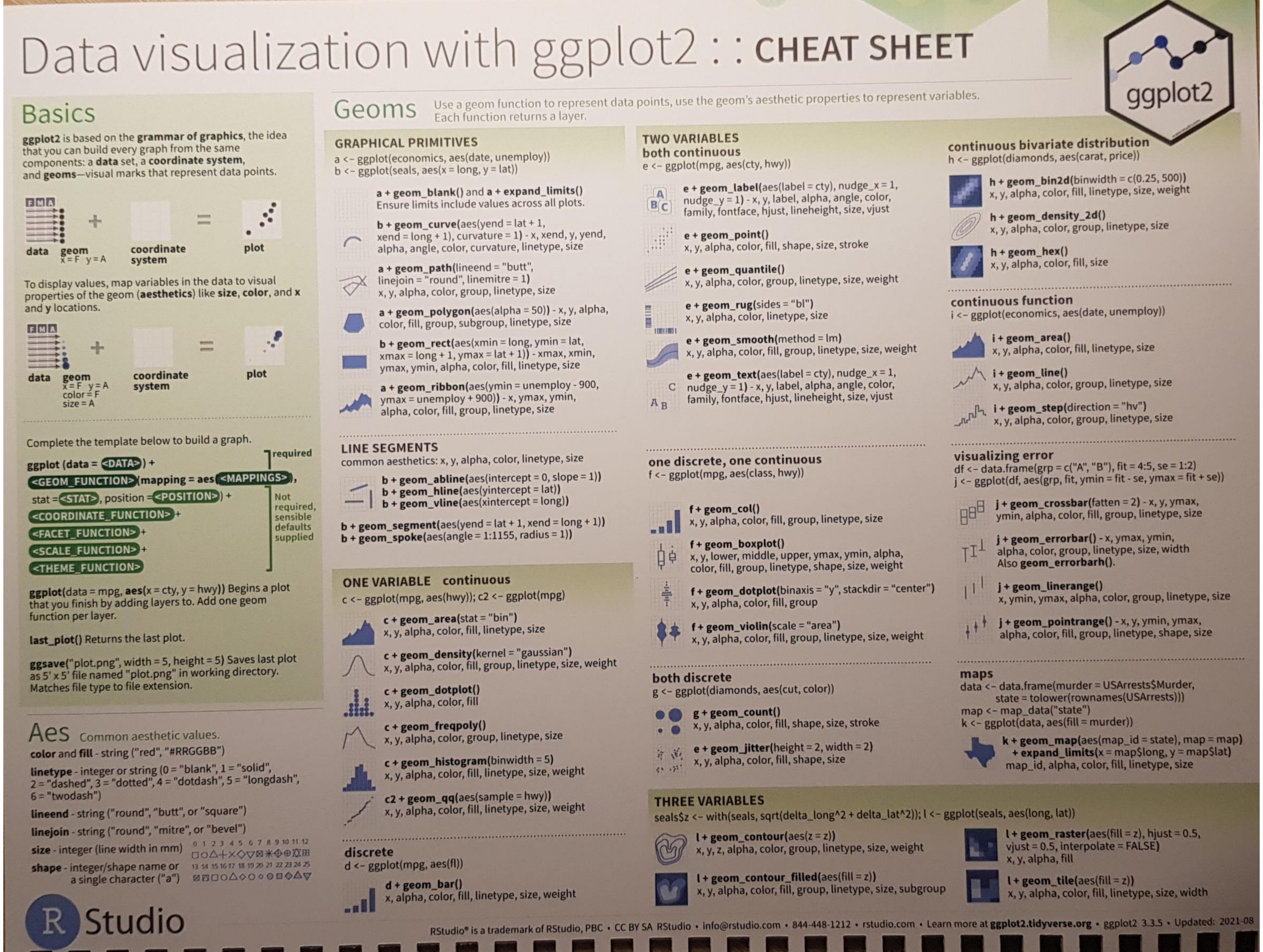
rstudio::conf

2022 / CHEATSHEETS

FROM RStudio



Data visualization with ggplot2 : : CHEAT SHEET

ggplot2

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.

F M A

data + geom x = F y = A + coordinate system = plot

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

F M A

data + geom x = F y = A color = F size = A + coordinate system = plot

Complete the template below to build a graph.

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

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

last_plot() Returns the last plot.

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

Aes Common aesthetic values.

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

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

lineend - string ("round", "butt", or "square")

linejoin - string ("round", "mitre", or "bevel")

size - integer (line width in mm)

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

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
⊠ ◪ ■ ● ▲ ◆ ● • ● ■ ◆ ▲ ▼

RStudio

Geoms

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

GRAPHICAL PRIMITIVES

a <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))

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

b + geom_curve(aes(yend = lat + 1, xend = long + 1), curvature = 1) - x, xend, y, yend, alpha, angle, color, curvature, linetype, size

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

a + geom_polygon(aes(alpha = 50)) - x, y, alpha, color, fill, group, subgroup, linetype, size

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

a + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)) - x, ymax, ymin, alpha, color, fill, group, linetype, size

LINE SEGMENTS

common aesthetics: x, y, alpha, color, linetype, size

b + geom_abline(aes(intercept = 0, slope = 1))
b + geom_hline(aes(yintercept = lat))
b + geom_vline(aes(xintercept = long))

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

ONE VARIABLE continuous

c <- ggplot(mpg, aes(hwy)); c2 <- ggplot(mpg)

c + geom_area(stat = "bin")
x, y, alpha, color, fill, linetype, size

c + geom_density(kernel = "gaussian")
x, y, alpha, color, fill, group, linetype, size, weight

c + geom_dotplot()
x, y, alpha, color, fill

c + geom_freqpoly()
x, y, alpha, color, group, linetype, size

c + geom_histogram(binwidth = 5)
x, y, alpha, color, fill, linetype, size, weight

c2 + geom_qq(aes(sample = hwy))
x, y, alpha, color, fill, linetype, size, weight

discrete

d <- ggplot(mpg, aes(fl))

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

TWO VARIABLES

both continuous

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

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

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

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

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

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

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

one discrete, one continuous

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

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

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

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

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

both discrete

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

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

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

continuous bivariate distribution

h <- ggplot(diamonds, aes(carat, price))

h + geom_bin2d(binwidth = c(0.25, 500))
x, y, alpha, color, fill, linetype, size, weight

h + geom_density_2d()
x, y, alpha, color, group, linetype, size

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

continuous function

i <- ggplot(economics, aes(date, unemploy))

i + geom_area()
x, y, alpha, color, fill, linetype, size

i + geom_line()
x, y, alpha, color, group, linetype, size

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

visualizing error

df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)
j <- ggplot(df, aes(grp, fit, ymin = fit - se, ymax = fit + se))

j + geom_crossbar(fatten = 2) - x, y, ymax, ymin, alpha, color, fill, group, linetype, size

j + geom_errorbar() - x, ymax, ymin, alpha, color, group, linetype, size, width
Also **geom_errorbarh()**.

j + geom_linerange()
x, ymin, ymax, alpha, color, group, linetype, size

j + geom_pointrange() - x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

maps

data <- data.frame(murder = USArrests$Murder,
state = tolower(rownames(USArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))

k + geom_map(aes(map_id = state), map = map)
+ expand_limits(x = map$long, y = map$lat)
map_id, alpha, color, fill, linetype, size

THREE VARIABLES

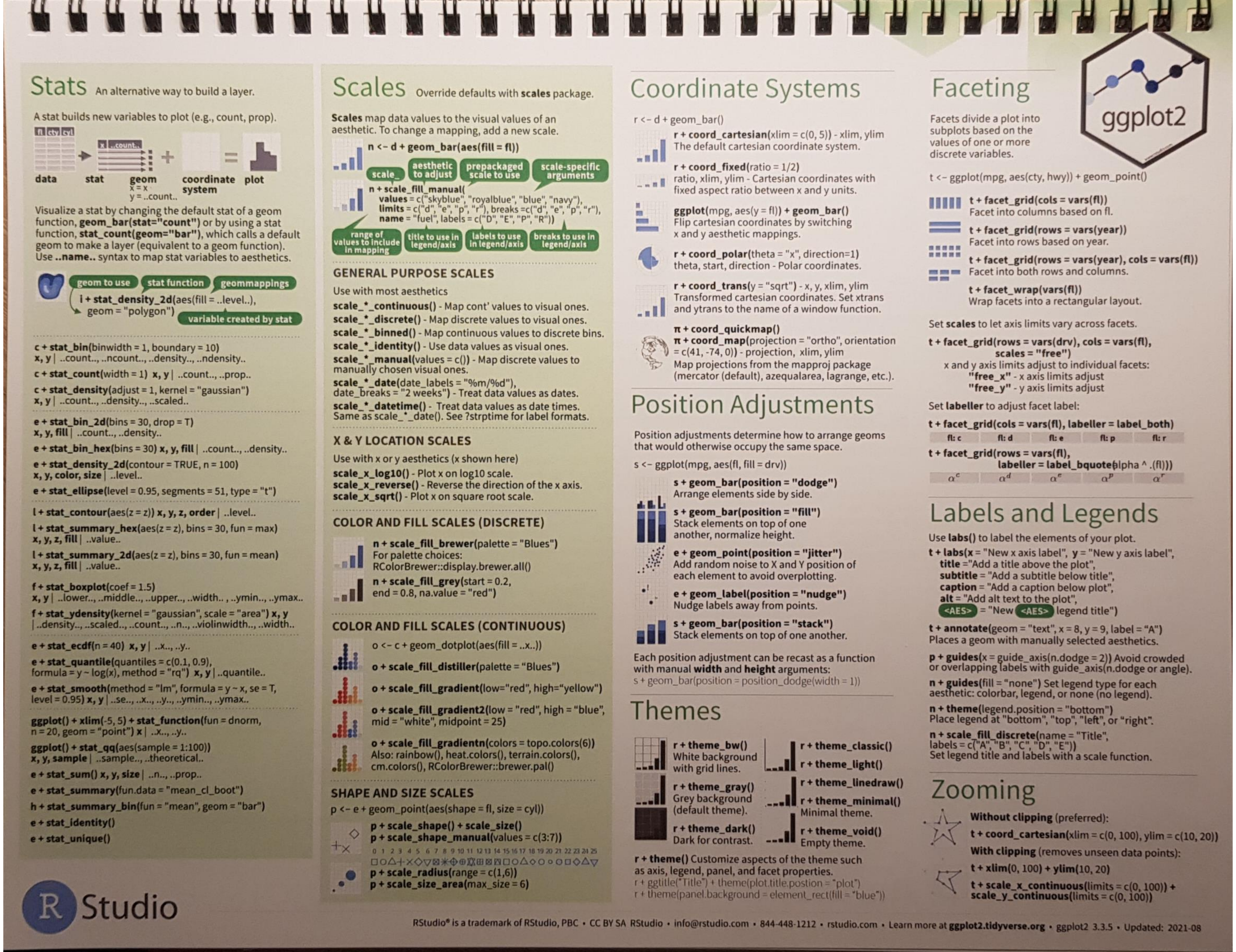
seals$z <- with(seals, sqrt(delta_long^2 + delta_lat^2)); l <- ggplot(seals, aes(long, lat))

l + geom_contour(aes(z = z))
x, y, z, alpha, 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).

fl cty cyl · x ..count..

data → stat → geom + coordinate system = plot

geom x = x y = ..count..

Visualize a stat by changing the default stat of a geom function, **geom_bar(stat="count")** or by using a stat function, **stat_count(geom="bar")**, which calls a default geom to make a layer (equivalent to a geom function). Use **..name..** syntax to map stat variables to aesthetics.

geom to use · stat function · geommappings

```
i + stat_density_2d(aes(fill = ..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()
```

R Studio

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.
ggplot(mpg, aes(y = fl)) + geom_bar()
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_quickmap()
π + coord_map(projection = "ortho", orientation
= c(41, -74, 0)) - projection, xlim, ylim
Map projections from the mapproj package
(mercator (default), azequalarea, lagrange, etc.).
```

Position Adjustments

Position adjustments determine how to arrange geoms that would otherwise occupy the same space.

```
s <- ggplot(mpg, aes(fl, fill = drv))
s + geom_bar(position = "dodge")
Arrange elements side by side.
s + geom_bar(position = "fill")
Stack elements on top of one
another, normalize height.
e + geom_point(position = "jitter")
Add random noise to X and Y position of
each element to avoid overplotting.
e + geom_label(position = "nudge")
Nudge labels away from points.
s + geom_bar(position = "stack")
Stack elements on top of one another.
```

Each position adjustment can be recast as a function with manual **width** and **height** arguments:

```
s + geom_bar(position = position_dodge(width = 1))
```

Themes

```
r + theme_bw()
White background
with grid lines.
r + theme_gray()
Grey background
(default theme).
r + theme_dark()
Dark for contrast.
r + theme_classic()
r + theme_light()
r + theme_linedraw()
r + theme_minimal()
Minimal theme.
r + theme_void()
Empty theme.
```

r + theme() Customize aspects of the theme such as axis, legend, panel, and facet properties.

```
r + ggtitle("Title") + theme(plot.title.postion = "plot")
r + theme(panel.background = element_rect(fill = "blue"))
```

Faceting

ggplot2

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

```
t <- ggplot(mpg, aes(cty, hwy)) + geom_point()
t + facet_grid(cols = vars(fl))
Facet into columns based on fl.
t + facet_grid(rows = vars(year))
Facet into rows based on year.
t + facet_grid(rows = vars(year), cols = vars(fl))
Facet into both rows and columns.
t + facet_wrap(vars(fl))
Wrap facets into a rectangular layout.
```

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

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

Set **labeller** to adjust facet label:

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

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

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

α^c α^d α^e α^p α^r

Labels 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))
```

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **ggplot2.tidyverse.org** • ggplot2 3.3.5 • Updated: 2021-08

Data transformation with dplyr : : CHEAT SHEET

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

Each **variable** is in its own **column** & Each **observation**, or **case**, is in its own **row**

pipes

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

Summarise Cases

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

summary function

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

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

Group Cases

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

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

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

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

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

Manipulate Cases

EXTRACT CASES

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

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

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

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

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

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

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

Logical and boolean operators to use with filter()

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

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

ARRANGE CASES

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

ADD CASES

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

Manipulate Variables

EXTRACT VARIABLES

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

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

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

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

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

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

MANIPULATE MULTIPLE VARIABLES AT ONCE

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

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

MAKE NEW VARIABLES

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

vectorized function

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

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

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

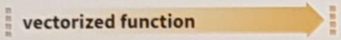
RStudio

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **dplyr.tidyverse.org** • dplyr 1.0.7 • Updated: 2021-07

Vectorized Functions

TO USE WITH MUTATE ()

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



OFFSET

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

CUMULATIVE AGGREGATE

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

RANKING

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

MATH

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

MISCELLANEOUS

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

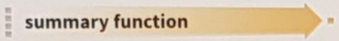
```
starwars %>%
  mutate(type = case_when(
    height > 200 | mass > 200 ~ "large",
      species == "Droid"        ~ "robot",
      TRUE                      ~ "other")
      )
```

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

Summary Functions

TO USE WITH SUMMARISE ()

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



COUNT

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

POSITION

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

LOGICAL

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

ORDER

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

RANK

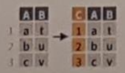
quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

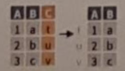
IQR() - Inter-Quartile Range
mad() - median absolute deviation
sd() - standard deviation
var() - variance

Row Names

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



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

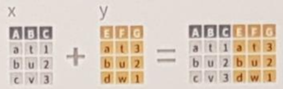


tibble::**column_to_rownames()**
Move col into row names.
column_to_rownames(a, var = "C")

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

Combine Tables

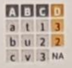
COMBINE VARIABLES



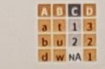
bind_cols(…, .name_repair) Returns tables placed side by side as a single table. Column lengths must be equal. Columns will NOT be matched by id (to do that look at Relational Data below), so be sure to check that both tables are ordered the way you want before binding.

