `user_input` using an AI framework of your choice (e.g., chatlas, LangChain, etc).

```
from chatlas import ChatOpenAI
from shiny.express import ui

chat_client = ChatOpenAI()
chat = ui.Chat("chat")
chat.ui(
  messages=["**Hi!** How can I help?"]
)

@chat.on_user_submit
async def _(user_input: str):
  x = await chat_client.stream_async(
    user_input
  )
  await chat.append_message_stream(x)
```

Express / Core

- An **app.py** that imports from `shiny.express` uses 'Express mode' to make development faster.
- Express extends "Core" Shiny to make UI and server logic one in the same.
- Core may be more suitable for sophisticated apps where a decoupling of UI and server is beneficial.

```
import matplotlib.pyplot as plt
import numpy as np
from shiny import App, ui

app_ui = ui.page_fixed(
  ui.input_slider(
    "n", "Sample Size", 0, 100, 50
  )
)

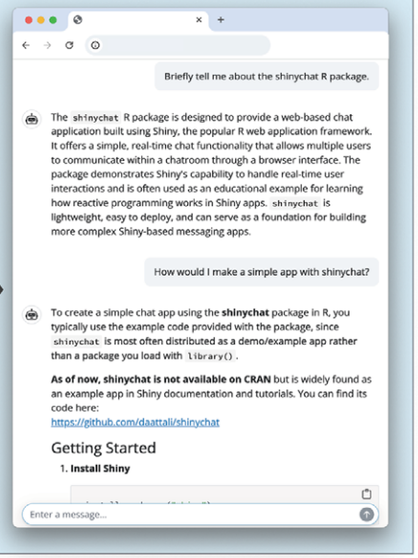
def server(input):
  @render.plot
  def dist():
    x = np.random.randn(input.n())
    plt.hist(x, range=[-3, 3])

app = App(app_ui, server)
```


CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **shiny.posit.co/py** • HTML cheatsheets at **pos.it/cheatsheets** • shiny 1.4.0 • Updated: 2025-07

AI chatbots with shinychat (R) : : CHEAT SHEET

Create a basic chatbot

1. Set up the chatbot UI

Create a **chat UI element** with `chat_ui()`.

2. Create a chat

Initialize a **chat** with a `chat_*()` function, like `chat_openai()`.

3. Handle communication between the user and chatbot

Set up a **reactive listener** that waits for the user to submit a message.

Submit input to the chatbot and **return** asynchronously streaming results.

Append the response to the UI element so the user can see the model's reply.

```
# app.R
library(shiny)
library(shinychat)

ui <- bslib::page_fluid(
  chat_ui("chat")
)

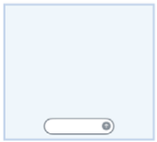
server <- function(input, output, session) {
  chat <- ellmer::chat_openai()

  observeEvent(input$chat_user_input, {
    stream <- chat$stream_async(input$chat_user_input)
    chat_append("chat", stream)
  })
}

shinyApp(ui, server)
```



Save and restore chat state

Bookmark the chat state so the user can return to the conversation if they reload the app.

```
server <- function(input, output, session) {
  chat_client <- ellmer::chat_anthropic()
  chat_restore("chat", chat_client)
  # rest of server function ...
}

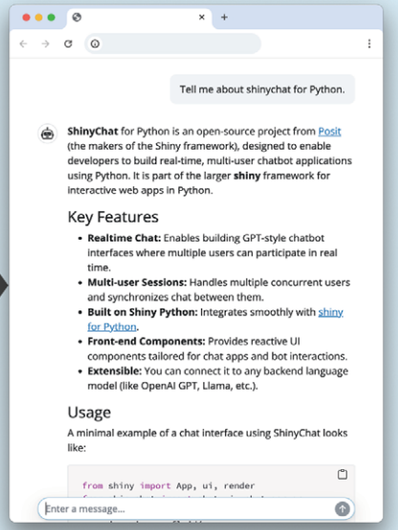
shinyApp(ui, server, enableBookmarking = "url")
```

Update the user input

Programmatically update the content in the user-input box and optionally submit it.

```
observeEvent(input$update_value, {
  update_chat_user_input(
    "chat",
    value = "New input"
  )
})
```

To update the placeholder text, use the `placeholder` argument.

To submit the input, use `submit = TRUE`.

Setup

CHOOSE A MODEL PROVIDER

Initialize a chat and specificy a model provider with an `ellmer::chat_*()` function.

Anthropic Claude: **`chat_anthropic()`**
AWS Bedrock: **`chat_aws_bedrock()`**
Azure OpenAI: **`chat_azure_openai()`**
Databricks: **`chat_databricks()`**
DeepSeek: **`chat_deepseek()`**
Google Gemini: **`chat_google_gemini()`**
Groq: **`chat_groq()`**
Hugging Face: **`chat_huggingface()`**
Ollama: **`chat_ollama()`**
OpenAI: **`chat_openai()`**
perplexity.ai: **`chat_perplexity()`**

Use the model argument to specify a model. e.g.,

```
chat_openai(
  model = "o3"
)
```

Visit https://ellmer.tidyverse.org/#providers to see **all available providers.**

SPECIFY CREDENTIALS

Authenticate with your LLM provider. Many providers require an API key, which you can specify in your **.Renviron** file.

```
#.Renviron

OPENAI_API_KEY=your-key-123
ANTHROPIC_API_KEY=your-key-456
GOOGLE_API_KEY=your-key-789
```

Open .Renviron with `usethis::edit_r_environ()`.

Prompts and messages

ADD A SYSTEM PROMPT

Use a system prompt to tell the chatbot how you'd like it to behave.

```
# prompt.md

You are a helpful assistant who
answers questions briefly…
```

```
chat_*(system_prompt = readLines("prompt.md"))
```

Specify a system prompt using the `system_prompt` argument.

ADD STARTUP MESSAGES AND SUGGESTIONS

Add startup messages or suggestions to the conversation.

```
chat_ui(id = "chat", messages = "**Hello!** How can I help?")
```

Hello! How can I help?

Use messages to supply messages that will appear when the chat starts.

```
chat_ui(
  id = "chat",
  messages = '<span class="suggestion submit">Tell me a joke</span>'
)
```

Tell me a joke ↵

Use the `suggestion` and `submit` CSS classes to suggest inputs to the user and make them clickable.

Layouts

CHOOSE A LAYOUT FOR YOUR CHATBOT

Use functions from the bslib package to create a layout for your chatbot app.

SCREEN-FILLING LAYOUT



```
ui <- page_fillable(
  chat_ui("chat"),
  fillable_mobile = TRUE
)
```

Chat fills the page.

SIDEBAR LAYOUT

```
ui <- page_sidebar(
  sidebar = sidebar(
    chat_ui("chat", height = "100%"),
    style = "height: 100%;"
  ),
  fillable = TRUE
)
```

Chat fills the sidebar.

CARD LAYOUT

```
ui <- page_fillable(
  card(
    chat_ui(id = "chat")
  ),
  fillable_mobile = TRUE
)
```

Chat appears inside a card.

AI chatbots with shinychat (**Python**) : : CHEAT SHEET

Create a basic chatbot

1. Set up the chatbot UI

Create and display the **chat interface.**

2. Create a chat

Initialize a **Chat client** with a `Chat*()` function like `ChatOpenAI()`.

3. Handle communication between the user and chatbot

Define a callback that runs **when the user submits a message.**

Submit input to the chatbot and **return** asynchronously streaming results.

Append the response to the UI element so the user can see the model's reply.

```
# app.py
from chatlas import ChatOpenAI
from shiny.express import ui

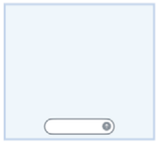
chat = ui.Chat(id="my_chat")
chat.ui()

chat_client = ChatOpenAI()

@chat.on_user_submit
async def handle_user_input(user_input: str):
    response = await chat_client.stream_async(
        user_input
    )
    await chat.append_message_stream(
        response
    )
```

This cheatsheet uses **Shiny Express**. To see examples for Shiny Core, see https://shiny.posit.co/py/docs/genai-chatbots.html.



Save and restore chat state

Bookmark the chat state so the user can return to the conversation if they reload the app.

```
chat_client = ChatOpenAI()
chat = ui.Chat(id="chat")

chat.enable_bookmarking(
  chat_client,
  bookmark_store="url", # or "server"
  bookmark_on="response", # or None
)
```

Update the user input

Programmatically update the content in the user-input box and optionally submit it.

```
@reactive.effect
@reactive.event(input.update_value)
def _():
  chat.update_user_input(
    value="New input"
  )
```

To update the placeholder text, use the `placeholder` parameter.

To submit the input, use `submit=True`.

Setup

CHOOSE A MODEL PROVIDER

Initialize a chat and specificy a model provider with a `chatlas` function.

Anthropic Claude: **`ChatAnthropic()`**
AWS Bedrock: **`ChatBedrockAnthropic()`**
Azure OpenAI: **`ChatAzureOpenAI()`**
Databricks: **`ChatDatabricks()`**
Google Gemini: **`ChatGoogle()`**
Groq: **`ChatGroq()`**
Ollama: **`ChatOllama()`**
OpenAI: **`ChatOpenAI()`**
perplexity.ai: **`ChatPerplexity()`**

You can also use other AI frameworks, like LangChain or Pydantic, to handle model communciation.

Use the `model` parameter to specify a model. e.g., `ChatOpenAI(model="o3")`

Visit https://posit-dev.github.io/chatlas/reference to see **all available providers.**

SPECIFY CREDENTIALS

Authenticate with your LLM provider. Many providers require an API key, which you can specify in a **.env** file and load with **dotenv**.

```
#.env

OPENAI_API_KEY=your-key-123
ANTHROPIC_API_KEY=your-key-456
GOOGLE_API_KEY=your-key-789
```

See the provider's Chat function documentation for specifics on authentication.

Prompts and messages

ADD A SYSTEM PROMPT

Use a system prompt to tell the chatbot how you'd like it to behave.

```
# prompt.md

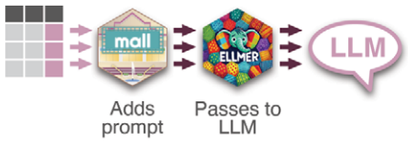
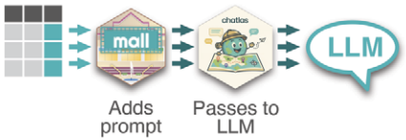
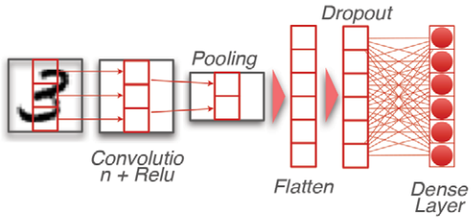
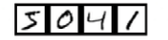
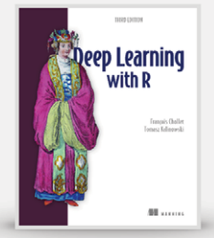
You are a helpful assistant who
answers questions briefly…
```

```
ChatOpenAI(system_prompt=open("prompt.md", "r").read())
```

Specify a system prompt using the `system_prompt` parameter.

ADD STARTUP MESSAGES AND SUGGESTIONS

Add startup messages or suggestions to the conversation.

```
chat.ui(messages=["**Hello!** How can I help?"])
```

Hello! How can I help?

Use messages to supply messages that will appear when the chat starts.

```
chat.ui(
  messages=['<span class="suggestion submit">Tell me a joke</span>']
)
```

Tell me a joke

Use the `suggestion` and `submit` CSS classes to suggest inputs to the user and make them clickable.

Layouts

CHOOSE A LAYOUT FOR YOUR CHATBOT

Use functions from the bslib package to create a layout for your chatbot app.

SCREEN-FILLING LAYOUT



```
ui.page_opts(
  fillable=True,
  fillable_mobile=True,
)

chat = ui.Chat(id="chat")
chat.ui()
```

Chat fills the page.

SIDEBAR LAYOUT

```
chat = ui.Chat(id="chat")

with ui.sidebar(width=300, style="height:100%"):
  chat.ui(height="100%")

"Main content"
```

Chat fills the sidebar.

CARD LAYOUT

```
ui.page_opts(
  fillable=True,
  fillable_mobile=True
)

chat = ui.Chat(id="chat")

with ui.card():
    chat.ui()
```

Chat appears inside a card that fills the page.

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **https://posit-dev.github.io/shinychat/** • HTML cheatsheets at **pos.it/cheatsheets** • shinychat 0.2.0 • Updated: 2025-07

Natural Language Processing using LLMs in R : : CHEAT

Intro

Use LLM's to perform NLP row-wise over a data frame. **`mall`** comes with pre-defined prompts that perform specific NLP operations, and then places the results in a new column. Use **OpenAI**, **Ollama**, **Anthropic** and many others thanks to its integration with **`ellmer`**:



`mall`'s data frame functions are designed with 'tidy' principals in mind, so they work will **`tidyverse`** packages.

`mall` also includes functions that work with string vectors.

Getting Started

1. Load the libraries

```
library(mall)
library(ellmer)
```

2. Create a vendor specific chat connection

```
chat <- chat_openai()
```

3. Pass the chat object to `mall`

```
llm_use(chat)
```

For dataframes:

```
data("reviews") # Sample product reviews
reviews |>
  llm_sentiment(review)
```

For vectors:

```
llm_vec_sentiment(c("I am happy", "I am sad"))
```

CONNECT AUTOMATICALLY

As a convenience, mall is able to automatically establish a connection with the LLM. To do this you can use the ***.mall_chat*** option:

```
options(.mall_chat=ellmer::chat_openai(model="gpt-4o"
))
```

Add this line to your **.Rprofile** file in order for that code to run every time you start R. You can call **`usethis::edit_r_profile()`** to edit.

NLP Functions

SENTIMENT ANALYSIS

llm_sentiment(.data, col, options = c("positive", "negative", "neutral"), pred_name = ".sentiment", additional_prompt = "**)**
llm_sentiment(reviews, review)

llm_vec_sentiment(x, options = c("positive", "negative", "neutral"), additional_prompt = "", preview = FALSE**)**
llm_vec_sentiment(c("I am happy", "I am sad"))

Special arguments:

options - Customize the sentiments to check for: `options = c("positive", "negative")`. Use 'tilde' to mask the results, for example `c("positive" ~ 1, "negative" ~ 0))` returns 1 for positive and 0 for negative.

EXTRACT

Extract specific entity, or entities, from the provided text

llm_extract(.data, col, labels, expand_cols = FALSE, additional_prompt = "", pred_name = ".extract"**)**
llm_extract(reviews, review, labels = "product")

llm_vec_extract(x, labels = c(), additional_prompt = "", preview = FALSE**)**
llm_vec_extract("bob smith, 123 3rd street", c("name", "address"))

Special arguments:

labels - A vector to specify the entities to identify

expand_cols - If multiple labels, this indicates if the labels will show up in their own column (data frames only)

SUMMARIZE

Summarize text into a specified number of words

llm_summarize(.data, col, max_words = 10, pred_name = ".summary", additional_prompt = "")
llm_summarize(reviews, review, max_words = 5)

llm_vec_summarize(x, max_words = 10, additional_prompt = "", preview = FALSE**)**
llm_vec_summarize("This has been the best TV I've ever used. Great screen, and sound.", max_words = 5)

VERIFY

Check if a statement is true or not based on the provided text

llm_verify(.data, col, what, yes_no = factor(c(1, 0)), pred_name = ".verify", additional_prompt = "")
llm_verify(reviews, review, "is the customer happy")

llm_vec_verify(x, what, yes_no = factor(c(1, 0)), additional_prompt = "", preview = FALSE**)**
llm_verify(c("I am happy", "I am sad"), "is the person happy")

Special arguments:

yes_no - Customize what it returns for true/ false with a vector `yes_no = c("y", "n")`.

CLASSIF

Classify the provided text as one of the options provided via the `labels`

llm_classify(.data, col, labels, pred_name = ".classify", additional_prompt = "")
llm_classify(reviews, review, c("appliance", "computer"))

llm_vec_classify(x, labels, additional_prompt = "", preview = FALSE**)**
llm_vec_classify(c("this is important!", "just whenever"), c("urgent", "not urgent"))

Special arguments:

labels - A character vector with at least 2 labels to classify the text as

TRANSLATE

Translate into target language

llm_translate(.data, col, language, pred_name = ".translation", additional_prompt = "")
llm_translate(reviews, review, "spanish")

llm_vec_translate(x, language, additional_prompt = "", preview = FALSE**)**
llm_vec_translate("grass is green", "spanish")

Special arguments:

language - Target language. No origin language is passed since the LLM detects it automatically.

CUSTOM

Create your own prompt

llm_custom(.data, col, prompt = "", pred_name = ".pred", valid_resps = "")
my_prompt <-"Answer a question. Return only the answer, no explanation. Acceptable answers are 'yes', 'no'. Answer this about the following text, is this a happy customer?:"
llm_custom(reviews , review, my_prompt)

llm_vec_custom(x, prompt = "", valid_resps = NULL**)**

Special arguments:

valid_resps - A vector to specify the set of answers expected back. `mall` will change those not in the set to NA.

SHARED ARGUMENTS

additional_prompt - Appends instructions to the LLM.
llm_classify(reviews, review, c("appliance", "computers"), additional_prompt = "Consider TVs as appliances")

pred_name - Name of the new column. Defaults are set based on the NLP operation. *(Data frames only)*
llm_vec_translate("grass is green", "spanish", pred_name = "in_spanish")

preview - Returns what it would be sent to the LLM instead *(Vectors only)*

Other Features

OLLAMA DIRECT

If Ollama is the only LLM provider you are using, then a simplified way to connect is available which does not require an `ellmer` Chat object. Simply pass "ollama" as the `backend`, and specify the model:

```
llm_use("ollama", model= "llama3.2")
```

CACHING RESULTS

By default, `mall` saves the LLM results in a temp folder. To specify a folder call:

```
llm_use(chat, .cache = "<my folder>")
```

To turn off use:

```
llm_use(chat, .cache = "")
```

Natural Language Processing using LLMs in Python : :

Intro

Use LLM's to perform NLP row-wise over a data frame. **`mall`** comes with pre-defined prompts that perform specific NLP operations, and then places the results in a new column. Use **OpenAI**, **Ollama**, **Anthropic** and many others thanks to its integration with **`chatlas`**:



`mall` works as an extension for Polars data frames. It also works with string vectors.