RELATIONAL DATA

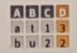
Use a "**Mutating Join**" to join one table to columns from another, matching values with the rows that they correspond to. Each join retains a different combination of values from the tables.



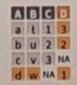
left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), …, keep = FALSE, na_matches = "na") Join matching values from y to x.



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

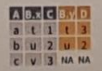


inner_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), …, keep = FALSE, na_matches = "na") Join data. Retain only rows with matches.

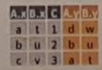


full_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), …, keep = FALSE, na_matches = "na") Join data. Retain all values, all rows.

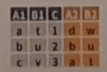
COLUMN MATCHING FOR JOINS



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

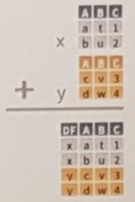


Use a named vector, **by = c("col1" = "col2")**, to match on columns that have different names in each table.
left_join(x, y, by = c("C" = "D"))



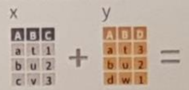
Use **suffix** to specify the suffix to give to unmatched columns that have the same name in both tables.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

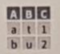
COMBINE CASES



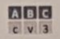
bind_rows(…, .id = NULL) Returns tables one on top of the other as a single table. Set .id to a column name to add a column of the original table names (as pictured).

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



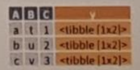


semi_join(x, y, by = NULL, copy = FALSE, …, na_matches = "na") Return rows of x that have a match in y. Use to see what will be included in a join.



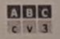
anti_join(x, y, by = NULL, copy = FALSE, …, na_matches = "na") Return rows of x that do not have a match in y. Use to see what will not be included in a join.

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

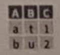


nest_join(x, y, by = NULL, copy = FALSE, keep = FALSE, name = NULL, …) Join data, nesting matches from y in a single new data frame column.

SET OPERATIONS



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



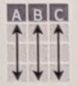
setdiff(x, y, …)
Rows that appear in x but not y.

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

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

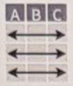
Data tidying with tidyr : : CHEAT SHEET

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



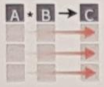
Each **variable** is in its own **column**

&



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

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



Preserve **cases** in vectorized operations

Tibbles

AN ENHANCED DATA FRAME

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

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

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

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

CONSTRUCT A TIBBLE

tibble(…) Construct by columns.

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

tribble(…) Construct by rows.

```
tribble(~x,  ~y,
         1, "a",
         2, "b",
         3, "c")
```

Both make this tibble

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

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

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

is_tibble(x) Test whether x is a tibble.

Reshape Data

- Pivot data to reorganize values into a new layout.

table4a

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

→

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

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

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

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

table2

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

→

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

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

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

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

Split Cells

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

table5

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

→

country	year
A	1999
A	2000
B	1999
B	2000

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

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

table3

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

→

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

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

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

table3

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

→

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

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

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

Expand Tables

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

x

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

→

x1	x2
A	1
A	2
B	1
B	2

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

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

x

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

→

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

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

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

Handle Missing Values

Drop or replace explicit missing values (NA).

x

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

→

x1	x2
A	1
D	3

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

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

x

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

→

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

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

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

x

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

→

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

replace_na(data, replace) Specify a value to replace NA in selected columns.

```
replace_na(x, list(x2 = 2))
```

R Studio

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **tidyr.tidyverse.org** • tibble 3.1.2 • tidyr 1.1.3 • Updated: 2021-08

Nested Data

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

Use a nested data frame to:

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

CREATE NESTED DATA

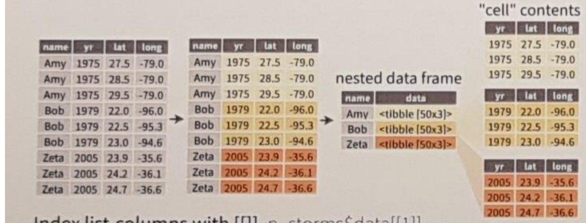
nest(data, …) Moves groups of cells into a list-column of a data frame. Use alone or with dplyr::**group_by()**:

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

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

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

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



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

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

nested data frame

name	data
Amy	<tibble [50x3]>
Bob	<tibble [50x3]>
Zeta	<tibble [50x3]>

"cell" contents

yr	lat	long
1975	27.5	-79.0
1975	28.5	-79.0
1975	29.5	-79.0

yr	lat	long
1979	22.0	-96.0
1979	22.5	-95.3
1979	23.0	-94.6

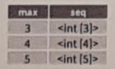
yr	lat	long
2005	23.9	-35.6
2005	24.2	-36.1
2005	24.7	-36.6

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

CREATE TIBBLES WITH LIST-COLUMNS

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

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



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

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

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

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

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

OUTPUT LIST-COLUMNS FROM OTHER FUNCTIONS

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

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

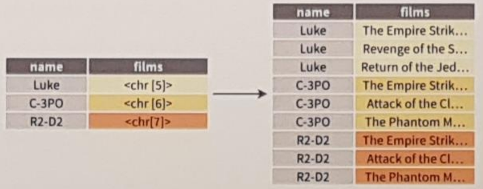
RESHAPE NESTED DATA

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

```
n_storms %>% unnest(data)
```

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

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

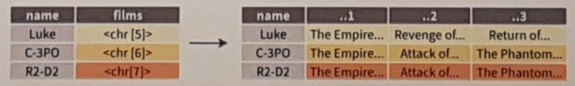


name	films
Luke	<chr [5]>
C-3PO	<chr [6]>
R2-D2	<chr[7]>

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

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

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

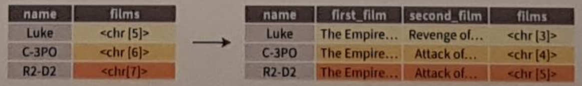


name	films
Luke	<chr [5]>
C-3PO	<chr [6]>
R2-D2	<chr[7]>

name	..1	..2	..3
Luke	The Empire...	Revenge of...	Return of...
C-3PO	The Empire...	Attack of...	The Phantom...
R2-D2	The Empire...	Attack of...	The Phantom...

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)
```



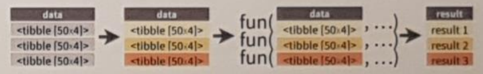
name	films
Luke	<chr [5]>
C-3PO	<chr [6]>
R2-D2	<chr[7]>

name	first_film	second_film	films
Luke	The Empire…	Revenge of…	<chr [3]>
C-3PO	The Empire…	Attack of…	<chr [4]>
R2-D2	The Empire…	Attack of…	<chr [5]>

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.**



data
<tibble [50x4]>
<tibble [50x4]>
<tibble [50x4]>

data
<tibble [50x4]>
<tibble [50x4]>
<tibble [50x4]>

result
result 1
result 2
result 3

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.

RStudio

Data import with the tidyverse : : **CHEAT SHEET**

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(),
#   sex = col_character(),
#   earn = col_double()
# )
```

age is an integer

sex is a character

earn is a double (numeric)

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)

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()** - "?"

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

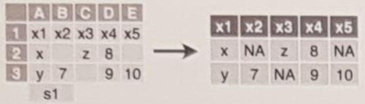
```
# col types: skip, guess, integer, logical, character
read_csv(
  file,
  col_type = "_?ilc"
)
```

RStudio

Import Spreadsheets

with readxl

READ EXCEL FILES

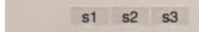


read_excel(path, sheet = NULL, range = NULL)
Read a .xls or .xlsx file based on the file extension. See front page for more read arguments. Also **read_xls()** and **read_xlsx()**.
read_excel("excel_file.xlsx")

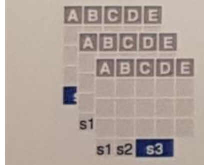
READ SHEETS

A B C D E
s1 s2 s3

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_dfr() to read multiple files into one data frame.

```
path <- "your_file_path.xlsx"
path %>% excel_sheets() %>%
  set_names() %>%
  map_dfr(read_excel, path = path)
```

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

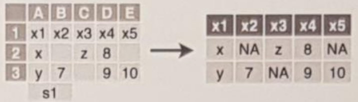
logical	numeric	text	date	list
TRUE	2	hello	1947-01-08	hello
FALSE	3.45	world	1956-10-21	1

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

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

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

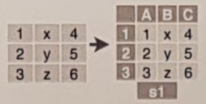
URLs are in the form:
https://docs.google.com/spreadsheets/d/
SPREADSHEET_ID/edit#gid=**SHEET_ID**

gs4_get(ss) Get spreadsheet meta data.

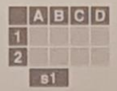
gs4_find(...) Get data on all spreadsheet files.

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

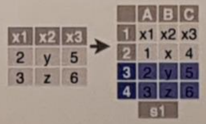
WRITE SHEETS



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.

OTHER USEFUL EXCEL PACKAGES

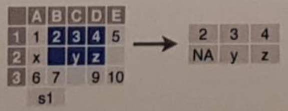
For functions to write data to Excel files, see:

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

For working with non-tabular Excel data, see:

- **tidyxl**

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()**.

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**.

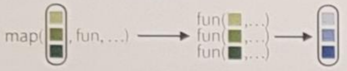
Apply functions with purrr : : CHEAT SHEET

Map Functions

ONE LIST

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

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



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

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

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

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

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

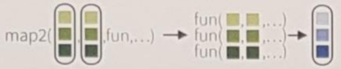
map_dfr(.x, .f, ..., .id = NULL)
Return a data frame created by row-binding.
```
map_dfr(x, summary)
```

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

TWO LISTS

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

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



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

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

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

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

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

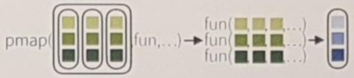
map2_dfr(.x, .y, .f, ..., .id = NULL) Return a data frame created by row-binding.
```
map2_dfr(l1, l2,
~ as.data.frame(c(.x, .y)))
```

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

MANY LISTS

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

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



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

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

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

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

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

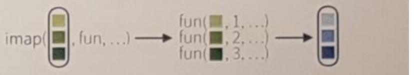
pmap_dfr(.l, .f, ..., .id = NULL) Return a data frame created by row-binding.
```
pmap_dfr(list(l1, l2),
~ as.data.frame(c(.x, .y)))
```

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

LISTS AND INDEXES

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

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



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

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

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

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

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

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

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

Function Shortcuts

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

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

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

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

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

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

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

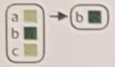
```
imap(list(a, b, c), ~ paste0(.y, ": ", .x)
```
outputs "index: value" for each item

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

RStudio

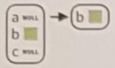
Work with Lists

Filter



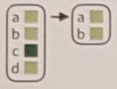
keep(.x, .p, …)
Select elements that pass a logical test.
Conversely, **discard**().

```
keep(x, is.na)
```



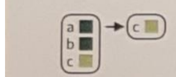
compact(.x, .p = identity)
Drop empty elements.

```
compact(x)
```



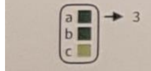
head_while(.x, .p, …)
Return head elements until one does not pass.
Also **tail_while**().

```
head_while(x, is.character)
```



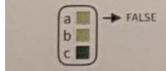
detect(.x, .f, ..., dir = c("forward", "backward"), .right = NULL, .default = NULL)
Find first element to pass.

```
detect(x,  is.character)
```



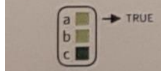
detect_index(.x, .f, ..., dir = c("forward", "backward"), .right = NULL) Find index of first element to pass.

```
detect_index(x, is.character)
```



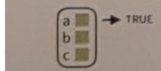
every(.x, .p, …)
Do all elements pass a test?

```
every(x, is.character)
```



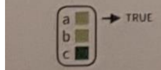
some(.x, .p, …)
Do some elements pass a test?

```
some(x,  is.character)
```



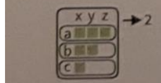
none(.x, .p, …)
Do no elements pass a test?

```
none(x, is.character)
```



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

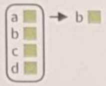
```
has_element(x, "foo")
```



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

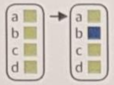
```
vec_depth(x)
```

Index



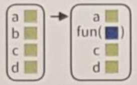
pluck(.x, ..., .default=NULL)
Select an element by name or index. Also **attr_getter()** and **chuck()**.

```
pluck(x, "b")
x %>% pluck("b")
```



assign_in(x, where, value)
Assign a value to a location using pluck selection.

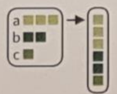
```
assign_in(x, "b", 5)
x %>% assign_in("b", 5)
```



modify_in(.x, .where, .f)
Apply a function to a value at a selected location.

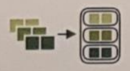
```
modify_in(x, "b", abs)
x %>% modify_in("b", abs)
```

Reshape



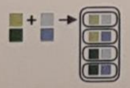
flatten(.x) Remove a level of indexes from a list.
Also **flatten_chr()** etc.

```
flatten(x)
```



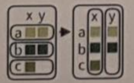
array_tree(array, margin = NULL) Turn array into list.
Also **array_branch()**.

```
array_tree(x, margin = 3)
```



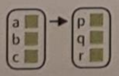
cross2(.x, .y, .filter = NULL)
All combinations of .x and .y.
Also **cross()**, **cross3()**, and **cross_df()**.

```
cross2(1:3, 4:6)
```



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

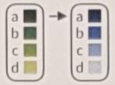
```
transpose(x)
```



set_names(x, nm = x)
Set the names of a vector/list directly or with a function.