Getting Started

Start by creating a new LLM connection

```
from chatlas import ChatOpenAI
chat = ChatOpenAI()
```

DATA FRAMES

1. Load the library

```
import mall
```

2. Read or load your data frame

```
reviews = mall.MallData.reviews # Sample product reviews
```

3. Pass the chat object to `mall`

```
reviews.llm.use(chat)
```

4. Access NLP functions via `.llm`

```
reviews.llm.sentiment('review')
```

STRING VECTORS

1. Load the `LLMVec` class

```
from mall import LLMVec
```

2. Create a new `LLMVec` object

```
llm = LLMVec(chat)
```

3. Pass a vector to a function in the new object

```
llm.sentiment(['I am happy', 'I am sad'])
```

NLP Functions

SENTIMENT ANALYSIS

<DataFrame>.**llm.sentiment(**col, options = ['positive', 'negative', 'neutral'], additional='', pred_name ='sentiment'**)**
reviews.llm.sentiment("review")

< LLMVec object >.**sentiment(**x, options=['positive', 'negative', 'neutral'], additional=''**)**
llm.sentiment(['I am happy', 'I am sad'])

Special arguments:

options - Customize the sentiments to check for: `options = ["positive", "negative"]`. Use a DICT object to mask the results, for example `{"positive" : 1, "negative" : 0}` returns 1 for positive and 0 for negative.

EXTRACT

Extract specific entity, or entities, from the provided text

<DataFrame>.**llm.extract(**col, labels='', expand_cols = False, additional = '', pred_name = 'extract'**)**
reviews.llm.extract("review", labels = "product")

< LLMVec object >.**extract(**x, labels='', additional=''**)**
llm.extract(['bob smith, 123 3rd street'], labels=['name', 'address'])

Special arguments:

labels - A vector to specify the entities to identify

expand_cols - If multiple labels, this indicates if the labels will show up in their own column (data frames only)

SUMMARIZE

Summarize text into a specified number of words

<DataFrame>.**llm.summarize(**x, max_words=10, additional=''**)**
reviews.llm.summarize("review", 5)

< LLMVec object >.**summarize(**x, max_words=10, additional=''**)**
llm.summarize(['This has been the best TV Ive ever used. Great screen, and sound.'], max_words = 5)

VERIFY

Check if a statement is true or not based on the provided text

<DataFrame>.**llm.verify(**col, what='', yes_no=[1, 0], additional='', pred_name='verify'**)**
reviews.llm.verify("review", "is the customer happy")

< LLMVec object >.**verify(**x, what='', yes_no=[1, 0], additional=''**)**
llm.verify(['I am happy', 'I am sad'], what = 'Is the person happy?')

Special arguments:

yes_no - Customize what it returns for true/false with a vector `yes_no = ["y", "n"]`.

CLASSIFY

Classify the provided text as one of the options provided via the `labels`

<DataFrame>.**llm.classify(**col, labels='', additional='', pred_name='classify'**)**
reviews.llm.classify("review", ["appliance", "computer"])

< LLMVec object >.**classify(**x, labels='', additional=''**)**
llm.classify(['this is important!', 'there is no rush'], ['urgent', 'not urgent'])

Special arguments:

labels - A character vector with at least 2 labels to classify the text as

TRANSLATE

Translate into target language

<DataFrame>.**llm.translate(**col, language='', additional='', pred_name='translation'**)**
reviews.llm.translate("review", "spanish")

< LLMVec object >.**translate(**x, language='', additional=''**)**
llm.translate(['the grass is green'], language = 'spanish')

Special arguments:

language - Target language. No origin language is passed since the LLM detects it automatically.

CUSTOM

Create your own prompt

<DataFrame>.**llm.custom(**col, prompt='', valid_resps='', pred_name='custom'**)**
my_prompt ='Answer a question. Return only the answer, no explanation. Acceptable answers are 'yes', 'no'. Answer this about the following text, is this a happy customer?:'
reviews.llm.custom("review", prompt = my_prompt)

< LLMVec object >.**custom(**x, prompt='', valid_resps=''**)**

Special arguments:

valid_resps - A vector to specify the set of answers expected back.

SHARED ARGUMENTS

additional - Appends more instructions to the LLM.
reviews.llm.classify("review", ["appliance", "computer"],additional="Consider TVs as appliances")

pred_name - Name of the new column. Defaults are set based on the NLP operation. *(Data frames only)*

Other Features

OLLAMA DIRECT

If Ollama is the only LLM provider you are using, then a simplified way to connect is available which does not require an `ellmer` Chat object. Simply pass "ollama" as the `backend`, and specify the model:

Dataframe:
```
reviews.llm.use('ollama','llama3.2')
```
Vector:
```
llm = LLMVec('ollama','llama3.2')
```

CACHING RESULTS

By default, `mall` saves the LLM results in a temp folder. To specify a folder call:

Dataframe:
```
reviews.llm.use(chat, _cache='<my folder>')
```
Vector:
```
llm = LLMVec(chat, _cache='<my folder>')
```

To turn off use:

Dataframe:
```
reviews.llm.use(chat, _cache='')
```
Vector:
```
llm = LLMVec(chat, _cache='')
```

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **mlverse.github.io/mall** • HTML cheatsheets at **pos.it/cheatsheets** • mall 0.2.0 • Updated: 2025-07

Deep Learning with *keras3* : : CHEAT SHEET

Intro

Keras enables fast experimentation with "neural networks". It supports *convolution networks* (vision) and *recurrent networks* (text and time series). It provides the freedom to work with JAX, Tensorflow, and Torch, plus the freedom to build models that can seamlessly move across these frameworks.

keras3 provides easy access to the Keras vast API. It also makes getting started with Keras faster by automatically creating a working Python environment with all the needed libraries pre-installed.

Model building

1. DEFINE A MODEL

keras_input() / keras_model() - Defines a Functional Model with inputs and outputs.
inputs <- keras_input(<input-shape>)
outputs <- inputs |> layer_dense() |> layer_...
model <- keras_model(inputs, outputs)

keras_model_sequential() - Define a Sequential Model composed of a linear stack of layers
model <- keras_model_sequential(<input-shape>) |> layer_dense() |> layer_...

Model() - Subclass the base Model class

2. INSPECT THE MODEL

summary(<model>)- A summary of a Keras Model

plot(<model>)- Plot the model. Needs graphviz to work: graphviz.gitlab.io/download

3. COMPILE THE MODEL

compile()- Configure aspects of the model such as optimizer, loss, metrics, weights and others.
model |> compile(loss = "categorical_crossentropy", optimizer = "rmsprop", metrics = "accuracy")

4. FIT THE MODEL

fit()- Trains the model for a fixed number of dataset iterations (epochs)
model |> fit(<x>, <y>, epochs = 30, batch_size = 128, validation_split = 0.2)

5. EVALUATE THE MODEL

evaluate(<model>)- Returns the loss and metrics
model |> evaluate(<x test>,<y test>)

predict(<model>, x)- Generates predictions

Layers

CORE LAYERS

layer_dense() - A layer connected to all neurons in the preceding layer

layer_einsum_dense() - A dense layer of arbitrary dimensionality

layer_embedding() - Acts as a mapping function, it stores a dense vector for each word in the vocabulary

layer_lambda() - Allows arbitrary expressions to be used as a layer

layer_masking() - Masks a sequence by using a mask value to skip time steps

CONVOLUTION LAYERS

Layers that create a convolution kernel that is convolved with the layer input over one, two, or three dimensions to produce a tensor of outputs.

layer_conv_1d() /
layer_conv_1d_transpose() -

Layer of a single dimension (temporal). Transpose does the opposite, deconvolution.

layer_conv_2d() /
layer_conv_2d_transpose() -

Two dimensional layer (image). Transpose does the opposite, deconvolution.

layer_conv_3d() /
layer_conv_3d_transpose() -

Three dimensional layer (images over volumes). Transpose does the opposite, deconvolution.

layer_depthwise_conv_1d() / layer_depthwise_conv_2d() - A type of convolution in which each input channel is convolved with a different kernel.
layer_separable_conv_1d() / layer_separable_conv_2d() - A depthwise convolution that acts separately on channels, followed by a pointwise convolution that mixes channels.

NORMALIZATION LAYERS

layer_batch_normalization() - Operates across the batch dimension

layer_layer_normalization() - Operates across the feature dimension

layer_group_normalization() - Operates across channels

layer_spectral_normalization() - Controls the Lipschitz constant of the weights

layer_rms_normalization() - Root Mean Square

REGULARIZATION LAYERS

layer_batch_normalization() - Maintains the mean output close to 0 and the output standard deviation close to 1.

layer_gaussian_noise()

Layers that randomly "drop" a fraction of input units during training by setting them to 0

layer_dropout() - Non 0 inputs are scaled up
layer_alpha_dropout() - Keeps original mean and variance

layer_gaussian_dropout() - 1-centers Gaussian noise
layer_spatial_dropout_1d() /
layer_spatial_dropout_2d() /

layer_spatial_dropout_3d() - Drops entire dimensional feature maps instead of individual elements

POOLING LAYERS

Layers that reduce each dimension of the input while retaining the most important features

layer_max_pooling_1d() /
layer_max_pooling_2d() /
layer_max_pooling_3d() - Maximum value over a specified window size (pool size)

layer_average_pooling_1d() /
layer_average_pooling_2d() /
layer_average_pooling_3d()
Average value over a specified window size (pool size)

layer_global_max_pooling_1d() /
layer_global_max_pooling_2d() /
layer_global_max_pooling_3d()
Maximum value of the dimension

layer_global_average_pooling_1d() /
layer_global_average_pooling_2d() /
layer_global_average_pooling_3d()
Average value of the dimension

Convolution Neural Network (CNN)