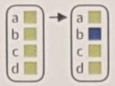
```
set_names(x, c("p", "q", "r"))
set_names(x, tolower)
```

Modify



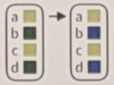
modify(.x, .f, ...) Apply a function to each element. Also **modify2()**, and **imodify()**.

```
modify(x, ~.+ 2)
```



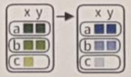
modify_at(.x, .at, .f, ...) Apply a function to selected elements.
Also **map_at()**.

```
modify_at(x, "b", ~.+ 2)
```



modify_if(.x, .p, .f, ...) Apply a function to elements that pass a test. Also **map_if()**.

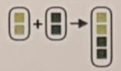
```
modify_if(x, is.numeric,~.+2)
```



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

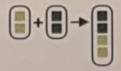
```
modify_depth(x, 2, ~.+ 2)
```

Combine



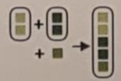
append(x, values, after = length(x)) Add values to end of list.

```
append(x, list(d = 1))
```



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

```
prepend(x, list(d = 1))
```



splice(…) Combine objects into a list, storing S3 objects as sub-lists.

```
splice(x, y, "foo")
```

List-Columns

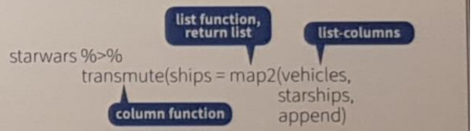
max	seq
3	<int [3]>
4	<int [4]>
5	<int [5]>

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

WORK WITH LIST-COLUMNS

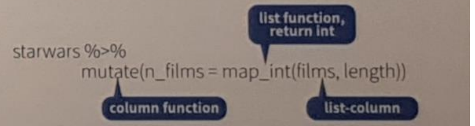
Manipulate list-columns like any other kind of column, using **dplyr** functions like **mutate()** and **transmute()**. Because each element is a list, use **map functions** within a column function to manipulate each element.

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



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

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

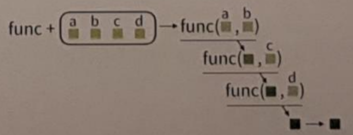


```
starwars %>%
      mutate(n_films = map_int(films, length))
```

Reduce

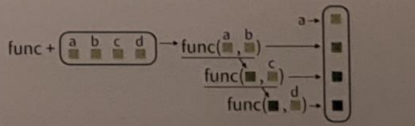
reduce(.x, .f, ..., .init, .dir = c("forward", "backward")) Apply function recursively to each element of a list or vector. Also **reduce2()**.

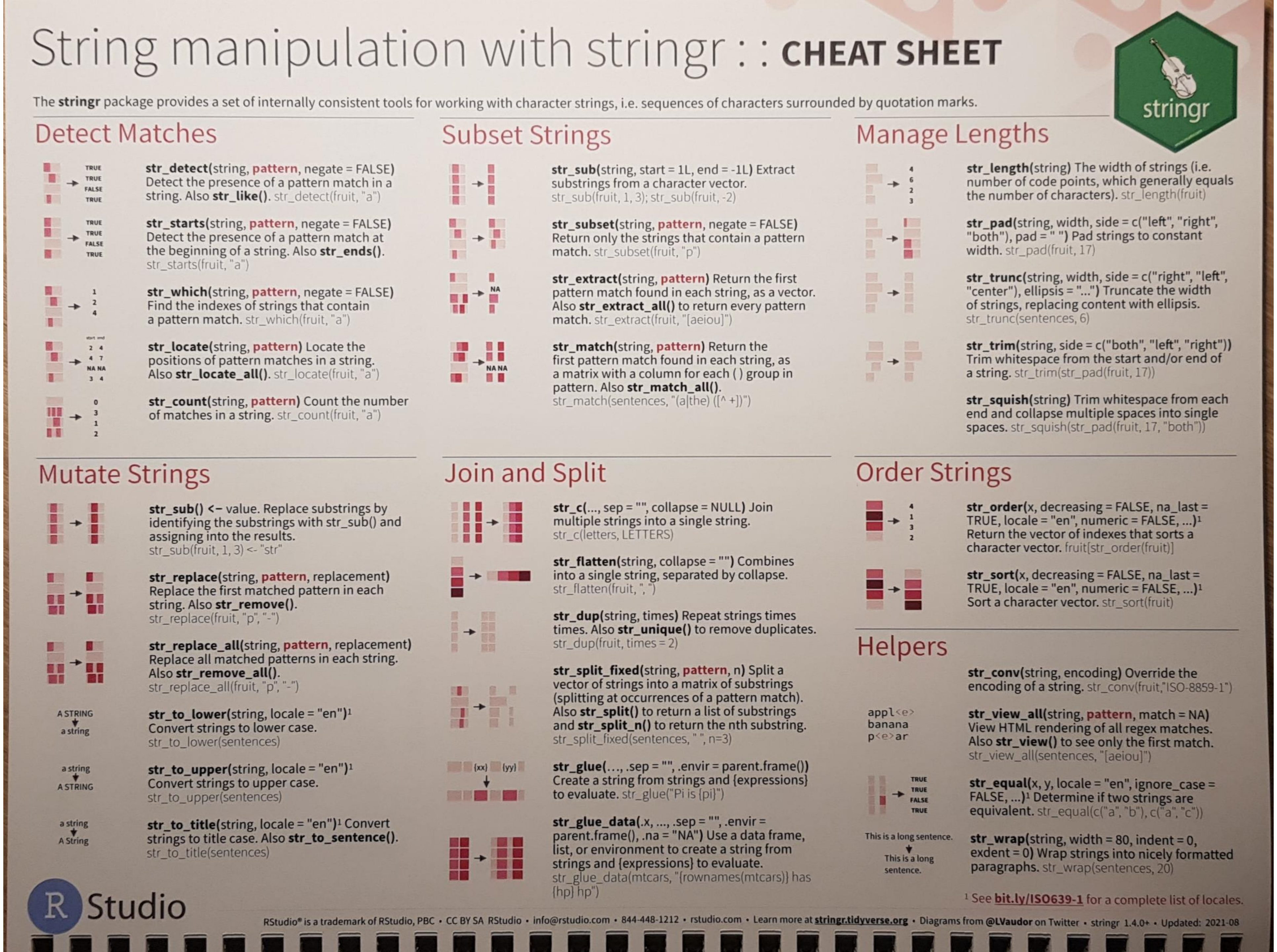
```
reduce(x, sum)
```



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

```
accumulate(x, sum)
```





String manipulation with stringr : : CHEAT SHEET

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

Detect Matches

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

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

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

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

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

Mutate Strings

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

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

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

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

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

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

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) ([^ +])")

Join and Split

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

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

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

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

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

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

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"))

Order Strings

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

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

Helpers

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

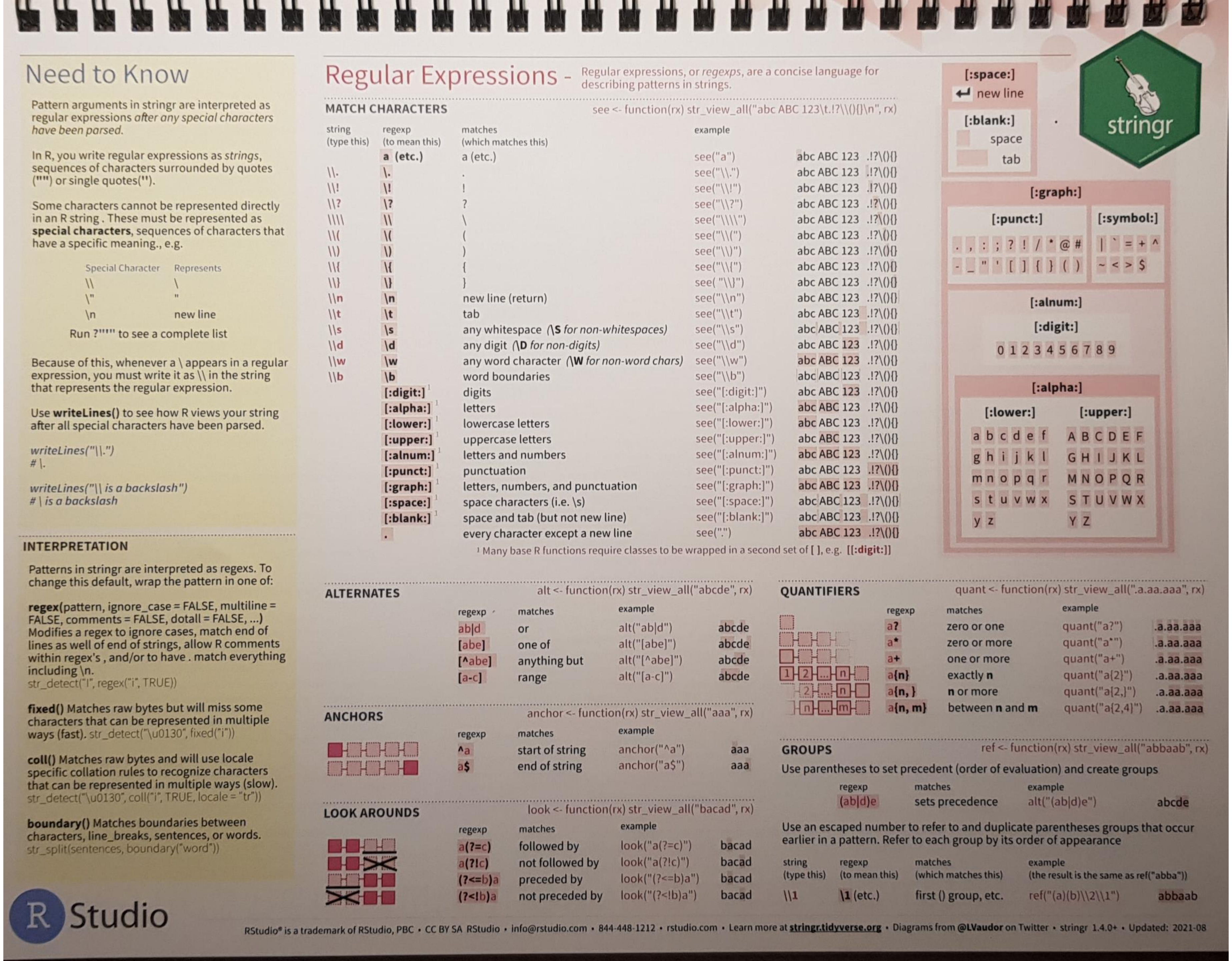
str_view_all(string, **pattern**, match = NA) View HTML rendering of all regex matches. Also **str_view()** to see only the first match. str_view_all(sentences, "[aeiou]")

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

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

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

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **stringr.tidyverse.org** • Diagrams from **@LVaudor** on Twitter • stringr 1.4.0+ • Updated: 2021-08



Need to Know

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

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

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

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

Run ?"''" to see a complete list

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

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

writeLines("\\.")
# \.

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

INTERPRETATION

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

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

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

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

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

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

MATCH CHARACTERS

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

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

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

[:space:]
↵ new line

[:blank:]
space
tab

[:graph:]

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

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

[:alnum:]

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

[:alpha:]

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

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

ALTERNATES

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

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

ANCHORS

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

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

LOOK AROUNDS

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

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

QUANTIFIERS

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

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

GROUPS

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

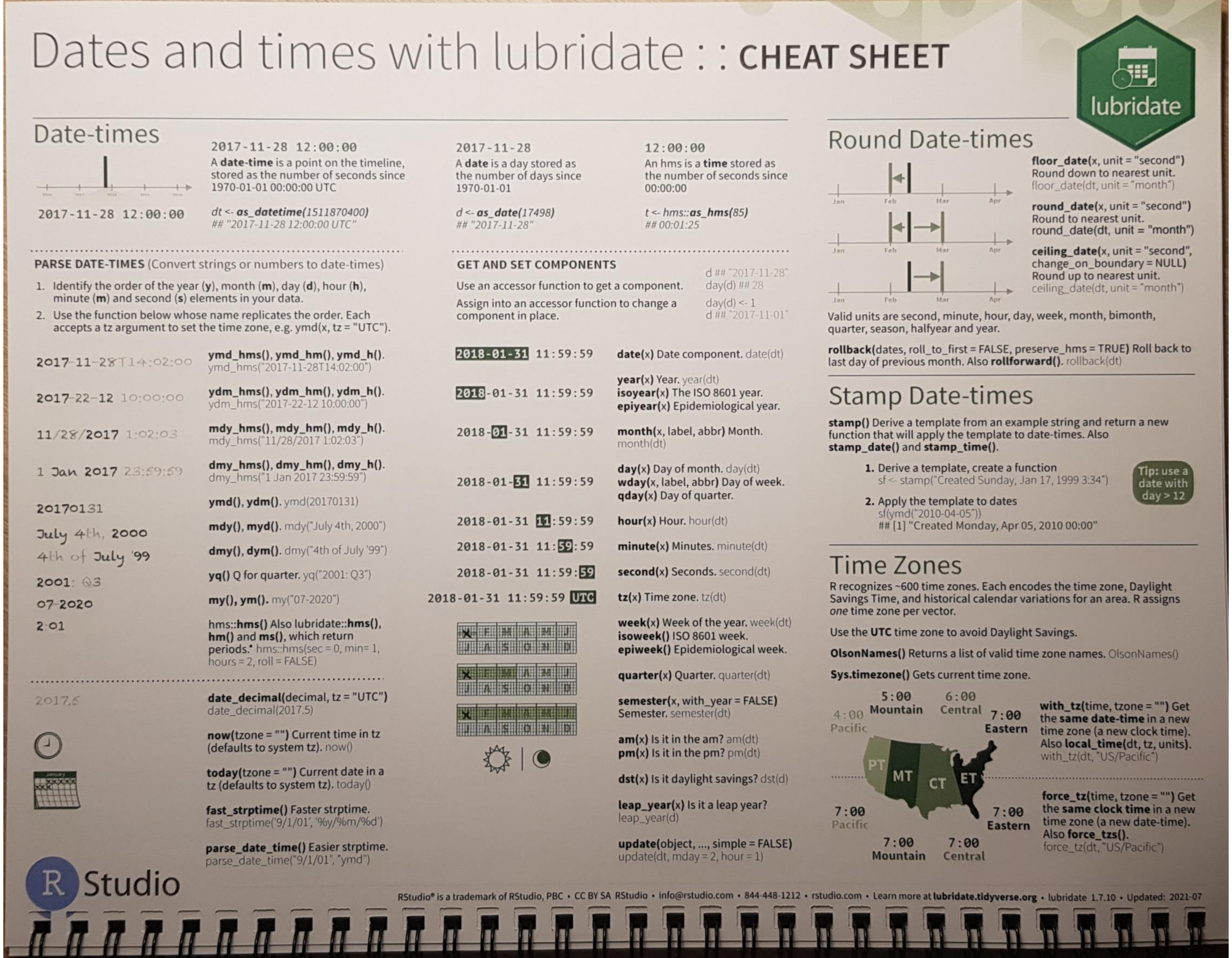
Use parentheses to set precedent (order of evaluation) and create groups

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

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

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

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **stringr.tidyverse.org** • Diagrams from **@LVaudor** on Twitter • stringr 1.4.0+ • Updated: 2021-08



Dates and times with lubridate : : CHEAT SHEET

lubridate

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(sec = 0, min= 1, hours = 2, roll = FALSE)
2017.5	**date_decimal**(decimal, tz = "UTC") date_decimal(2017.5)
	now(tzone = "") Current time in tz (defaults to system tz). now()
	today(tzone = "") Current date in a tz (defaults to system tz). today()
	fast_strptime() Faster strptime. fast_strptime('9/1/01', '%y/%m/%d')
	parse_date_time() Easier strptime. parse_date_time("9/1/01", "ymd")

GET AND SET COMPONENTS

Use an accessor function to get a component.

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

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

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

2018-01-31 11:59:59	**date**(x) Date component. date(dt)
2018-01-31 11:59:59	**year**(x) Year. year(dt) **isoyear**(x) The ISO 8601 year. **epiyear**(x) Epidemiological year.
2018-**01**-31 11:59:59	**month**(x, label, abbr) Month. month(dt)
2018-01-**31** 11:59:59	**day**(x) Day of month. day(dt) **wday**(x, label, abbr) Day of week. **qday**(x) Day of quarter.
2018-01-31 **11**:59:59	**hour**(x) Hour. hour(dt)
2018-01-31 11:**59**:59	**minute**(x) Minutes. minute(dt)
2018-01-31 11:59:**59**	**second**(x) Seconds. second(dt)
2018-01-31 11:59:59 **UTC**	**tz**(x) Time zone. tz(dt)
	week(x) Week of the year. week(dt) **isoweek()** ISO 8601 week. **epiweek()** Epidemiological week.
	quarter(x) Quarter. quarter(dt)
	semester(x, with_year = FALSE) Semester. semester(dt)
☼ | ◐	**am**(x) Is it in the am? am(dt) **pm**(x) Is it in the pm? pm(dt)
	dst(x) Is it daylight savings? dst(d)
	leap_year(x) Is it a leap year? leap_year(d)
	update(object, ..., simple = FALSE) update(dt, mday = 2, hour = 1)

Round Date-times

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

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

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

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

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

Stamp Date-times

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

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

Tip: use a date with day > 12

Time Zones

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

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

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

Sys.timezone() Gets current time zone.

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

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

RStudio

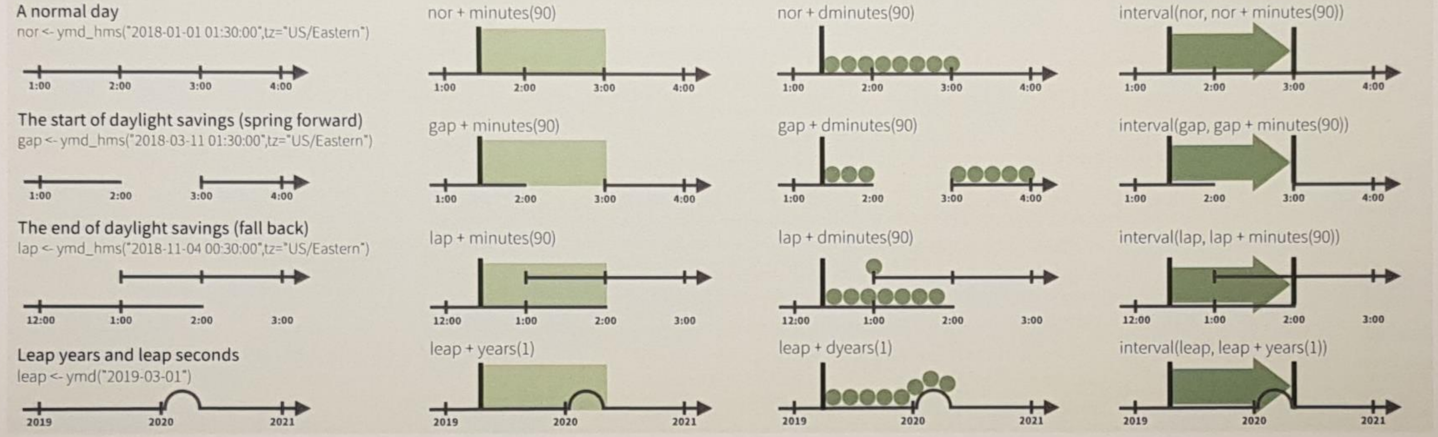
Math with Date-times — Lubridate provides three classes of timespans to facilitate math with dates and date-times.

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

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

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

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



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")
```

```
nor + minutes(90)
gap + minutes(90)
lap + minutes(90)
leap + years(1)
```

```
nor + dminutes(90)
gap + dminutes(90)
lap + dminutes(90)
leap + dyears(1)
```

```
interval(nor, nor + minutes(90))
interval(gap, gap + minutes(90))
interval(lap, lap + minutes(90))
interval(leap, leap + years(1))
```

lubridate

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

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

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

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

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

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

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

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

PERIODS

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

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

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

Number of months | Number of days | etc.

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

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

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

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

DURATIONS

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

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

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

Exact length in seconds | Equivalent in common units

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

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

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

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

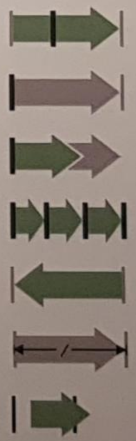
INTERVALS

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

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

Start Date | End Date

```
i <- interval(ymd("2017-01-01"), d)      ## 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())

RStudio

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **lubridate.tidyverse.org** • lubridate 1.7.10 • Updated: 2021-07

rmarkdown : : CHEAT SHEET

What is rmarkdown?

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

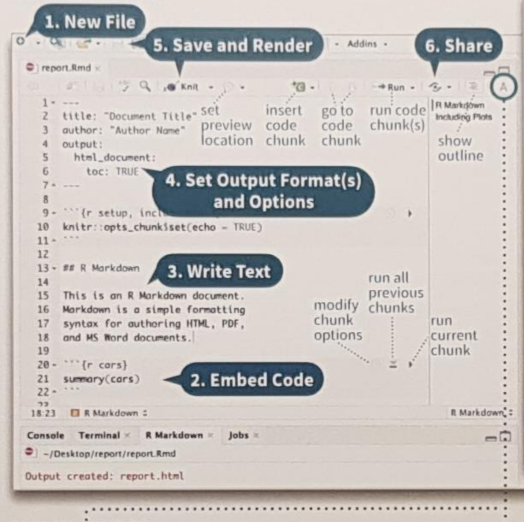
Dynamic Documents · Knit together plots, tables, and results with narrative text. Render to a variety of formats like HTML, PDF, MS Word, or MS Powerpoint.

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

Workflow

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

SOURCE EDITOR



1. New File

5. Save and Render

6. Share

4. Set Output Format(s) and Options

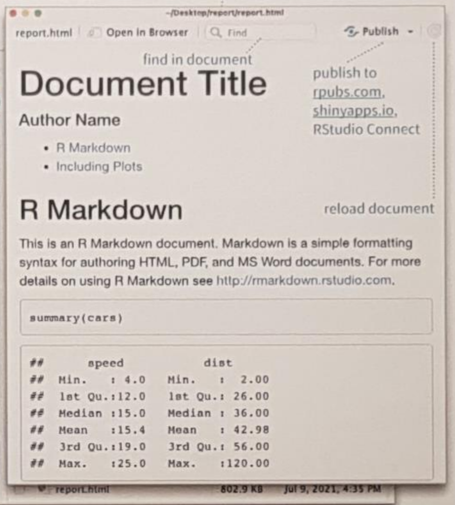
3. Write Text

2. Embed Code

~~~
---
title: "Document Title"
author: "Author Name"
output:
  html_document:
    toc: TRUE
---

```{r setup, incl
knitr::opts_chunk$set(echo = TRUE)
```

## R Markdown

This is an R Markdown document.
Markdown is a simple formatting
syntax for authoring HTML, PDF,
and MS Word documents.

```{r cars}
summary(cars)
```
~~~

```
~/Desktop/report/report.Rmd

Output created: report.html
```

RENDERED OUTPUT



file path to output document
~/Desktop/report/report.html
report.html
Open in Browser
Find
Publish
find in document
publish to rpubs.com, shinyapps.io, RStudio Connect
reload document
report.html
802.9 KB
Jul 9, 2021, 4:35 PM

Document Title

Author Name

- R Markdown
- Including Plots

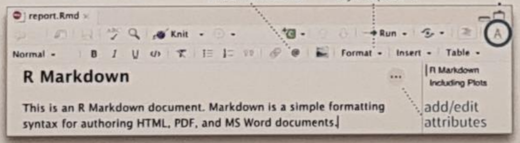
R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

VISUAL EDITOR



R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.

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).

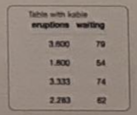


Table with kable

eruptions	waiting
3.600	79
1.800	54
3.333	74
2.283	62

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

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.	Plain text.
End a line with two spaces to start a new paragraph.	End a line with two spaces to start a new paragraph.
Also end with a backslash\ to make a new line.	Also end with a backslash to make a new line.
italics and **bold**	*italics* and **bold**
superscript^2^/subscript~2~	superscript^2/subscript$_2$
`~~strikethrough~~`	~~strikethrough~~
escaped: `\* \_ \\`	escaped: * _ \
endash: `--`, emdash: `---`	endash: –, emdash: —
# Header 1	**Header 1**
## Header 2	**Header 2**
...	...
###### Header 6	**Header 6**
- unordered list - 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	1. ordered list 2. item 2 • item 2a (indent 1 tab) • item 2b
<link url>	http://www.rstudio.com/
[This is a link.](link url)	This is a link.
[This is another link][id]. **At the end of the document:** [id]: link url	This is another link.
![Caption](image.png) or ![Caption][id2] **At the end of the document:** [id2]: image.png	Caption.
`verbatim code`	`verbatim code`

```
multiple lines
of verbatim code
```
	`multiple lines`
	`of verbatim code`

> block quotes	block quotes
equation: `$e^{i \pi} + 1 = 0$`	equation: $e^{i\pi} + 1 = 0$
equation block: `$$E = mc^{2}$$`	equation block: $$E = mc^2$$
horizontal rule: ---	horizontal rule:

```
| Right | Left | Default | Center |
|------:|:-----|---------|:------:|
| 12 | 12 | 12 | 12 |
| 123 | 123 | 123 | 123 |
| 1 | 1 | 1 | 1 |
```

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

HTML Tabsets

```
# Results {.tabset}
## Plots text
text

## Tables
more text
```

Results

Plots Tables

text

Set Output Formats and their Options in YAML

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

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

Indent format 2 characters, indent options 4 characters

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

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

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

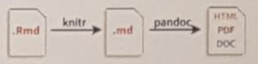
IMPORTANT OPTIONS	DESCRIPTION	HTML	PDF	MS Word	MS PPT
anchor_sections	Show section anchors on mouse hover (TRUE or FALSE)	X			
citation_package	The LaTeX package to process citations ("default", "natbib", "biblatex")		X		
code_download	Give readers an option to download the .Rmd source code (TRUE or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

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

Render

When you render a document, rmarkdown:

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



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

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

Share

Publish on RStudio Connect to share R Markdown documents securely, schedule automatic updates, and interact with parameters in real time.
rstudio.com/products/connect/

More Header Options

PARAMETERS

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

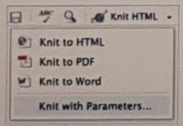
1. **Add parameters** in the header as sub-values of params.

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

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

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

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



REUSABLE TEMPLATES

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

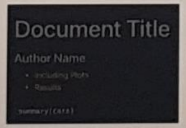
```
---
name: "My Template"
---
```

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

BOOTSWATCH THEMES

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

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



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

CUSTOM THEMES

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

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

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

STYLING WITH CSS AND SCSS

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

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

Apply CSS styling by writing HTML tags directly or:

- Use markdown to apply style attributes inline.

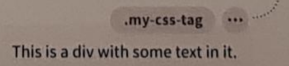
Bracketed Span
A [green]{.my-color} word.

A green word.

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

All of these words are green.

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



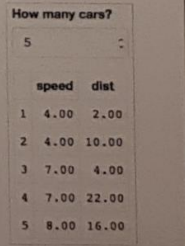
This is a div with some text in it.

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**.

Shiny : : CHEAT SHEET

Building 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).

Save your template as **app.R**. Keep your app in a directory along with optional extra files.

To generate the template, type **shinyapp** and press **Tab** in the RStudio IDE or go to **File > New Project > New Directory > Shiny Web Application**

In **ui** nest R functions to build an HTML interface

Tell the **server** how to render outputs and respond to inputs with R

```
# app.R
library(shiny)

ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

server <- function(input, output, session) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

shinyApp(ui = ui, server = server)
```

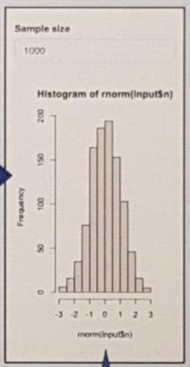
Customize the UI with **Layout Functions**

Add Inputs with ***Input() functions**

Add Outputs with ***Output() functions**

Wrap code in **render*() functions** before saving to output

Refer to UI inputs with **input$<id>** and outputs with **output$<id>**



Call **shinyApp()** to combine **ui** and **server** into an interactive app!

```
●●● app-name
    app.R
    DESCRIPTION
    README
    R/
    www/
```

- The directory name is the app name
- (optional) used in showcase mode
- (optional) directory of supplemental .R files that are sourced automatically, must be named "**R**"
- (optional) directory of files to share with web browsers (images, CSS, .js, etc.), must be named "**www**"

Launch apps stored in a directory with **runApp**(<path to directory>).

See annotated examples of Shiny apps by running **runExample**(<example name>). Run **runExample()** with no arguments for a list of example names.

Share

Share your app in three ways:

1. **Host it on shinyapps.io**, a cloud based service from RStudio. 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 directory>")
2. **Purchase RStudio Connect**, a publishing platform for R and Python. **rstudio.com/products/connect/**
3. **Build your own Shiny Server** **rstudio.com/products/shiny/shiny-server/**

Outputs

render*() and *Output() functions work together to add R output to the UI.

DT::**renderDataTable(**expr, options, searchDelay, callback, escape, env, quoted, outputArgs**)**	**dataTableOutput(**outputId**)**
renderImage(expr, env, quoted, deleteFile, outputArgs**)**	**imageOutput(**outputId, width, height, click, dblclick, hover, brush, inline**)**
renderPlot(expr, width, height, res, …, alt, env, quoted, execOnResize, outputArgs**)**	**plotOutput(**outputId, width, height, click, dblclick, hover, brush, inline**)**
renderPrint(expr, env, quoted, width, outputArgs**)**	**verbatimTextOutput(**outputId, placeholder**)**
renderTable(expr, striped, hover, bordered, spacing, width, align, rownames, colnames, digits, na, …, env, quoted, outputArgs**)**	**tableOutput(**outputId**)**
foo **renderText(**expr, env, quoted, outputArgs, sep**)**	**textOutput(**outputId, container, inline**)**

renderUI(expr, env, quoted, outputArgs**)**	**uiOutput(**outputId, inline, container, …**)**
	htmlOutput(outputId, inline, container, …**)**

These are the core output types. See **htmlwidgets.org** for many more options.