TRAIN IMAGE RECOGNIZER ON MNIST DATA

a.k.a. deep learning's 'hello world' example



```
library(keras3)
# Input layer: use MNIST images
.[.[x_train, y_train], .[x_test, y_test]] <-
  dataset_mnist()
# Rescale and categorize
x_train <- x_train/255; x_test <- x_test/255
num_classes <- 10
y_train <- to_categorical(y_train, num_classes)
y_test <- to_categorical(y_test, num_classes)
# Define the model and layers
model <-
  keras_model_sequential(
    input_shape = c(28, 28, 1)) |>
  layer_conv_2d(
    filters = 32, kernel_size = c(3, 3),
    activation = "relu") |>
  layer_max_pooling_2d(pool_size = c(2, 2)) |>
  layer_conv_2d(
    filters = 64,
    kernel_size = c(3, 3),
    activation = "relu") |>
  layer_max_pooling_2d(pool_size = c(2, 2)) |>
  layer_flatten() |>
  layer_dropout(rate = 0.5) |>
  layer_dense(units = num_classes,
              activation = "softmax")
# Inspect the model
summary(model)
plot(model)
# Compile (define loss and optimizer)
model |>
  compile(
    loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(),
    metrics = c("accuracy")
  )
# Fit the model
model |> fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
# Evaluate and predict
model |> evaluate(x_test, y_test)
model |> predict(x_test)
# Save the full model
save_model(model, "mnist-classifier.keras")
# Deploy for serving inference
dir.create("mnist")
export_savedmodel(model, "mnist/1")
rsconnect::deployTFModel("mnist")
```

Deep Learning with *keras3* : : CHEAT SHEET

Preprocessing Layers

NUMERICAL FEATURES

layer_normalization() -Normalizes continuous features

layer_discretization() - Buckets by ranges

CATEGORICAL FEATURES

layer_category_encoding() - Encodes integer features

layer_hashing() - Hashes and bins features

layer_hashed_crossing() - Crosses features using the “hashing trick”

layer_string_lookup() / layer_integer_lookup - Maps a set of arbitrary text or integers via a table-based vocabulary lookup

DATA FRAMES

layer_feature_space() - Main function to specify the how the features are to be pre-processed

```
feature_space <- layer_feature_space(
    features = list(float_var = feature_float_normalized(),
                    string_var = feature_string_categorical(),
                    int_var = feature_integer_categorical()))
```

feature_float()

feature_float_rescaled()

feature_float_normalized()

feature_float_discretized(num_bins)

feature_integer_categorical()

feature_string_categorical()

adapt() - Calculates the the normalizations, bins, and other conversions against the data set

adapt(feature_space, <TF dataset>)

TEXT

layer_text_vectorization() / get_vocabulary() / set_vocabulary() - Maps a set of arbitrary text or integers via a table-based vocabulary lookup

AUDIO

layer_mel_spectrogram() -Converts signal to Mel

IMAGES

layer_resizing() - Changes the size of images

layer_rescaling() - Multiplies scale by a given number

layer_center_crop() - Crops images to the given size

layer_auto_contrast() - Makes differences between pixels more obvious

IMAGE AUGMENTATION

layer_aug_mix()
layer_cut_mix()
layer_mix_up()
layer_solarization()
layer_random_contrast()
layer_random_crop()
layer_random_erasing()
layer_random_flip()
layer_random_grayscale()
layer_random_hue()
layer_random_invert()
layer_random_rotation()
layer_random_shear()
layer_random_zoom()
layer_random_saturation()
layer_random_sharpness()
layer_random_translation()
layer_random_brightness()
layer_random_color_jitter()
layer_random_perspective()
layer_random_posterization()
layer_random_gaussian_blur()
layer_random_color_degeneration()

Data Loading

text_dataset_from_directory()

audio_dataset_from_directory() - Reads .wav files

image_dataset_from_directory() - This function supports JPEG, Bitmap, PNG and GIF

image_load() - Loads image into a PIL format

image_from_array() - Converts array into PIL format

image_to_array() - Converts PIL image to array

timeseries_dataset_from_array() - Creates a dataset of sliding windows over a time series

pad_sequences() - Pads sequences to the same length

Save and Deploy

I/O OPERATIONS

save_model(“<path>”) / load_model(“<path>”) - Manage models using the ".keras" file format.

save_model_weights() / load_model_weights() - Manage model weights to/from ".h5" files.

save_model_config() / load_model_config() - Manage model architecture to/from a ".json" file.

DEPLOY

export_savedmodel(<model>, "<path>/1") - Save a TF SavedModel for inference.

rsconnect::deployTFModel(“<path>”) - Deploy a TF SavedModel to Posit Connect for inference.

Operations

Introduced in Keras v.3, these are low-level operations that will work the same in JAX, TF and Torch

CORE

op_associative_scan() - Faster op_scan() if the function performs a binary associative operation

op_cast() - Casts to specified dtype

op_cond() - Conditionally applies one of two functions

op_convert_to_array() - Tensor to R array

op_convert_to_numpy() - Tensor to Numpy array

op_convert_to_tensor() - Array to tensor

op_custom_gradient() - allows fine grained control over the gradients of a sequence for operations.

op_dtype() - Returns tensor’s dtype

op_fori_loop() - For loop, return tensor

op_is_tensor() - Checks for backend specific tensor

op_map() - Applies a function to every single value in the tensor

op_rearrange() - Rearranges tensor to specification

op_scan() - Maps a function, but retaining state

op_scatter() - Modifies tensor in specified locations with zeroes

op_scatter_update() - It modifies tensor in specified locations that do not need to be contiguous

op_searchsorted() - Performs a binary search

op_shape() - Returns tensor’s shape

op_slice() - Extract specific location within the tensor

op_slice_update() - It modifies tensor in a specified location

op_stop_gradient() - Stops gradient computation

op_subset() - Get specific section of the tensor

op_switch() - Applies one of several functions passed to the function based on an index argument

op_unstack() - Splits tensor into a list of tensors separated along the specified axis

op_vectorized_map() - Applies a function to every single value in the tensor. Designed for vectorized operations

op_while_loop() - While loop implementation

IMAGES

op_image_affine_transform() - Applies the given transform

op_image_crop() - Crops to specified height and width

op_image_gaussian_blur() - Applies Gaussian blur

op_image_extract_patches() - Extracts from image

op_image_hsv_to_rgb() - Converts from HSV to RGB

op_image_map_coordinates() - Maps input array to new coordinates by interpolation

op_image_pad() - Pad images with zeroes to specified height and width

op_image_perspective_transform() - Applies a perspective transformation to the image

op_image_resize() - Resize image to size using the specified interpolation method

op_image_rgb_to_grayscale() - Converts from RGB to grayscale

op_image_rgb_to_hsv() - Converts from RGB to HSV

NEURAL NETWORK

op_average_pool()
op_batch_normalization()
op_binary_crossentropy()
op_celu()
op_conv()
op_conv_transpose()
op_ctc_loss()
op_depthwise_conv()
op_elu()
op_gelu()
op_glu()
op_hard_shrink()
op_hard_sigmoid()
op_hard_silu()
op_hard_tanh()
op_leaky_relu()
op_log_sigmoid()
op_log_softmax()
op_max_pool()
op_moments()
op_multi_hot()
op_normalize()
op_one_hot()
op_polar()
op_psnr()
op_relu()
op_relu6()
op_rms_normalization()
op_selu()
op_separable_conv()
op_sigmoid()
op_silu()
op_soft_shrink()
op_softmax()
op_softplus()
op_softsign()
op_sparse_plus()
op_sparsemax()
op_squareplus()
op_tanh_shrink()
op_threshold()
op_unravel_index()
op_categorical_crossentropy()
op_sparse_categorical_crossentropy()

Learn More

Deep Learning with R
3rd Edition



CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **keras.posit.co** • HTML cheatsheets at **pos.it/cheatsheets** • keras3 1.4.0 • Updated: 2025-07

REST APIs with plumber: : CHEATSHEET

Introduction to REST APIs

Web APIs use **HTTP** to communicate between **client** and **server**.

HTTP

HTTP is built around a **request** and a **response**. A **client** makes a request to a **server**, which handles the request and provides a response. Requests and responses are specially formatted text containing details and data about the exchange between client and server.

REQUEST

```
curl -v "http://httpbin.org/get"

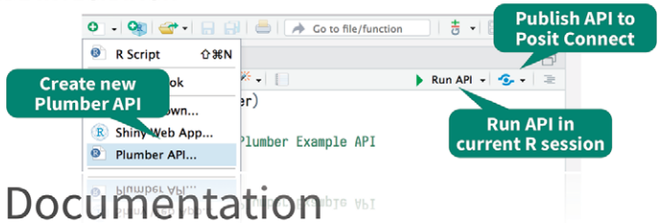
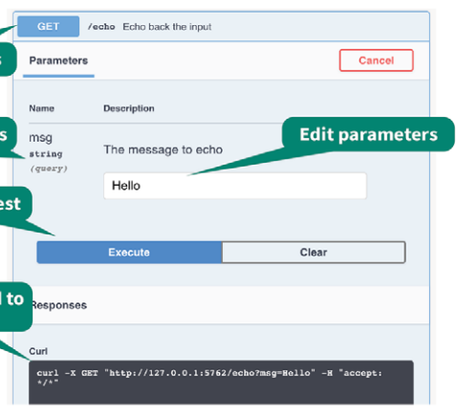
#> GET  /  get HTTP/1.1
#> Host: httpbin.org
#> User-Agent: curl/7.55.1
#> Accept: */*
#
# Request Body
```

HTTP Method · Path · HTTP Version · Headers · Message body

RESPONSE

```
#< HTTP/1.1 200 OK
#< Connection: keep-alive
#< Date: Thu, 02 Aug 2018 18:22:22 GMT
#
# Response Body
```

HTTP Version · Status code · Reason phrase · Headers · Message body

Plumber: Build APIs with R

Plumber uses special comments to turn any arbitrary R code into API endpoints. The example below defines a function that takes the `msg` argument and returns it embedded in additional text.

```
library(plumber)

#* @apiTitle Plumber Example API

#* Echo back the input
#* @param msg The message to echo
#* @get /echo
function(msg = "") {
  list(
    msg = paste0(
      "The message is: '", msg, "'")
  )
}
```

Plumber comments begin with #* · @ decorators define API characteristics · HTTP Method · /<path> is used to define the location of the endpoint

Plumber pipeline

Plumber endpoints contain R code that is executed in response to an HTTP request. Incoming requests pass through a set of mechanisms before a response is returned to the client.