Inputs

Collect values from the user.

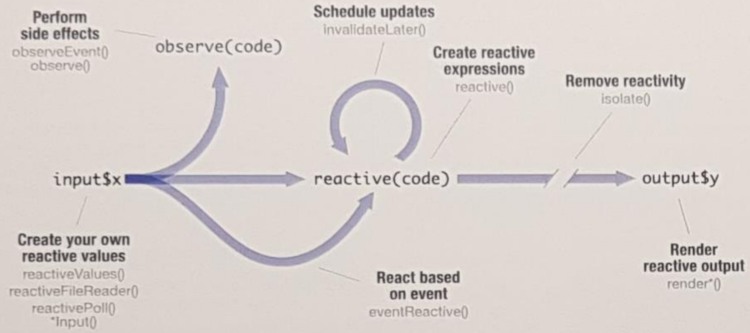
Access the current value of an input object with **input$<inputId>**. Input values are **reactive**.

Action	**actionButton(**inputId, label, icon, width, …**)**
Link	**actionLink(**inputId, label, icon, …**)**
☑ Choice 1 ☑ Choice 2 ☐ Choice 3	**checkboxGroupInput(**inputId, label, choices, selected, inline, width, choiceNames, choiceValues**)**
☑ Check me	**checkboxInput(**inputId, label, value, width**)**
	dateInput(inputId, label, value, min, max, format, startview, weekstart, language, width, autoclose, datesdisabled, daysofweekdisabled**)**
	dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator, width, autoclose**)**
Choose File	**fileInput(**inputId, label, multiple, accept, width, buttonLabel, placeholder**)**
1	**numericInput(**inputId, label, value, min, max, step, width**)**
....	**passwordInput(**inputId, label, value, width, placeholder**)**
● Choice A ○ Choice B ○ Choice C	**radioButtons(**inputId, label, choices, selected, inline, width, choiceNames, choiceValues**)**
Choice 1 Choice 1 Choice 2	**selectInput(**inputId, label, choices, selected, multiple, selectize, width, size**)** Also **selectizeInput()**
	sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post, timeFormat, timezone, dragRange**)**
Apply Changes	**submitButton(**text, icon, width**)** (Prevent reactions for entire app)
Enter text	**textInput(**inputId, label, value, width, placeholder**)** Also **textAreaInput()**

RStudio

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 YOUR OWN REACTIVE VALUES

```
# *Input() example
ui <- fluidPage(
 textInput("a","","A")
)
```

```
#reactiveValues example
server <-
function(input,output){
 rv <- reactiveValues()
 rv$number <- 5
}
```

***Input() functions**
(see front page)

Each input function creates a reactive value stored as **input$<inputId>**.

reactiveValues(...)

Creates a list of reactive values whose values you can set.

CREATE REACTIVE EXPRESSIONS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textInput("z","","Z"),
 textOutput("b"))

server <-
function(input,output){
 re <- reactive({
paste(input$a,input$z)})
 output$b <- renderText({
   re()
  })
}
shinyApp(ui, server)
```

reactive(x, env, quoted, label, domain**)**

Reactive expressions:

- **cache** their value to reduce computation
- can be called elsewhere
- notify dependencies when invalidated

Call the expression with function syntax, e.g. `re()`.

REACT BASED ON EVENT

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go"),
 textOutput("b")
)

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

shinyApp(ui, server)
```

eventReactive(eventExpr, valueExpr, event.env, event.quoted, value.env, value.quoted, ..., label, domain, ignoreNULL, ignoreInit**)**

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

RENDER REACTIVE OUTPUT

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

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

shinyApp(ui, server)
```

render*() functions
(see front page)

Builds an object to display. Will rerun code in body to rebuild the object whenever a reactive value in the code changes.

Save the results to **output$<outputId>**.

PERFORM SIDE EFFECTS

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 actionButton("go","Go")
)

server <-
function(input,output){
 observeEvent(input$go,{
   print(input$a)
  })
}

shinyApp(ui, server)
```

observeEvent(eventExpr, handlerExpr, event.env, event.quoted, handler.env, handler.quoted, ..., label, suspended, priority, domain, autoDestroy, ignoreNULL, ignoreInit, once**)**

Runs code in 2nd argument when reactive values in 1st argument change. See **observe()** for alternative.

REMOVE REACTIVITY

```
library(shiny)
ui <- fluidPage(
 textInput("a","","A"),
 textOutput("b")
)

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

shinyApp(ui, server)
```

isolate(expr**)**

Runs a code block. Returns a **non-reactive** copy of the results.

UI - An app's UI is an HTML document.

Use Shiny's functions to assemble this HTML with R.

```
fluidPage(
  textInput("a","")                 Returns
)                                    HTML
## <div class="container-fluid">
##  <div class="form-group shiny-input-container">
##     <label for="a"></label>
##     <input id="a" type="text"
##        class="form-control" value=""/>
##   </div>
## </div>
```

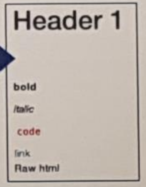
HTML

Add static HTML elements with **tags**, a list of functions that parallel common HTML tags, e.g. `tags$a()`. Unnamed arguments will be passed into the tag; named arguments will become tag attributes.

Run **names(tags)** for a complete list.
`tags$h1("Header") -> <h1>Header</h1>`

The most common tags have wrapper functions. You do not need to prefix their names with **tags$**

```
ui <- fluidPage(
  h1("Header 1"),
  hr(),
  br(),
  p(strong("bold")),
  p(em("italic")),
  p(code("code")),
  a(href="", "link"),
  HTML("<p>Raw html</p>")
)
```



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(rel = "stylesheet",
  type = "text/css", href = "<file name>"))
```

JS

To include JavaScript, use **includeScript()** or
1. Place the file in the **www** subdirectory
2. Link to it with:

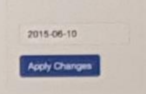
```
tags$head(tags$script(src = "<file name>"))
```

!"#$%& To include an image:
1. Place the file in the **www** subdirectory
2. Link to it with `img(src="<file name>")`

Layouts

Combine multiple elements into a "single element" that has its own properties with a panel function, e.g.

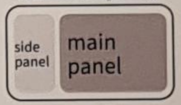
```
wellPanel(
  dateInput("a", ""),
  submitButton()
)
```



absolutePanel()	navlistPanel()
conditionalPanel()	sidebarPanel()
fixedPanel()	tabPanel()
headerPanel()	tabsetPanel()
inputPanel()	titlePanel()
mainPanel()	wellPanel()

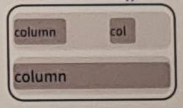
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

sidebarLayout()



```
ui <- fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
)
```

fluidRow()

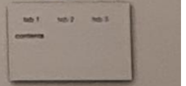


```
ui <- fluidPage(
 fluidRow(column(width = 4),
   column(width = 2, offset = 3)),
 fluidRow(column(width = 12))
)
```

Also **flowLayout(), splitLayout(), verticalLayout(), fixedPage(), and fixedRow().**

Layer tabPanels on top of each other, and navigate between them, with:

```
ui <- fluidPage( tabsetPanel(
   tabPanel("tab 1", "contents"),
   tabPanel("tab 2", "contents"),
   tabPanel("tab 3", "contents")))
```



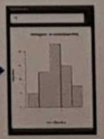
```
ui <- fluidPage( navlistPanel(
   tabPanel("tab 1", "contents"),
   tabPanel("tab 2", "contents"),
   tabPanel("tab 3", "contents")))
```

```
ui <- navbarPage(title = "Page",
   tabPanel("tab 1", "contents"),
   tabPanel("tab 2", "contents"),
   tabPanel("tab 3", "contents"))
```

Themes

Use the **bslib** package to add existing themes to your Shiny app ui, or make your own.

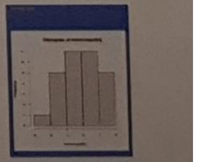
```
library(bslib)
ui <- fluidPage(
  theme = bs_theme(
    bootswatch = "darkly",
    ...
  )
)
```



bootswatch_themes() Get a list of themes.

Build your own theme by customizing individual arguments.

```
bs_theme(bg = "#558AC5",
         fg = "#F9B02D",
         ...)
```



?bs_theme for a full list of arguments.

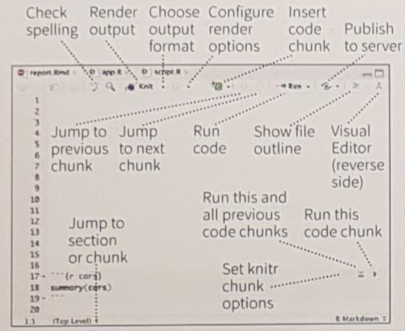
bs_themer() Place within the server function to use the interactive theming widget.

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **shiny.rstudio.com** • Font Awesome 5.15.3 • shiny 1.6.0 • Updated: 2021-07

RStudio IDE : : CHEAT SHEET

Documents and Apps

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



Check spelling

Render output

Choose output format

Configure render options

Insert code chunk

Publish to server

Jump to previous chunk

Jump to next chunk

Run code

Show file outline

Visual Editor (reverse side)

Run this and all previous code chunks

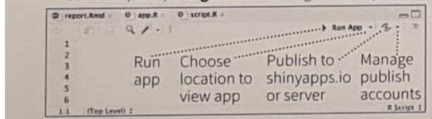
Run this code chunk

Jump to section or chunk

Set knitr chunk options

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



Run app

Choose location to view app

Publish to shinyapps.io or server

Manage publish accounts

Source Editor

Navigate backwards/ forwards

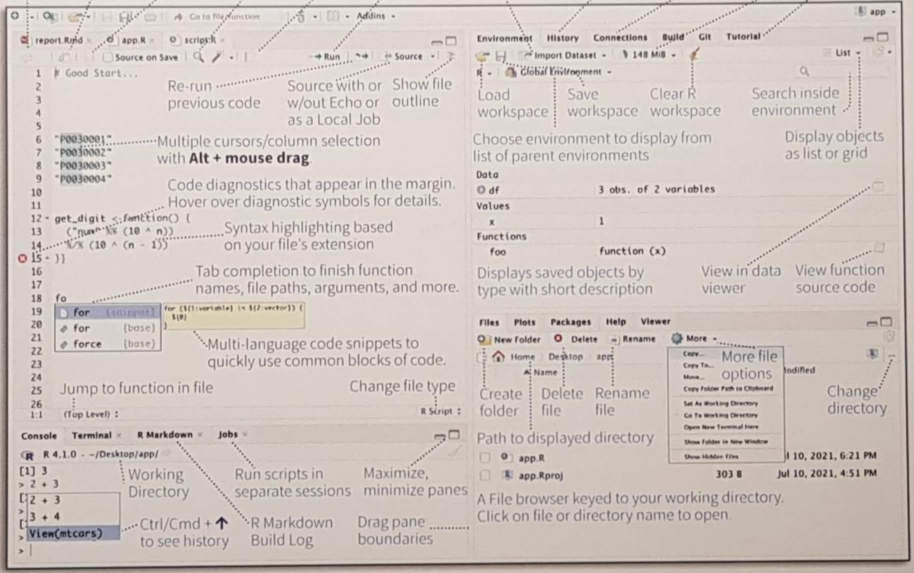
Open in new window

Save

Find and replace

Compile as notebook

Run selected code



```
# Good Start...

"P0030001"
"P0030002"
"P0030003"
"P0030004"

get_digit <- function() {
  ("num" %/% (10 ^ n))
  %/% (10 ^ (n - 1))
}

fo
```

Re-run previous code

Source with or w/out Echo or as a Local Job

Show file outline

Multiple cursors/column selection with **Alt + mouse drag**.

Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.

Syntax highlighting based on your file's extension

Tab completion to finish function names, file paths, arguments, and more.

Multi-language code snippets to quickly use common blocks of code.

Jump to function in file

Change file type

```
R 4.1.0 · ~/Desktop/app/
[1] 3
> 2 + 3
> 3 + 4
> View(mtcars)
>
```

Working Directory

Run scripts in separate sessions

Maximize, minimize panes

Ctrl/Cmd + ↑ to see history

R Markdown Build Log

Drag pane boundaries

Tab Panes

Import data with wizard

History of past commands to run/copy

Manage external databases

View memory usage

R tutorials

Load workspace

Save workspace

Clear R workspace

Search inside environment

Choose environment to display from list of parent environments

Display objects as list or grid

Displays saved objects by type with short description

View in data viewer

View function source code

More file options

Create folder

Delete file

Rename file

Change directory

Path to displayed directory

A File browser keyed to your working directory. Click on file or directory name to open.

Version Control

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

A Added D Deleted M Modified R Renamed ? Untracked

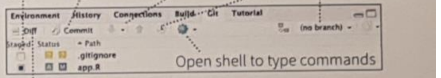
Stage files:

Commit staged files

Push/Pull to remote

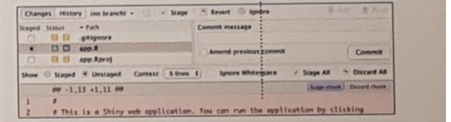
View History

Current branch



Open shell to type commands

Show file diff to view file differences

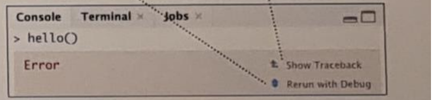


Debug Mode

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

Launch debugger mode from origin of error

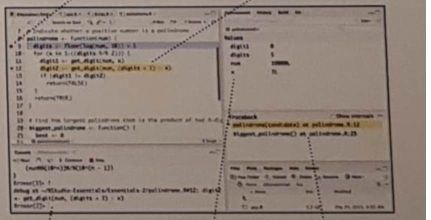
Open traceback to examine the functions that R called before the error occurred