FILTERS

Filters can forward requests (after potentially mutating them), throw errors, or return a response without forwarding the request. Filters are defined similarly to endpoints using the `@filter [name]` tag. By default, filters apply to all endpoints. Endpoints can opt out of filters using the `@preempt` tag.

PARSER

Parsers determine how Plumber parses the incoming request body. By default Plumber parses the request body as JavaScript Object Notation (JSON). Other parsers, including custom parsers, are identified using the `@parser [parser name]` tag. All registered parsers can be viewed with `registered_parsers()`.

ENDPOINT

Endpoints define the R code that is executed in response to incoming requests. These endpoints correspond to HTTP methods and respond to incoming requests that match the defined method.

METHODS

- `@get` - request a resource
- `@post` - send data in body
- `@put` - store / update data
- `@delete` - delete resource
- `@head` - no request body
- `@options` - describe options
- `@patch` - partial changes
- `@use` - use all methods

SERIALIZER

Serializers determine how Plumber returns results to the client. By default Plumber serializes the R object returned into JavaScript Object Notation (JSON). Other serializers, including custom serializers, are identified using the `@serializer [serializer name]` tag. All registered serializers can be viewed with `registered_serializers()`.

```
library(plumber)

#* @filter log
function(req, res) {
  print(req$HTTP_USER_AGENT)
  forward()
}

#* Convert request body to uppercase
#* @preempt log
#* @parser json
#* @post /uppercase
#* @serializer json
function(req, res) {
  toupper(req$body)
}
```

Identify as filter · Filter name · Forward request · Endpoint description · Opt out of the log filter · Parser · HTTP Method · Endpoint path · Serializer

Running Plumber APIs

Plumber APIs can be run programmatically from within an R session.

```
library(plumber)

plumb("plumber.R") %>%
  pr_run(port = 5762)
```

Path to API definition · Specify API port

This runs the API on the host machine supported by the current R session.

IDE INTEGRATION



Create new Plumber API · Publish API to Posit Connect · Run API in current R session

Documentation

Plumber APIs automatically generate an OpenAPI specification file. This specification file can be interpreted to generate a dynamic user-interface for the API. The default interface is generated via Swagger.



Endpoint details · Parameter details · Edit parameters · Send request · curl command used to send request

Interact with the API

Once the API is running, it can be interacted with using any HTTP client. Note that using `httr` requires using a separate R session from the one serving the API.

```
(resp <- httr::GET("localhost:5762/echo?msg=Hello"))
#> Response [http://localhost:5762/echo?msg=Hello]
#>   Date: 2018-08-07 20:06
#>   Status: 200
#>   Content-Type: application/json
#>   Size: 35 B
httr::content(resp, as = "text")
#> [1] "{\"msg\":[\"The message is: 'Hello'\"]}"
```

Programmatic Plumber

Tidy Plumber

Plumber is exceptionally customizable. In addition to using special comments to create APIs, APIs can be created entirely programatically. This exposes additional features and functionality. Plumber has a convenient "tidy" interface that allows API routers to be built piece by piece. The following example is part of a standard **plumber.R** file.

```
library(plumber)

#* @plumber
function(pr) {
  pr %>%
    pr_get(path = "/echo",
           handler = function(msg = "") {
             list(msg = paste0(
               "The message is: '",
               msg,
               "'")
             )
           }) %>%
    pr_get(path = "/plot",
           handler = function() {
             rand <- rnorm(100)
             hist(rand)
           },
           serializer = serializer_png()) %>%
    pr_post(path = "/sum",
            handler = function(a, b) {
              as.numeric(a) + as.numeric(b)
            })
}
```

Use @plumber tag

Function that accepts and modifies a plumber router

"Tidy" functions for building out Plumber API

OpenAPI

Plumber automatically creates an OpenAPI specification file based on Plumber comments. This file can be further modified using **pr_set_api_spec()** with either a function that modifies the existing specification or a path to a **.yaml** or **.json** specification file.

```
library(plumber)

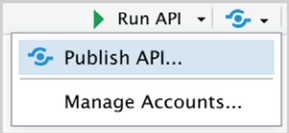
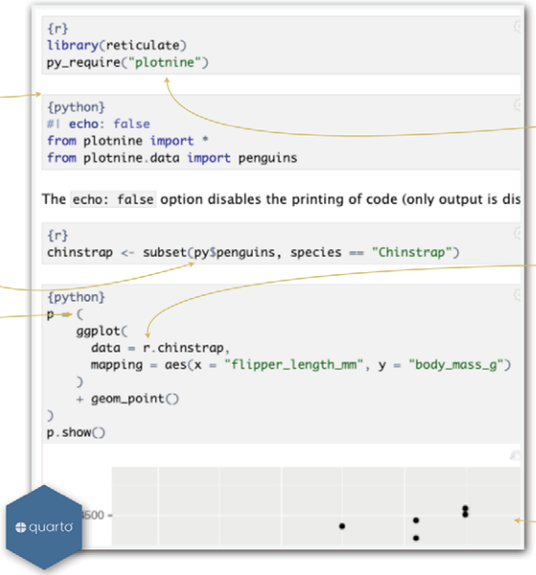
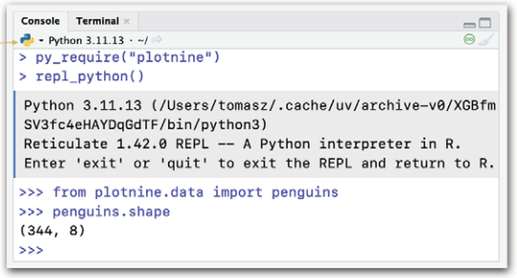
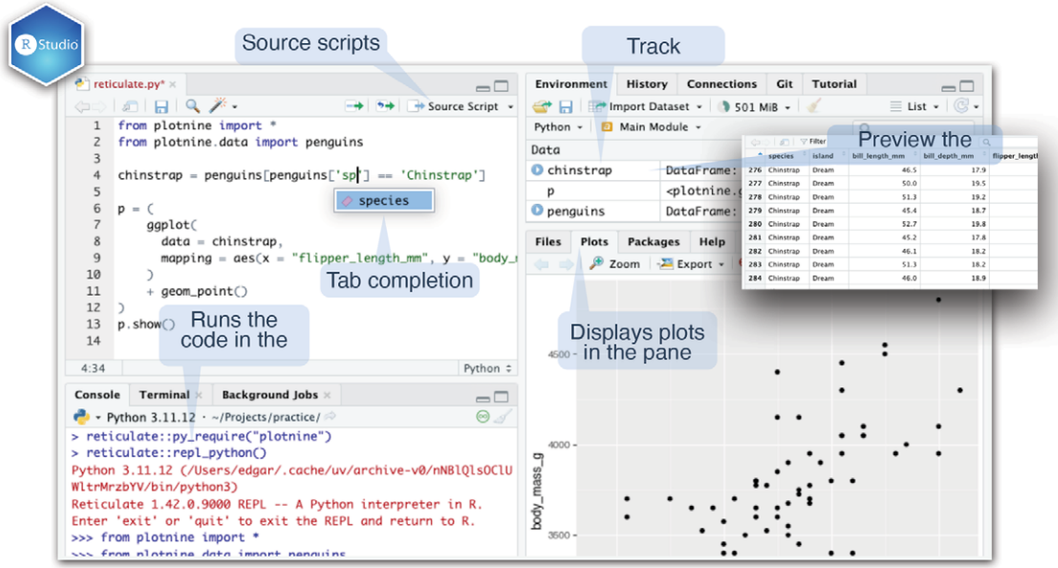
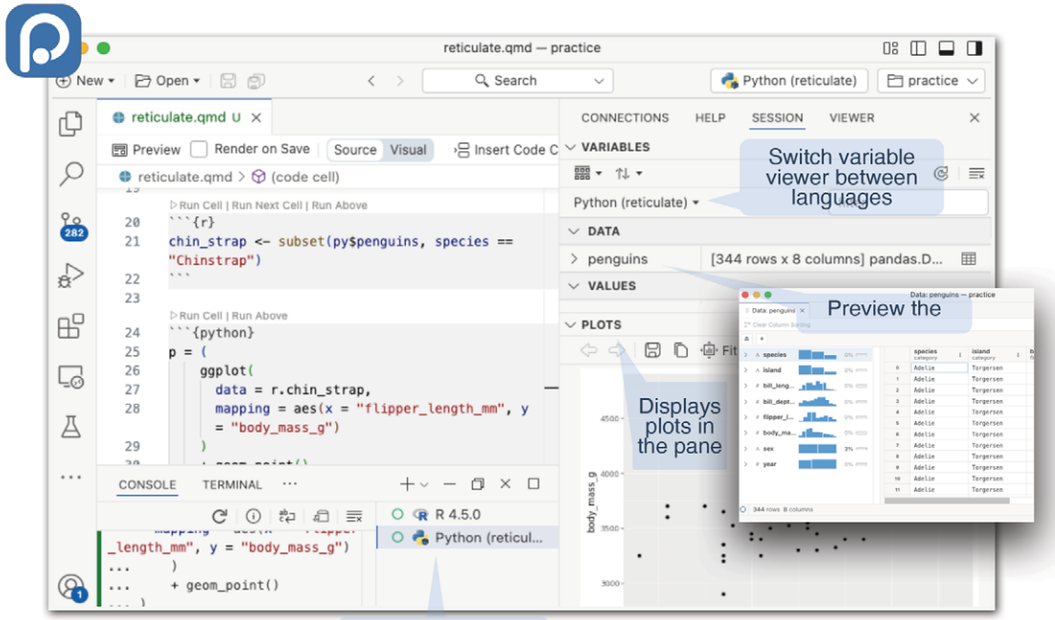
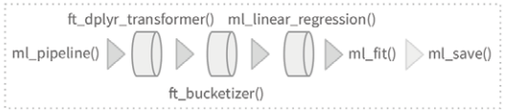
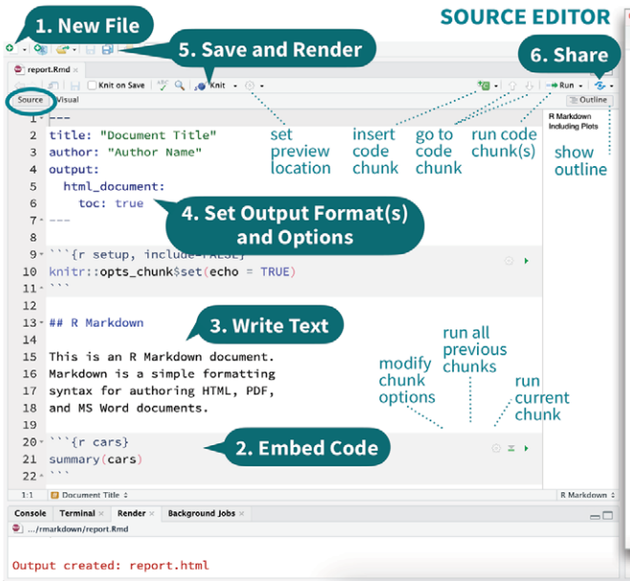
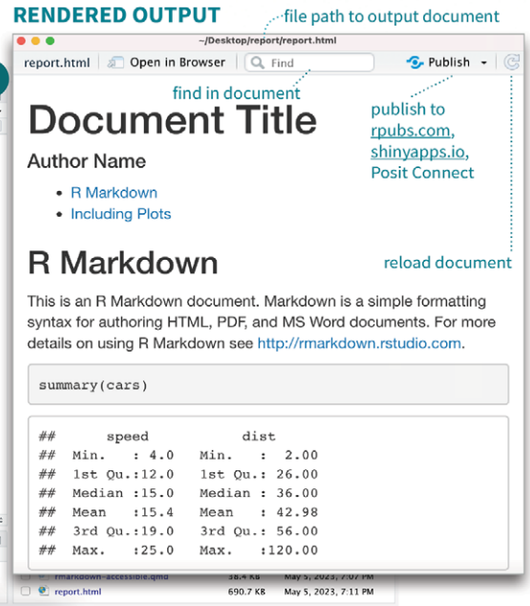
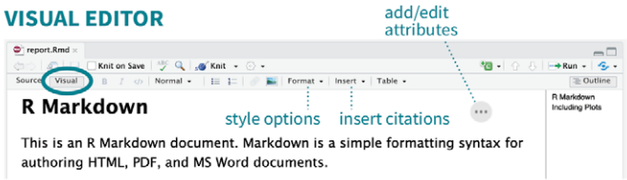
#* @param msg The message to echo
#* @get /echo
function(msg = "") {
  list(
    msg = paste0(
      "The message is: '", msg, "'")
  )
}

#* @plumber
function(pr) {
  pr %>%
    pr_set_api_spec(function(spec) {
      spec$paths[["/echo"]]$get$summary <-
        "Echo back the input"
      spec
    })
}
```

Function that receives and modifies the existing specification

Return the updated specification

By default, Swagger is used to interpret the OpenAPI specification file and generate the user interface for the API. Other interpreters can be used to adjust the look and feel of the user interface via **pr_set_docs()**.

Advanced Plumber

REQUEST and RESPONSE

Plumber provides access to special **req** and **res** objects that can be passed to Plumber functions. These objects provide access to the request submitted by the client and the response that will be sent to the client. Each object has several components, the most helpful of which are outlined below:

Name	Example	Description
req		
req$pr	plumber::pr()	The Plumber router processing the request
req$body	list(a=1)	Typically the same as **argsBody**
req$argsBody	list(a=1)	The parsed body output
req$argsPath	list(c=3)	The values of the path arguments
req$argsQuery	list(e=5)	The parsed output from **req$QUERY_STRING**
req$cookies	list(cook = "a")	A list of cookies
req$REQUEST_METHOD	"GET"	The method used for the HTTP request
req$PATH_INFO	"/"	The path of the incoming HTTP request
req$HTTP_*	"HTTP_USER_AGENT"	All of the HTTP headers sent with the request
req$bodyRaw	charToRaw("a=1")	The **raw()** contents of the request body
res		
res$headers	list(header = "abc")	HTTP headers to include in the response
res$setHeader()	setHeader("foo", "bar")	Sets an HTTP header
res$setCookie()	setCookie("foo", "bar")	Sets an HTTP cookie on the client
res$removeCookie	removeCookie("foo")	Removes an HTTP cookie
res$body	"{\"a\":[1]}"	Serialized output
res$status	200	The response HTTP status code
res$toResponse()	toResponse()	A list of status, headers, and body

ASYNC PLUMBER

Plumber supports asynchronous execution via the **future** R package. This pattern allows Plumber to concurrently process multiple requests.

```
library(plumber)
future::plan("multisession")

#* @get /slow
function() {
  promises::future_promise({
    slow_calc()
  })
}
```

Set the execution plan

Slow calculation

MOUNTING ROUTERS

Plumber routers can be combined by mounting routers into other routers. This can be beneficial when building routers that involve several different endpoints and you want to break each component out into a separate router. These separate routers can even be separate files loaded using **plumb()**.

```
library(plumber)

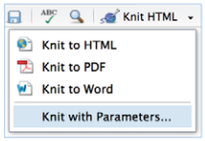
route <- pr() %>%
  pr_get("/foo", function() "foo")

#* @plumber
function(pr) {
  pr %>%
    pr_mount("/bar", route)
}
```

Create an initial router

Mount one router into another

In the above example, the final route is **/bar/foo**.

RUNNING EXAMPLES

Some packages, like the Plumber package itself, may include example Plumber APIs. Available APIs can be viewed using **available_apis()**. These example APIs can be run with **plumb_api()** combined with **pr_run()**.

```
library(plumber)

plumb_api(package = "plumber",
          name = "01-append",
          edit = TRUE) %>%
  pr_run()
```

Identify the package name and API name

Optionally open the file for editing

Run the example API

Deploying Plumber APIs

Once Plumber APIs have been developed, they often need to be deployed somewhere to be useful. Plumber APIs can be deployed in a variety of different ways. One of the easiest way to deploy Plumber APIs is using Posit Connect, which supports push button publishing from the RStudio IDE.



Using Python in R with *reticulate* : : CHEAT SHEET

Getting Started

Reticulate makes it very easy to get started with Python in R. Simply use py_require() to let Reticulate know which Python version and packages you will need:

```
library(reticulate)
py_require("polars", python_version="3.12")
pl <- import("polars")
```

An isolated Python virtual environment that you will not need to manage is created, this eliminates the risk of the environment becoming unstable overtime.

Reticulate uses an extremely fast Python package manager called **uv**. Some features of this system are::

- 10x–100x faster than pip at setting up virtual environments
- Downloads, installs and manages Python versions
- Supports macOS, Linux and Windows

Reticulate will automatically download and install **uv** for you in a location that does not make system changes to your machine.

Manage environments

Manage Virtualenv and Conda environments. Use if py_require() is not an option, or if using an existing Python virtual environment.

Create new	virtualenv_create() / conda_create() *virtualenv_create("my-env")*
Use in R session	use_virtualenv() / use_condaenv() / use_python() *use_virtualenv("my-env")*
List available	virtualenv_list() / conda_list()
Install packages	py_install() / virtualenv_install() / conda_install() *py_install("polars", "my-env")*
Delete from disk	virtualenv_remove() / conda_remove() *virtualenv_remove("my-env")*

Calling Python

IMPORT PYTHON MODULES

Import any Python module into R, and access the attributes of a module with **$**.

import(module, as = NULL, convert = TRUE, delay_load = FALSE**)** - Import a Python module. If convert = TRUE, Python objects are converted to their equivalent R types. Access the attributes of a module with $.

import_from_path(module, path = "." **)** - Import module from an arbitrary filesystem path.

import_main(convert = TRUE**)** - Import the main module, where Python executes code by default.

import_builtins(convert = TRUE**)** - Import Python's built-in functions.