```
> hello()
Error
```

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

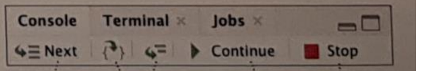
Highlighted line shows where execution has paused



Run commands in environment where execution has paused

Examine variables in executing environment

Select function in traceback to debug



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**

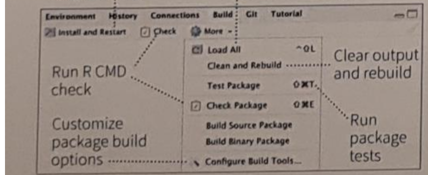
Enable roxygen documentation with **Tools > Project Options > Build Tools**

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

See package information in the **Build Tab**

Install package and restart R

Run devtools::load_all() and reload changes



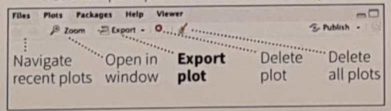
Run R CMD check

Customize package build options

Clear output and rebuild

Run package tests

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



Navigate recent plots

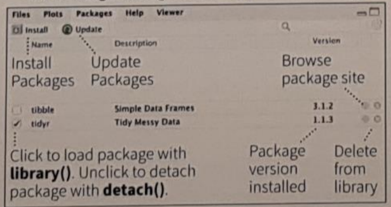
Open in window

Export plot

Delete plot

Delete all plots

GUI **Package** manager lists every installed package



Install Packages

Update Packages

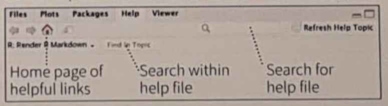
Browse package site

Click to load package with **library()**. Unclick to detach package with **detach()**.

Package version installed

Delete from library

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

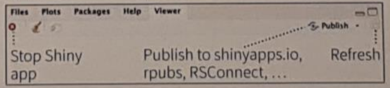


Home page of helpful links

Search within help file

Search for help file

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

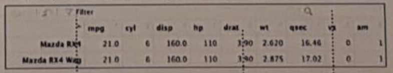


Stop Shiny app

Publish to shinyapps.io, rpubs, RSConnect, …

Refresh

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



Filter rows by value or value range

Sort by values

Search for value

RStudio

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at rstudio.com • Font Awesome 5.15.3 • RStudio IDE 1.4.1717 • Updated: 2021-07

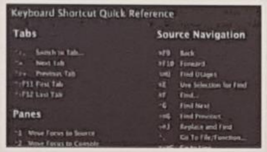
Keyboard Shortcuts

RUN CODE	Windows/Linux	Mac
Search command history	Ctrl+↑	Cmd+↑
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 %>% (pipe operator)	Ctrl+Shift+M	Cmd+Shift+M
(Un)Comment selection	Ctrl+Shift+C	Cmd+Shift+C

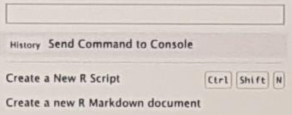
MAKE PACKAGES	Windows/Linux	Mac
Load All (devtools)	Ctrl+Shift+L	Cmd+Shift+L
Test Package (Desktop)	Ctrl+Shift+T	Cmd+Shift+T
Document Package	Ctrl+Shift+D	Cmd+Shift+D

DOCUMENTS AND APPS		
Knit 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

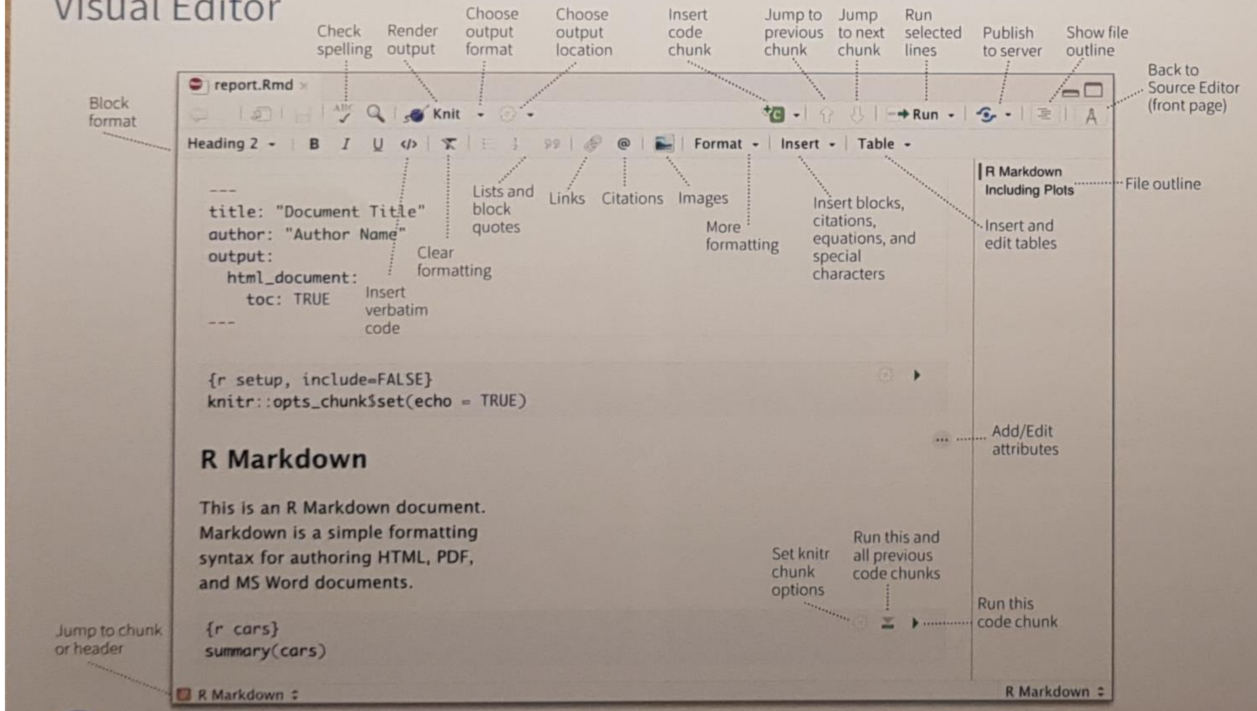
View the Keyboard Shortcut Quick Reference with **Tools > Keyboard Shortcuts** or **Alt/Option + Shift + K**



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


History Send Command to Console
Create a New R Script
Ctrl Shift N
Create a new R Markdown document

Visual Editor



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

```
{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.

```
{r cars}
summary(cars)
```

RStudio Workbench

WHY RSTUDIO WORKBENCH?

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

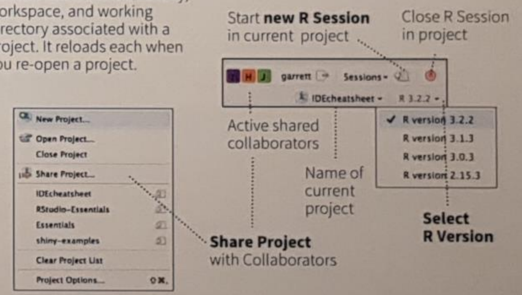
- open and run multiple R sessions at once
- tune your resources to improve performance
- administrative tools for managing user sessions
- collaborate real-time with others in shared projects
- switch easily from one version of R to a different version
- integrate with your authentication, authorization,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

Download a free 45 day evaluation at **www.rstudio.com/products/workbench/evaluation/**

Share Projects

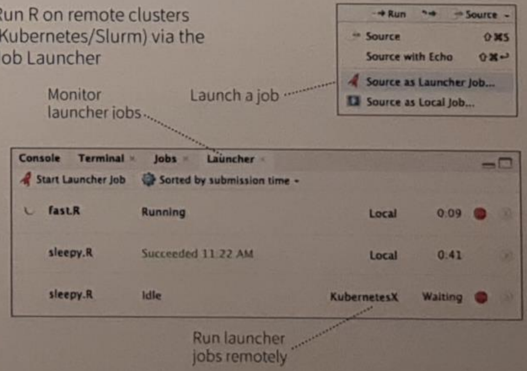
File > New Project

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



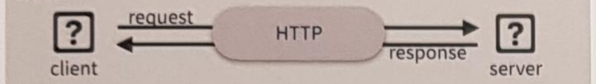
RStudio

REST APIs with plumber: : CHEAT SHEET

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
```

Path

HTTP Method

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.

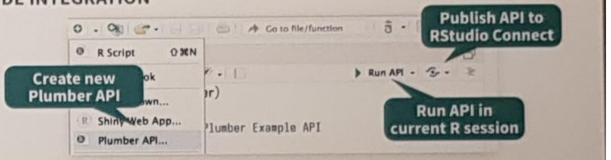
```
library(plumber)

plumb("plumber.R") %>%
  pr_run(port = 5762)
```

Path to API definition

Specify API port

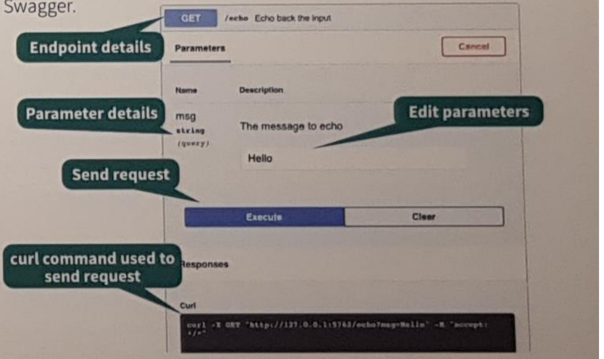
This runs the API on the host machine supported by the current R session.

IDE INTEGRATION



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.



```
curl -X GET "http://127.0.0.1:5762/echo?msg=Hello" -H "accept: */*"
```

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'\"]}"
```

RStudio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at www.rplumber.io • plumber 1.1.0 • Updated: 2021-03

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()`**.

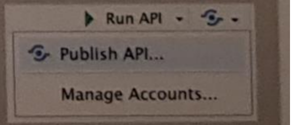
```
library(plumber)

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 RStudio Connect, which supports push button publishing from the RStudio IDE.

Use Python with R with reticulate : : CHEAT SHEET

The **reticulate** package lets you use Python and R together seamlessly in R code, in R Markdown documents, and in the RStudio IDE.

Python in R Markdown

(Optional) Build Python env to use.

Add **knitr::knit_engines$set(python = reticulate::eng_python)** to the setup chunk to set up the reticulate Python engine (not required for knitr >= 1.18).

Suggest the Python environment to use, in your setup chunk.

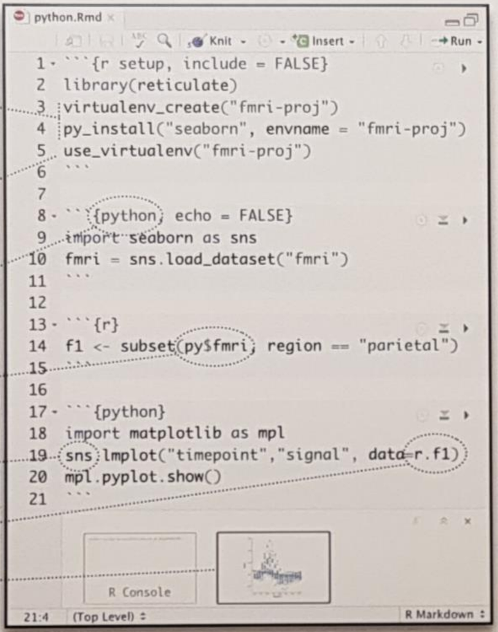
Begin Python chunks with **```{python}**. Chunk options like **echo**, **include**, etc. all work as expected.

Use the **py** object to access objects created in Python chunks from R chunks.

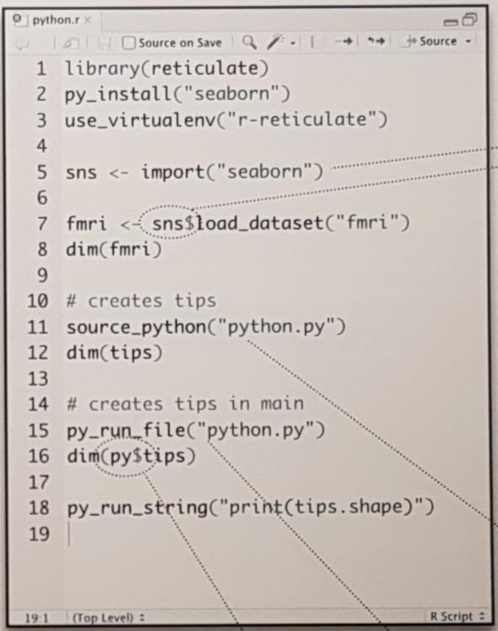
Python chunks all execute within a **single** Python session so you have access to all objects created in previous chunks.

Use the **r** object to access objects created in R chunks from Python chunks.

Output displays below chunk, including matplotlib plots.



~~~
```{r setup, include = FALSE}
library(reticulate)
virtualenv_create("fmri-proj")
py_install("seaborn", envname = "fmri-proj")
use_virtualenv("fmri-proj")
```

```{python, echo = FALSE}
import seaborn as sns
fmri = sns.load_dataset("fmri")
```

```{r}
f1 <- subset(py$fmri, region == "parietal")
```

```{python}
import matplotlib as mpl
sns.lmplot("timepoint","signal", data=r.f1)
mpl.pyplot.show()
```
~~~



```
library(reticulate)
py_install("seaborn")
use_virtualenv("r-reticulate")

sns <- import("seaborn")

fmri <- sns$load_dataset("fmri")
dim(fmri)

# creates tips
source_python("python.py")
dim(tips)

# creates tips in main
py_run_file("python.py")
dim(py$tips)

py_run_string("print(tips.shape)")

```

Python in R

Call Python from R code in three ways:

IMPORT PYTHON MODULES

Use **import()** to import any Python module. 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. Also **import_from_path()**. import("pandas")
- **import_main**(convert = TRUE) Import the main module, where Python executes code by default. import_main()
- **import_builtins**(convert = TRUE) Import Python's built-in functions. import_builtins()

SOURCE PYTHON FILES

Use **source_python()** to source a Python script and make the Python functions and objects it creates available in the calling R environment.

- **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 with **py_run_file()** or **py_run_string()**.

- **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
- **py_run_file**(file, local = FALSE, convert = TRUE) Run Python file in the main module. py_run_file("script.py")
- **py_eval**(code, convert = TRUE) Run a Python expression, return the result. Also **py_call()**. py_eval("1 + 1")

Access the results, and anything else in Python's **main** module, with **py**.

- **py** An R object that contains the Python main module and the results stored there. py$x

Object Conversion

Tip: To index Python objects begin at 0, use integers, e.g. 0L

Reticulate provides **automatic** built-in conversion between Python and R for many Python types.

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

Or, if you like, you can convert manually with

py_to_r(x) Convert a Python object to an R object. Also **r_to_py()**. py_to_r(x)

tuple(..., convert = FALSE) Create a Python tuple. tuple("a", "b", "c")

dict(..., convert = FALSE) Create a Python dictionary object. Also **py_dict()** to make a dictionary that uses Python objects as keys. dict(foo = "bar", index = 42L)

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)

py_main_thread_func(f) Create a function that will always be called on the main thread.

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()**. iterate(iter, print)

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))

Helpers

py_capture_output(expr, type = c("stdout", "stderr")) Capture and return Python output. Also **py_suppress_warnings()**. py_capture_output("x")

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_get_attr(x)

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_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!")})
```

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at rstudio.github.io/reticulate/ • reticulate 1.20.0 • Updated: 2021-08

Python in the IDE

Requires reticulate plus RStudio v1.2+. Some features require v1.4+.

Syntax highlighting for Python scripts and chunks.