SOURCE PYTHON FILES

Source a Python script and make the Python functions and objects it creates available in R

source_python(file, envir = parent.frame(), convert = TRUE**)** - Run a Python script, assigning objects to a specified R environment.
source_python("file.py")

RUN PYTHON CODE

Execute Python code into the **main** Python module. Access the results, and anything else in Python's **main** module, with **py$**.

py_run_file(file, local = FALSE, convert = TRUE**)** - Run Python file in the main module.
py_run_file("my-script.py")

py_eval(code, convert = TRUE**)** - Run a Python expression, return the result.
py_eval("1 + 1")

py_run_string(code, local = FALSE, convert = TRUE**)** - Run Python code (passed as a string) in the main module.
py_run_string("x = 10"); py$x

IN A NOTEBOOK

Call Python as a code chunk in **Quarto** and **R Markdown**

Begin Python chunks with **```{python}**
(Chunk options like echo, include, etc. all work as expected)

Use the **py** object to access objects created in Python chunks from R chunks.

Python chunks all execute within a **single** Python session so you have access to all objects created, and modules loaded, in previous chunks.



With **py_require()**, define which Python libraries will be used in the notebook

Use the **r** object to access objects created in R chunks from Python chunks

Output displays below chunk, including plots.

AS A PYTHON CONSOLE (REPL)

A REPL (Read, Eval, Print Loop) is a command line where you can run Python code and view the results.

repl_python(module = NULL, quiet = getOption("reticulate.repl.quiet", default = FALSE), input = NULL**)**

1. Use **py_require()** to define libraries and Python version to use
2. Open in the console with **repl_python()**, or by running code in a Python script with **Cmd + Enter** (Windows: **Ctrl + Enter**). Click on the language logo to toggle between R and Python.
3. Type commands at the **>>>** prompt.
4. Press **Enter** to run code.
5. Type **exit** to close and return to R console.



Using Python in R with *reticulate* : : CHEAT SHEET

Object Conversion

AUTOMATIC CONVERSIONS

Reticulate provides **automatic** built-in conversion between Python and R for many Python types.

py_to_r(x**)** Convert a Python object to an R object. Also **r_to_py()**.

R	Python
Single-element vector	Scalar
Multi-element vector	List
List of multiple types	Tuple
Named list	Dict
Matrix/Array	NumPy ndarray
Data Frame	Pandas DataFrame
Function	Python function
NULL, TRUE, FALSE	None, True, False

MANUAL CONVERSIONS

Specify how the objects will be converted

tuple(..., convert = FALSE**)** - Create a Python tuple. *tuple("a", "b", "c")*

dict(..., convert = FALSE**)** - Create a Python dictionary.
dict(foo = "bar", index = 42L)

py_dict() - A dictionary that uses Python objects as keys.
py_dict("foo", "bar")

np_array(data, dtype = NULL, order = "C"**)** - Create NumPy arrays.
np_array(c(1:8), dtype = "float16")

array_reshape(x, dim, order = c("C", "F")**)** - Reshape a Python array.
x <- 1:4; array_reshape(x, c(2, 2))

py_func(f**)** - Wrap an R function in a Python function
with the same signature.
py_func(xor)

iterate(it, f = base::identity, simplify = TRUE**)** - Apply an
R function to each value of a Python iterator or return the values as an R vector, draining the iterator as you
go. Also **iter_next()** and **as_iterator()**.

py_iterator(fn, completed = NULL**)** - Create a Python iterator from an R function.
seq_gen <- function(x){*n <- x; function() {n <<- n + 1; n}}; py_iterator(seq_gen(9))*

Python in the IDE

The RStudio and Positron IDE's provide first-class integration with Reticulate.





Helpers

py_capture_output(expr, type = c("stdout", "stderr"))** Capture and return Python output. Also **py_suppress_warnings()**.

py_get_attr(x, name, silent = FALSE**)** Get an attribute of a Python object. Also **py_set_attr()**, **py_has_attr()**, and **py_list_attributes()**.

py_help(object**)** Open the documentation page for a
Python object.
py_help(sns)

py_last_error() Get the last Python error encountered. Also **py_clear_last_error()** to clear the last error.

py_save_object(object, filename, pickle = "pickle", ...**)** Save and load Python objects with pickle. Also **py_load_object()**.
py_save_object(x, "x.pickle")

with(data, expr, as = NULL, ...**)** Evaluate an expression within a Python context manager.
py <- import_builtins();
with(py$open("output.txt", "w") %as% file,
{ file$write("Hello, there!")})

Choosing Python

Reticulate follows a specific order to discover and choose the Python environment to use

1. **RETICULATE_PYTHON** or **RETICULATE_PYTHON_ENV**
2. **use_python()** or **use_virtualenv()**, if called before import().
3. Working directory contains a virtual env: **./.venv**
4. Environments named after the imported module. e.g. ~/.virtualenvs/r-scipy/ for import("scipy")
5. **RETICULATE_PYTHON_FALLBACK**
6. The default virtualenv: r-reticulate
7. Specifications from **py_require()** *
8. OS' default Python (PATH or Windows registry)

* To have **py_require()** take more precedence, set RETICULATE_PYTHON="managed". It will become number 1 on the list.

This is a partial list of the order of discover, to see the full list *rstudio.github.io/reticulate/articles/versions.html#order-of-discovery*

Data Science in Spark with *sparklyr* : : CHEAT SHEET

Connect

DATABRICKS CONNECT (v2)

1. Open your .Renviron file: `usethis::edit_r_environ()`
2. In the .Renviron file add your Databricks Host Url and Token (PAT):
   - `DATABRICKS_HOST = [Your Host URL]`
   - `DATABRICKS_TOKEN = [Your PAT]`
3. Install extension: `install.packages("pysparklyr")`
4. Open connection:

```
sc <- spark_connect(
cluster_id = "[Your cluster's ID]",
method = "databricks_connect"
)
```

= Supported in Databricks Connect v2

STANDALONE CLUSTER

1. Install RStudio Server on one of the existing nodes or a server in the same LAN
2. Open a connection

```
spark_connect(master="spark://host:port",
    version = "3.2",
    spark_home = [path to Spark])
```

YARN CLIENT

1. Install RStudio Server on an edge node
2. Locate path to the cluster's Spark Home Directory, it normally is "/usr/lib/spark"
3. Basic configuration example

```
conf <- spark_config()
conf$spark.executor.memory <- "300M"
conf$spark.executor.cores <- 2
conf$spark.executor.instances <- 3
conf$spark.dynamicAllocation.enabled<-"false"
```

4. Open a connection

```
sc <- spark_connect(master = "yarn",
          spark_home = "/usr/lib/spark/",
          version = "2.1.0", config = conf)
```

YARN CLUSTER

1. Make sure to have copies of the **yarn-site.xml** and **hive-site.xml** files in the RStudio Server
2. Point environment variables to the correct paths

```
Sys.setenv(JAVA_HOME="[Path]")
Sys.setenv(SPARK_HOME ="[Path]")
Sys.setenv(YARN_CONF_DIR ="[Path]")
```

3. Open a connection

```
sc <- spark_connect(master = "yarn-cluster")
```

KUBERNETES

1. Use the following to obtain the Host and Port

```
system2("kubectl", "cluster-info")
```

2. Open a connection

```
sc <- spark_connect(config =
    spark_config_kubernetes(
     "k8s://https://[HOST]>:[PORT]",
     account = "default",
     image = "docker.io/owner/repo:version"
  ))
```

LOCAL MODE

No cluster required. Use for learning purposes only

1. Install a local version of Spark: `spark_install()`
2. Open a connection

```
sc <- spark_connect(master="local")
```

CLOUD

Azure - `spark_connect(method = "synapse")`

Qubole- `spark_connect(method = "qubole")`

Import

Import data into Spark, not R

READ A FILE INTO SPARK

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

CSV	**spark_read_csv(** header = TRUE, columns=NULL, infer_schema=TRUE, delimiter = ",", quote= "\"", escape = "\\", charset = "UTF-8", null_value = NULL**)**
JSON	**spark_read_json()**
PARQUET	**spark_read_parquet()**
TEXT	**spark_read_text()**
DELTA	**spark_read_delta()**

FROM A TABLE

dplyr::**tbl**(scr, ...) - Creates a reference to the table without loading its data into memory

dbplyr::**in_catalog()** - Enables a three part table address

```
x <- tbl(sc,in_catalog("catalog", "schema", "table"))
```

Import
- From R (**copy_to()**)
- Read a file (**spark_read_**)
- Read Hive table (**tbl()**)

Wrangle
- **dplyr** verb
- **tidyr** commands
- Feature transformer (**ft_**)
- Direct Spark SQL (**DBI**)

Visualize
- Collect result, plot in R

Model
- Spark MLlib (**ml_**)
- H2O Extension

Communicate
Collect results into R share using **Quarto**

R for Data Science, Wickham, Çetinkaya-Rundel, Grolemund

R DATA FRAME INTO SPARK

dplyr::**copy_to(**dest, df, name**)**

Apache Arrow accelerates data transfer between R and Spark. To use, simply load the library

```
library(sparklyr)
library(arrow)
```

Wrangle

DPLYR VERBS

Translates into Spark SQL statements

```
copy_to(sc, mtcars) |>
  mutate(trm = ifelse(am == 0,
        "auto", "man")) |>
  group_by(trm) |>
  summarise_all(mean)
```

TIDYR

pivot_longer() - Collapse several columns into two.

pivot_wider() - Expand two columns into several.

nest() / unnest() - Convert groups of cells into list-columns, and vice versa.

unite() / separate() - Split a single column into several columns, and vice versa.

fill() - Fill NA with the previous value

FEATURE TRANSFORMERS

ft_binarizer() - Assigned values based on threshold

ft_bucketizer() - Numeric column to discretized column

ft_count_vectorizer() - Extracts a vocabulary from document

ft_discrete_cosine_transform() - 1D discrete cosine transform of a real vector

ft_elementwise_product() - Element- wise product between 2 cols

ft_hashing_tf() - Maps a sequence of terms to their term frequencies using the hashing trick.

ft_idf() - Compute the Inverse Document Frequency (IDF) given a collection of documents.

ft_imputer() - Imputation estimator for completing missing values, uses the mean or the median of the columns.

ft_index_to_string() - Index labels back to label as strings

ft_interaction() - Takes in Double and Vector columns and outputs a flattened vector of their feature interactions.

ft_max_abs_scaler() - Rescale each feature individually to range [-1, 1]

ft_min_max_scaler() - Rescale each feature to a common range [min, max] linearly

ft_ngram() - Converts the input array of strings into an array of n-grams

ft_bucketed_random_projection_lsh()
ft_minhash_lsh() - Locality Sensitive Hashing functions for Euclidean distance and Jaccard distance (MinHash)

Data Science in Spark with *sparklyr* : : CHEAT SHEET

ft_normalizer() - Normalize a vector to have unit norm using the given p-norm

ft_one_hot_encoder()- Continuous to binary vectors

ft_pca() - Project vectors to a lower dimensional space of top k principal components.

ft_quantile_discretizer() - Continuous to binned categorical values.

ft_regex_tokenizer() - Extracts tokens either by using the provided regex pattern to split the text.

ft_robust_scaler() - Removes the median and scales according to standard scale.

ft_standard_scaler() - Removes the mean and scaling to unit variance using column summary statistics

ft_stop_words_remover() - Filters out stop words from input

ft_string_indexer() - Column of labels into a column of label indices.

ft_tokenizer() - Converts to lowercase and then splits it by white spaces

ft_vector_assembler() - Combine vectors into single row-vector

ft_vector_indexer() - Indexing categorical feature columns in a dataset of Vector

ft_vector_slicer() - Takes a feature vector and outputs a new feature vector with a subarray of the original features

ft_word2vec() - Word2Vec transforms a word into a code

Visualize

DPLYR + GGPLOT2

```
copy_to(sc, mtcars) |>
  group_by(cyl) |>
  summarise(mpg_m = mean(mpg)) |>
  collect() |>
  ggplot() +
  geom_col(aes(cyl, mpg_m))
```

Summarize in Spark

Collect results in R

Create plot

Modeling

REGRESSION

ml_linear_regression() - Linear regression.

ml_aft_survival_regression() - Parametric survival regression model named accelerated failure time (AFT) model

ml_generalized_linear_regression() - GLM

ml_isotonic_regression() - Uses parallelized pool adjacent violators algorithm.

ml_random_forest_regressor() - Regression using random forests.

CLASSIFICATION

ml_linear_svc() - Classification using linear support vector machines

ml_logistic_regression() - Logistic regression

ml_multilayer_perceptron_classifier() - Based on the Multilayer Perceptron.

ml_naive_bayes() - It supports Multinomial NB which can handle finitely supported discrete data

ml_one_vs_rest() - Reduction of Multiclass, performs reduction using one against all strategy.

TREE

ml_decision_tree_classifier()|**ml_decision_tree()**|**ml_decision_tree_regressor()** - Classification and regression using decision trees

ml_gbt_classifier()|**ml_gradient_boosted_trees()** | **ml_gbt_regressor()** - Binary classification and regression using gradient boosted trees

ml_random_forest_classifier() - Classification and regression using random forests.

ml_feature_importances() | **ml_tree_feature_importance()** - Feature Importance for Tree Models

CLUSTERING

ml_bisecting_kmeans() - A bisecting k-means algorithm based on the paper

ml_lda() | **ml_describe_topics()** | **ml_log_likelihood()** | **ml_log_perplexity()** | **ml_topics_matrix()** - LDA topic model designed for text documents.

ml_gaussian_mixture() - Expectation maximization for multivariate Gaussian Mixture Models (GMMs)

ml_kmeans() | **ml_compute_cost()** |**ml_compute_silhouette_measure()** - Clustering with support for k-means

ml_power_iteration() - For clustering vertices of a graph given pairwise similarities as edge properties.

RECOMMENDATION

ml_als() | **ml_recommend()** - Recommendation using Alternating Least Squares matrix factorization

EVALUATION

ml_clustering_evaluator() - Evaluator for clustering

ml_evaluate() - Compute performance metrics

ml_binary_classification_evaluator() | **ml_binary_classification_eval()** | **ml_classification_eval()** - A set of functions to calculate performance metrics for prediction models.

FREQUENT PATTERN

ml_fpgrowth() | **ml_association_rules()** | **ml_freq_itemsets()** - A parallel FP-growth algorithm to mine frequent itemsets.

ml_freq_seq_patterns() | **ml_prefixspan()** - PrefixSpan algorithm for mining frequent itemsets.

STATS

ml_summary() - Extracts a metric from the summary object of a Spark ML model

ml_corr() - Compute correlation matrix

FEATURE

ml_chisquare_test(x,features,label) - Pearson's independence test for every feature against the label

ml_default_stop_words() - Loads the default stop words for the given language

UTILITIES

ml_call_constructor() - Identifies the associated sparklyr ML constructor for the JVM

ml_model_data() - Extracts data associated with a Spark ML model

ml_standardize_formula() - Generates a formula string from user inputs

ml_uid() - Extracts the UID of an ML object.

ML Pipelines

*Easily create a formal Spark Pipeline models using R. Save the Pipeline in native Sacala. It will have **no dependencies** on R.*

INITIALIZE AND TRAIN

ml_pipeline() - Initializes a new Spark Pipeline

ml_fit() - Trains the model, outputs a Spark Pipeline Model.

SAVE AND RETRIEVE

ml_save() - Saves into a format that can be read by Scala and PySpark .

ml_read() - Reads Spark object into sparklyr.



spark.posit.co/guides/pipelines

Distributed R

Run arbitrary R code at scale inside your cluster with **spark_apply()**. Useful when there you need functionality only available in R, and to solve 'embarrassingly parallel problems'

spark_apply(x, f, columns = NULL, memory = TRUE, group_by = NULL, name = NULL, barrier = NULL, fetch_result_as_sdf = TRUE)

```
copy_to(sc, mtcars) |>
  spark_apply(
    nrow, # R only function
    group_by = "am",
    columns = "am double, x long"
    )
```

More Info

spark.posit.co

therinspark.com

posit

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at **spark.posit.co** • sparklyr 1.9.1 • Updated: 2025-07

rmarkdown : : CHEATSHEET

What is rmarkdown?

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

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

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

Workflow

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







Embed Code with knitr

CODE CHUNKS

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

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

SET GLOBAL OPTIONS

Set options for the entire document in the first chunk.

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

INLINE CODE

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

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

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

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

Insert Citations

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

BUILD YOUR BIBLIOGRAPHY

- Add BibTeX or CSL bibliographies to the YAML header.

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

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

INSERT CITATIONS

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

Insert Tables

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

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

Table with kable

eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62

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

Write with Markdown

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

Plain text. End a line with two spaces to start a new paragraph. Also end with a backslash\ to make a new line.	Plain text. End a line with two spaces to start a new paragraph. Also end with a backslash to make a new line.
italics and **bold** superscript^2^/subscript~2~ ~~strikethrough~~ escaped: * _ \\ endash: --, emdash: ---	*italics* and **bold** superscript^2/subscript$_2$ ~~strikethrough~~ escaped: * _ \ endash: –, emdash: —
# Header 1 ## Header 2 ... ###### Header 6	**Header 1** **Header 2** ... **Header 6**

- unordered list
- item 2
  - item 2a (indent 1 tab)
  - item 2b
1. ordered list
2. item 2
  - item 2a (indent 1 tab)
  - item 2b
	• unordered list
	• item 2
	  • item 2a (indent 1 tab)
	  • item 2b
	1. ordered list
	2. item 2
	  • item 2a (indent 1 tab)
	  • item 2b

<link url> [This is a link.](link url) [This is another link][id]. At the end of the document: [id]: link url ![Caption](image.png) or ![Caption][id2] At the end of the document: [id2]: image.png	http://www.posit.co/ This is a link. This is another link. Caption.

`verbatim code`
``
multiple lines of verbatim code
``
	`verbatim code`
	``
	multiple lines of verbatim code
	``

> block quotes	block quotes
equation: $e^{i \pi} + 1 = 0$ equation block: $$E = mc^{2}$$	equation: $e^{i\pi} + 1 = 0$ equation block: $$E = mc^2$$
horizontal rule: ---	horizontal rule:

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

HTML Tabsets

```
## Results {.tabset}
### Plots
text

### Tables
more text
```

Results

Plots | Tables

text

Set Output Formats and their Options in YAML

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

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

Indent format 2 characters, indent options 4 characters

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

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

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

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

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

Render

When you render a document, rmarkdown:

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

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

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

Share

Publish on Posit Connect to share R Markdown documents securely, schedule automatic updates, and interact with parameters in real-time. **posit.co/products/enterprise/connect**.

More Header Options

PARAMETERS

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

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

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

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

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

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



REUSABLE TEMPLATES

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

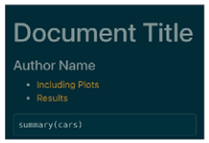
```
---
name: "My Template"
---
```

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

BOOTSWATCH THEMES

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

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




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

CUSTOM THEMES

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

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

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

STYLING WITH CSS AND SCSS

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

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

Apply CSS styling by writing HTML tags directly or:

- Use markdown to apply style attributes inline.

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

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

A green word.

All of these words are green.

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

.my-css-tag ...

This is a div with some text in it.

INTERACTIVITY

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

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

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

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

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

How many cars? 5

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

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

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

from posit

posit.co