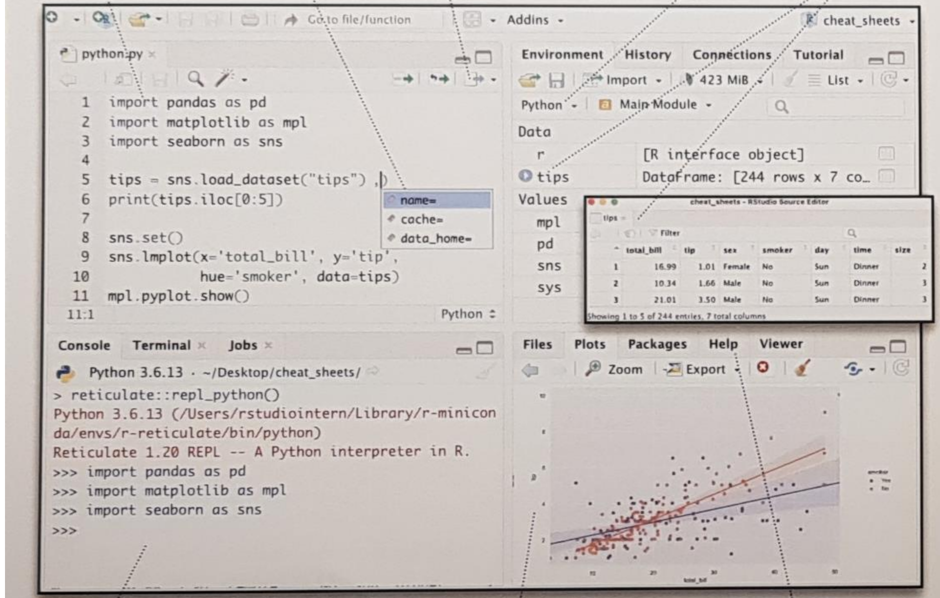
Tab completion for Python functions and objects (and Python modules imported in R scripts).

Source Python scripts.

Execute Python code line by line with **Cmd + Enter** (**Ctrl + Enter**).

View Python objects in the Environment Pane.

View Python objects in the Data Viewer.



```
import pandas as pd
import matplotlib as mpl
import seaborn as sns

tips = sns.load_dataset("tips") ,

print(tips.iloc[0:5])

sns.set()
sns.lmplot(x='total_bill', y='tip',
           hue='smoker', data=tips)
mpl.pyplot.show()
```

```
Data
  r       [R interface object]
  tips    DataFrame: [244 rows x 7 co…
Values
  mpl
  pd
  sns
  sys
```

	total_bill	tip	sex	smoker	day	time	size
1	16.99	1.01	Female	No	Sun	Dinner	2
2	10.34	1.66	Male	No	Sun	Dinner	3
3	21.01	3.50	Male	No	Sun	Dinner	3

Showing 1 to 5 of 244 entries, 7 total columns

```
Python 3.6.13 · ~/Desktop/cheat_sheets/
> reticulate::repl_python()
Python 3.6.13 (/Users/rstudiointern/Library/r-minicon
da/envs/r-reticulate/bin/python)
Reticulate 1.20 REPL -- A Python interpreter in R.
>>> import pandas as pd
>>> import matplotlib as mpl
>>> import seaborn as sns
>>>
```

A Python REPL opens in the console when you run Python code with a keyboard shortcut. Type **exit** to close.

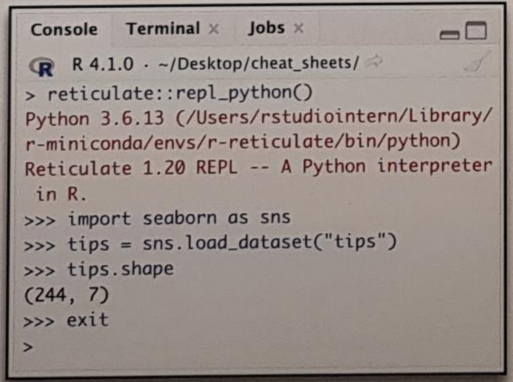
matplotlib plots display in plots pane.

Press **F1** over a Python symbol to display the help topic for that symbol.

Python REPL

A REPL (Read, Eval, Print Loop) is a command line where you can run Python code and view the results.

1. Open in the console with **repl_python()**, or by running code in a Python script with **Cmd + Enter** (**Ctrl + Enter**).
   - **repl_python**(module = NULL, quiet = getOption("reticulate.repl.quiet", default = FALSE), input = NULL) Launch a Python REPL. Run **exit** to close. repl_python()
2. Type commands at >>> prompt.
3. Press **Enter** to run code.
4. Type **exit** to close and return to R console.



```
R 4.1.0 · ~/Desktop/cheat_sheets/
> reticulate::repl_python()
Python 3.6.13 (/Users/rstudiointern/Library/
r-miniconda/envs/r-reticulate/bin/python)
Reticulate 1.20 REPL -- A Python interpreter
 in R.
>>> import seaborn as sns
>>> tips = sns.load_dataset("tips")
>>> tips.shape
(244, 7)
>>> exit
>
```

Configure Python

Reticulate binds to a local instance of Python when you first call **import()** directly or implicitly from an R session. To control the process, find or build your desired Python instance. Then suggest your instance to reticulate. **Restart R to unbind.**

Find Python

- **install_python**(version, list = FALSE, force = FALSE) Download and install Python. install_python("3.6.13")
- **py_available**(initialize = FALSE) Check if Python is available on your system. Also **py_module_available()** and **py_numpy_module().** py_available()
- **py_discover_config()** Return all detected versions of Python. Use **py_config()** to check which version has been loaded. py_config()
- **virtualenv_list()** List all available virtualenvs. Also **virtualenv_root()**. virtualenv_list()
- **conda_list**(conda = "auto") List all available conda envs. Also **conda_binary()** and **conda_version()**. conda_list()

Create a Python env

- **virtualenv_create**(envname = NULL, ...) Create a new virtual environment. virtualenv_create("r-pandas")
- **conda_create**(envname = NULL, ...) Create a new conda environment. conda_create("r-pandas", packages = "pandas")

Install Packages

Install Python packages with R (below) or the shell:

pip install SciPy
conda install SciPy

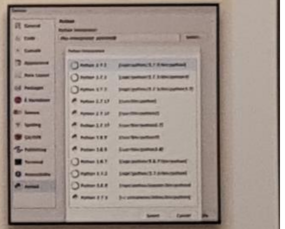
- **py_install**(packages, envname, ...) Installs Python packages into a Python env. py_install("pandas")
- **virtualenv_install**(envname, packages, ...) Install a package within a virtualenv. Also **virtualenv_remove()**. virtualenv_install("r-pandas", packages = "pandas")
- **conda_install**(envname, packages, ...) Install a package within a conda env. Also **conda_remove()**. conda_install("r-pandas", packages = "plotly")

Suggest an env to use

Set a default Python interpreter in the RStudio IDE Global or Project Options.

Go to **Tools > Global Options... > Python** for Global Options.

Within a project, go to **Tools > Project Options... >Python.**



Otherwise, to choose an instance of Python to bind to, reticulate scans the instances on your computer in the following order, **stopping at the first instance that contains the module called by import()**.

1. The instance referenced by the environment variable **RETICULATE_PYTHON** (if specified). **Tip: set in .Renviron file.**
   - **Sys.setenv**(RETICULATE_PYTHON = PATH) Set default Python binary. Persists across sessions! Undo with **Sys.unsetenv()**. Sys.setenv(RETICULATE_PYTHON = "/usr/local/bin/python")
2. The instances referenced by use_ functions if called before import(). Will fail silently if called after import unless **required = TRUE**.
   - **use_python**(python, required = FALSE) Suggest a Python binary to use by path. use_python("/usr/local/bin/python")
   - **use_virtualenv**(virtualenv = NULL, required = FALSE) Suggest a Python virtualenv. use_virtualenv("~/myenv")
   - **use_condaenv**(condaenv = NULL, conda = "auto", required = FALSE) Suggest a conda env to use. use_condaenv(condaenv = "r-nlp", conda = "/opt/anaconda3/bin/conda")
3. Within virtualenvs and conda envs that carry the same name as the imported module. e.g. ~/anaconda/envs/nltk for import("nltk")
4. At the location of the Python binary discovered on the system PATH (i.e. Sys.which("python"))
5. At customary locations for Python, e.g. /usr/local/bin/python, /opt/local/bin/python...

RStudio® is a trademark of RStudio, PBC • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at rstudio.github.io/reticulate/ • reticulate 1.20.0 • Updated: 2021-08

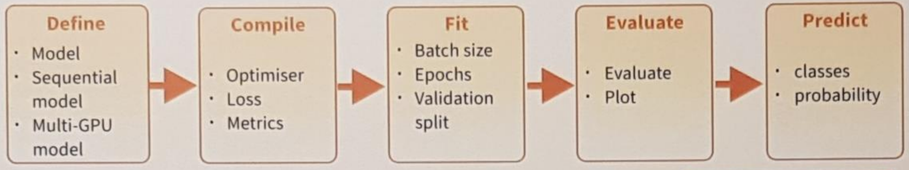
Deep Learning with Keras : : CHEAT SHEET

Keras TensorFlow

Intro

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation. It supports multiple back-ends, including TensorFlow, CNTK and Theano.

TensorFlow is a lower level mathematical library for building deep neural network architectures. The `keras` R package makes it easy to use Keras and TensorFlow in R.



Define
- Model
- Sequential model
- Multi-GPU model

Compile
- Optimiser
- Loss
- Metrics

Fit
- Batch size
- Epochs
- Validation split

Evaluate
- Evaluate
- Plot

Predict
- classes
- probability

https://keras.rstudio.com

https://www.manning.com/books/deep-learning-with-r

The "Hello, World!" of deep learning

Working with keras models

DEFINE A MODEL

keras_model() Keras Model

keras_model_sequential() Keras Model composed of a linear stack of layers

multi_gpu_model() Replicates a model on different GPUs

COMPILE A MODEL

compile(object, optimizer, loss, metrics = NULL**)** Configure a Keras model for training

FIT A MODEL

fit(object, x = NULL, y = NULL, batch_size = NULL, epochs = 10, verbose = 1, callbacks = NULL, …**)** Train a Keras model for a fixed number of epochs (iterations)

fit_generator() Fits the model on data yielded batch-by-batch by a generator

train_on_batch() test_on_batch() Single gradient update or model evaluation over one batch of samples

EVALUATE A MODEL

evaluate(object, x = NULL, y = NULL, batch_size = NULL**)** Evaluate a Keras model

evaluate_generator() Evaluates the model on a data generator

PREDICT

predict() Generate predictions from a Keras model

predict_proba() and **predict_classes()** Generates probability or class probability predictions for the input samples

predict_on_batch() Returns predictions for a single batch of samples

predict_generator() Generates predictions for the input samples from a data generator

OTHER MODEL OPERATIONS

summary() Print a summary of a Keras model

export_savedmodel() Export a saved model

get_layer() Retrieves a layer based on either its name (unique) or index

pop_layer() Remove the last layer in a model

save_model_hdf5(); load_model_hdf5() Save/Load models using HDF5 files

serialize_model(); unserialize_model() Serialize a model to an R object

clone_model() Clone a model instance

freeze_weights(); unfreeze_weights() Freeze and unfreeze weights

CORE LAYERS

layer_input() Input layer

layer_dense() Add a densely-connected NN layer to an output

layer_activation() Apply an activation function to an output

layer_dropout() Applies Dropout to the input

layer_reshape() Reshapes an output to a certain shape

layer_permute() Permute the dimensions of an input according to a given pattern

n **layer_repeat_vector()** Repeats the input n times

x f(x) **layer_lambda(**object, f**)** Wraps arbitrary expression as a layer

L1 L2 **layer_activity_regularization()** Layer that applies an update to the cost function based input activity

layer_masking() Masks a sequence by using a mask value to skip timesteps

layer_flatten() Flattens an input

INSTALLATION

The `keras` R package uses the Python keras library. You can install all the prerequisites directly from R.

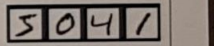
https://keras.rstudio.com/reference/install_keras.html

```
library(keras)
install_keras()
```

See ?install_keras for GPU instructions

This installs the required libraries in an Anaconda environment or virtual environment **'r-tensorflow'**.

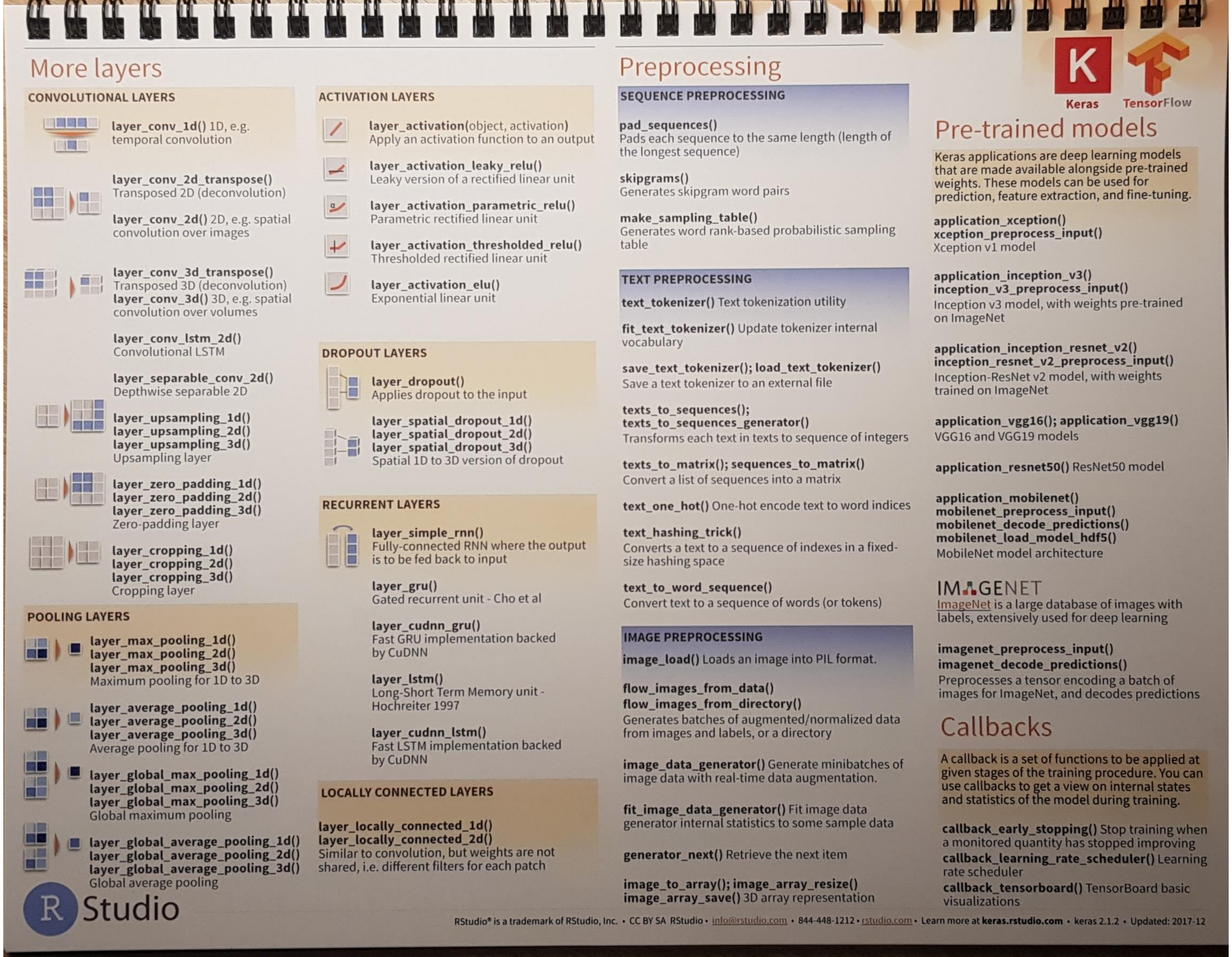
TRAINING AN IMAGE RECOGNIZER ON MNIST DATA



```
# input layer: use MNIST images
mnist <- dataset_mnist()
x_train <- mnist$train$x;  y_train <- mnist$train$y
x_test <- mnist$test$x;  y_test <- mnist$test$y

# reshape and rescale
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
x_train <- x_train / 255;  x_test <- x_test / 255

y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)

# defining the model and layers
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu',
              input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')

# compile (define loss and optimizer)
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_rmsprop(),
  metrics = c('accuracy')
)

# train (fit)
model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
model %>% evaluate(x_test, y_test)
model %>% predict_classes(x_test)
```

R Studio

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at keras.rstudio.com • keras 2.1.2 • Updated: 2017-12



Keras

TensorFlow

More layers

CONVOLUTIONAL LAYERS

layer_conv_1d() 1D, e.g. temporal convolution

layer_conv_2d_transpose() Transposed 2D (deconvolution)

layer_conv_2d() 2D, e.g. spatial convolution over images

layer_conv_3d_transpose() Transposed 3D (deconvolution)
layer_conv_3d() 3D, e.g. spatial convolution over volumes

layer_conv_lstm_2d() Convolutional LSTM

layer_separable_conv_2d() Depthwise separable 2D

layer_upsampling_1d()
layer_upsampling_2d()
layer_upsampling_3d()
Upsampling layer

layer_zero_padding_1d()
layer_zero_padding_2d()
layer_zero_padding_3d()
Zero-padding layer

layer_cropping_1d()
layer_cropping_2d()
layer_cropping_3d()
Cropping layer

POOLING LAYERS

layer_max_pooling_1d()
layer_max_pooling_2d()
layer_max_pooling_3d()
Maximum pooling for 1D to 3D

layer_average_pooling_1d()
layer_average_pooling_2d()
layer_average_pooling_3d()
Average pooling for 1D to 3D

layer_global_max_pooling_1d()
layer_global_max_pooling_2d()
layer_global_max_pooling_3d()
Global maximum pooling

layer_global_average_pooling_1d()
layer_global_average_pooling_2d()
layer_global_average_pooling_3d()
Global average pooling

ACTIVATION LAYERS

layer_activation(object, activation) Apply an activation function to an output

layer_activation_leaky_relu() Leaky version of a rectified linear unit

layer_activation_parametric_relu() Parametric rectified linear unit

layer_activation_thresholded_relu() Thresholded rectified linear unit

layer_activation_elu() Exponential linear unit

DROPOUT LAYERS

layer_dropout() Applies dropout to the input

layer_spatial_dropout_1d()
layer_spatial_dropout_2d()
layer_spatial_dropout_3d()
Spatial 1D to 3D version of dropout

RECURRENT LAYERS

layer_simple_rnn() Fully-connected RNN where the output is to be fed back to input

layer_gru() Gated recurrent unit - Cho et al

layer_cudnn_gru() Fast GRU implementation backed by CuDNN

layer_lstm() Long-Short Term Memory unit - Hochreiter 1997

layer_cudnn_lstm() Fast LSTM implementation backed by CuDNN

LOCALLY CONNECTED LAYERS

layer_locally_connected_1d()
layer_locally_connected_2d()
Similar to convolution, but weights are not shared, i.e. different filters for each patch

Preprocessing

SEQUENCE PREPROCESSING

pad_sequences() Pads each sequence to the same length (length of the longest sequence)

skipgrams() Generates skipgram word pairs

make_sampling_table() Generates word rank-based probabilistic sampling table

TEXT PREPROCESSING

text_tokenizer() Text tokenization utility

fit_text_tokenizer() Update tokenizer internal vocabulary

save_text_tokenizer(); load_text_tokenizer() Save a text tokenizer to an external file

texts_to_sequences();
texts_to_sequences_generator()
Transforms each text in texts to sequence of integers

texts_to_matrix(); sequences_to_matrix() Convert a list of sequences into a matrix

text_one_hot() One-hot encode text to word indices

text_hashing_trick() Converts a text to a sequence of indexes in a fixed-size hashing space

text_to_word_sequence() Convert text to a sequence of words (or tokens)

IMAGE PREPROCESSING

image_load() Loads an image into PIL format.

flow_images_from_data()
flow_images_from_directory()
Generates batches of augmented/normalized data from images and labels, or a directory

image_data_generator() Generate minibatches of image data with real-time data augmentation.

fit_image_data_generator() Fit image data generator internal statistics to some sample data

generator_next() Retrieve the next item

image_to_array(); image_array_resize()
image_array_save() 3D array representation

Pre-trained models

Keras applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

application_xception()
xception_preprocess_input()
Xception v1 model

application_inception_v3()
inception_v3_preprocess_input()
Inception v3 model, with weights pre-trained on ImageNet

application_inception_resnet_v2()
inception_resnet_v2_preprocess_input()
Inception-ResNet v2 model, with weights trained on ImageNet

application_vgg16(); application_vgg19() VGG16 and VGG19 models

application_resnet50() ResNet50 model

application_mobilenet()
mobilenet_preprocess_input()
mobilenet_decode_predictions()
mobilenet_load_model_hdf5()
MobileNet model architecture

IMAGENET

ImageNet is a large database of images with labels, extensively used for deep learning

imagenet_preprocess_input()
imagenet_decode_predictions()
Preprocesses a tensor encoding a batch of images for ImageNet, and decodes predictions

Callbacks

A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training.

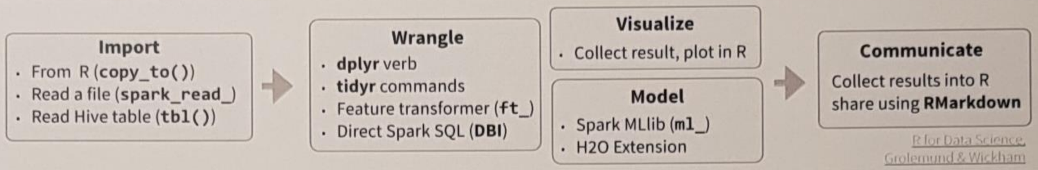
callback_early_stopping() Stop training when a monitored quantity has stopped improving
callback_learning_rate_scheduler() Learning rate scheduler
callback_tensorboard() TensorBoard basic visualizations

R Studio

Data Science in Spark with *sparklyr* : : **CHEAT SHEET**

Intro

sparklyr is an R interface for Apache Spark™. It enables us to write all of our analysis code in R, but have the actual processing happen inside Spark clusters. Easily manipulate and model large-scale using R and Spark via *sparklyr*.



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

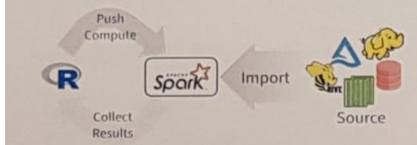
Model
- Spark MLlib (**`ml_`**)
- H2O Extension

Communicate

Collect results into R share using **RMarkdown**

R for Data Science, Grolemund & Wickham

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()**
ORC	**spark_read_orc()**
LIBSVM	**spark_read_libsvm()**
DELTA	**spark_read_delta()**
AVRO	**spark_read_avro()**

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

ARROW
```
library(sparklyr)
library(arrow)
```

FROM A TABLE IN HIVE

dplyr::**tbl(**scr, …**)** - Creates a reference to the table without loading it into memory

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)

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

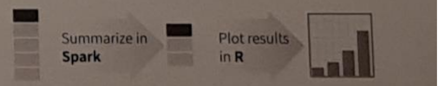
ft_vector_assembler() - Combine vectors into single row-vector

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) %>%          Summarize in Spark
  group_by(cyl) %>%
  summarise(mpg_m = mean(mpg)) %>%
  collect() %>%                  Collect results in R
  ggplot() +
    geom_col(aes(cyl, mpg_m))    Create plot
```

RStudio

Learn more at **spark.rstudio.com** • sparklyr 1.7.5 • Updated: 2022-02

Data Science in Spark with *sparklyr* : : CHEAT SHEET

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() - Currently implemented using parallelized pool adjacent violators algorithm. Only univariate (single feature) algorithm supported

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() - Classification model 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 Classification to Binary Classification. 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.

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

STATS

ml_summary() - Extracts a metric from the summary object of a Spark ML model

ml_corr() - Compute correlation matrix

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.

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, to be used in `ml_model` constructor

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. The saved model 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

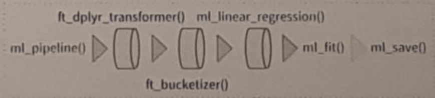
ml_save() - Saves into a format that can be read by Scala and PySpark .

ml_read() - Reads Spark object into sparklyr.

SQL AND DPLYR

ft_sql_transformer() - Creates a Pipeline step based on the SQL statement passed to the command.

ft_dplyr_transformer() - Creates a Pipeline step based on one or several dplyr commands.



spark.rstudio.com/guides/pipelines

More Info

spark.rstudio.com   therinspark.com

Sessions

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")
```

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 = "2.0.1",
   spark_home = [path to Spark])
```

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")
```

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"))
```

CLOUD

Databricks - `spark_connect(method = "databricks")`

Qubole- `spark_connect(method = "qubole")`

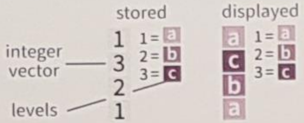
RStudio

Factors with forcats : : CHEAT SHEET

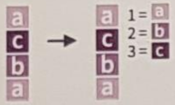
The **forcats** package provides tools for working with factors, which are R's data structure for categorical data.

Factors

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

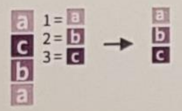


Create a factor with factor()



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

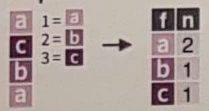
Return its levels with levels()



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

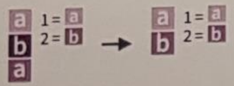
Use unclass() to see its structure

Inspect Factors



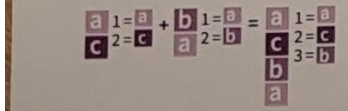
fct_count(f, sort = FALSE, prop = FALSE) Count the number of values with each level. fct_count(f)

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

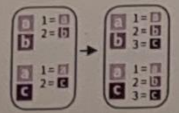


fct_unique(f) Return the unique values, removing duplicates. fct_unique(f)

Combine Factors

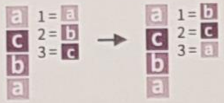


fct_c(…) Combine factors with different levels. Also **fct_cross()**.
f1 <- factor(c("a", "c"))
f2 <- factor(c("b", "a"))
fct_c(f1, f2)

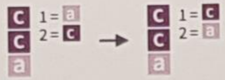


fct_unify(fs, levels = lvls_union(fs)) Standardize levels across a list of factors. fct_unify(list(f2, f1))

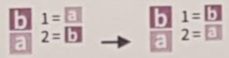
Change the order of levels



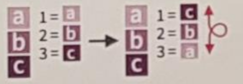
fct_relevel(.f, ..., after = 0L) Manually reorder factor levels. fct_relevel(f, c("b", "c", "a"))



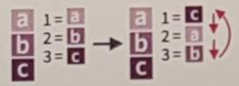
fct_infreq(f, ordered = NA) Reorder levels by the frequency in which they appear in the data (highest frequency first). Also **fct_inseq()**.
f3 <- factor(c("c", "c", "a"))
fct_infreq(f3)



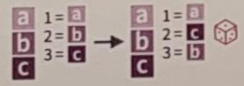
fct_inorder(f, ordered = NA) Reorder levels by order in which they appear in the data. fct_inorder(f2)



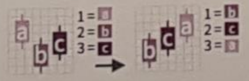
fct_rev(f) Reverse level order.
f4 <- factor(c("a","b","c"))
fct_rev(f4)



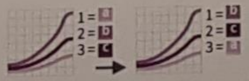
fct_shift(f) Shift levels to left or right, wrapping around end. fct_shift(f4)



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

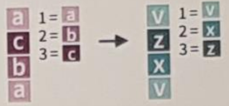


fct_reorder(.f, .x, .fun = median, ..., .desc = FALSE) Reorder levels by their relationship with another variable.
boxplot(data = PlantGrowth, weight ~ reorder(group, weight))

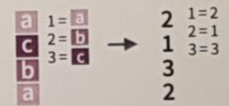


fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE) Reorder levels by their final values when plotted with two other variables.
ggplot(diamonds,aes(carat, price, color = fct_reorder2(color, carat, price))) + geom_smooth()

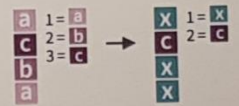
Change the value of levels



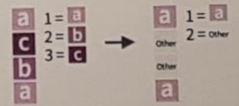
fct_recode(.f, ...) Manually change levels. Also **fct_relabel()** which obeys purrr::map syntax to apply a function or expression to each level.
fct_recode(f, v = "a", x = "b", z = "c")
fct_relabel(f, ~ paste0("x", .x))



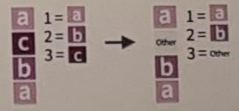
fct_anon(f, prefix = "") Anonymize levels with random integers. fct_anon(f)



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

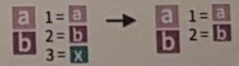


fct_lump_min(f, min, w = NULL, other_level = "Other") Lumps together factors that appear fewer than min times. Also **fct_lump_n()**, **fct_lump_prop()**, and **fct_lump_lowfreq()**. fct_lump_min(f, min = 2)

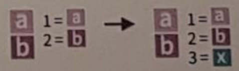


fct_other(f, keep, drop, other_level = "Other") Replace levels with "other." fct_other(f, keep = c("a", "b"))

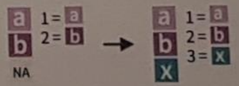
Add or drop levels



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



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



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