



Orange County R Users Group (OCRUG)

2020-04 Hackathon



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
Choose output location
Insert code chunk

```
1 # Good Start...
2 Cursors of shared users
3 Re-run previous code
4 Source with or without Echo
5 Show file outline
6 Multiple cursors/column selection with Alt + mouse drag.
7 Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.
8 Syntax highlighting based on your file's extension
9 Tab completion to finish function names, file paths, arguments, and more.
10 Multi-language code snippets to quickly use common blocks of code.
11 Displays saved objects by type with short description
12 View in data viewer
13 View function source code
14 Jump to function in file
15 Change file type
16 Maximize, minimize panes
17 Drag pane boundaries
18 Press ↑ to see command history
19 Working Directory
20 Path to displayed directory
21 A File browser keyed to your working directory. Click on file or directory name to open.
```

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

```
1 Run app
2 Choose location to view app
3 Publish to shinyapps.io or server
4 Manage publish accounts
5
```

Write Code

Navigate tabs
Open in new window
Save
Find and replace
Compile as notebook
Run selected code

```
1 # Good Start...
2 Cursors of shared users
3 Re-run previous code
4 Source with or without Echo
5 Show file outline
6 Multiple cursors/column selection with Alt + mouse drag.
7 Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.
8 Syntax highlighting based on your file's extension
9 Tab completion to finish function names, file paths, arguments, and more.
10 Multi-language code snippets to quickly use common blocks of code.
11 Displays saved objects by type with short description
12 View in data viewer
13 View function source code
14 Jump to function in file
15 Change file type
16 Maximize, minimize panes
17 Drag pane boundaries
18 Press ↑ to see command history
19 Working Directory
20 Path to displayed directory
21 A File browser keyed to your working directory. Click on file or directory name to open.
```

R Support

Import data with wizard
History of past commands to run/copy
Display.RPres slideshows
File > New File > R Presentation

```
1 # Good Start...
2 Cursors of shared users
3 Re-run previous code
4 Source with or without Echo
5 Show file outline
6 Multiple cursors/column selection with Alt + mouse drag.
7 Code diagnostics that appear in the margin. Hover over diagnostic symbols for details.
8 Syntax highlighting based on your file's extension
9 Tab completion to finish function names, file paths, arguments, and more.
10 Multi-language code snippets to quickly use common blocks of code.
11 Displays saved objects by type with short description
12 View in data viewer
13 View function source code
14 Jump to function in file
15 Change file type
16 Maximize, minimize panes
17 Drag pane boundaries
18 Press ↑ to see command history
19 Working Directory
20 Path to displayed directory
21 A File browser keyed to your working directory. Click on file or directory name to open.
```

Pro Features

Share Project Active shared with Collaborators
Start new R Session in current project
Close R Session in project
Select R Version
PROJECT SYSTEM
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.

RStudio opens plots in a dedicated Plots pane
Plots Navigate in recent plots
Open in recent window
Export plot Delete plot
Delete all plots
GUI Package manager lists every installed package
Files Install Packages
Update Packages
Create reproducible package library for your project
Click to load package with **library()**. Unclick to detach package with **detach()**
Packages Package version installed
Delete from library

RStudio opens documentation in a dedicated Help pane
Help 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
Viewer Stop Shiny app
Publish to shinyapps.io, rpubs, RSConnect, ...

View(<data>) opens spreadsheet like view of data set
Environment Filter
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 All 5.1 3.5 1.4 0.2 setosa
2
3 Filter rows by value or value range
Sort by values
Search for value

Debug Mode

Open with **debug()**, **browser()**, or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

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

Launch debugger mode from origin of error

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

Error in get_digit(num, x) : Show Traceback Error!

Step through code one line at a time

Step into and out of functions to run

Resume execution mode

Quit debug

Version Control

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

+ Stage files:
Show file diff
Commit staged files
Push/Pull to remote
View History
Added
Deleted
Modified
Renamed
Untracked
Environment History Git
file-with-changes.R
Open shell to type commands
master
current branch

Package Writing

File > New Project > New Directory > R Package

Turn project into package, Enable roxygen documentation with **Tools > Project Options > Build Tools**

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

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

Environment Filter
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 All 5.1 3.5 1.4 0.2 setosa
2
3 Filter rows by value or value range
Sort by values
Search for value



1 LAYOUT

Move focus to Source Editor
Move focus to Console
Move focus to Help
Show History
Show Files
Show Plots
Show Packages
Show Environment
Show Git/SVN
Show Build

Windows/Linux Mac

Ctrl+1	Ctrl+1
Ctrl+2	Ctrl+2
Ctrl+3	Ctrl+3
Ctrl+4	Ctrl+4
Ctrl+5	Ctrl+5
Ctrl+6	Ctrl+6
Ctrl+7	Ctrl+7
Ctrl+8	Ctrl+8
Ctrl+9	Ctrl+9
Ctrl+0	Ctrl+0

4 WRITE CODE

Attempt completion
Navigate candidates
Accept candidate
Dismiss candidates
Undo
Redo
Cut
Copy
Paste
Select All
Delete Line

Windows /Linux

Tab or Ctrl+Space	↑/↓
	Enter, Tab, or →
Esc	
Ctrl+Z	
Ctrl+Shift+Z	
Ctrl+X	
Ctrl+C	
Ctrl+V	
Ctrl+A	
Ctrl+D	
Select	Shift+[Arrow]
Select Word	Ctrl+Shift+←/→
Select to Line Start	Alt+Shift+←
Select to Line End	Alt+Shift+→
Select Page Up/Down	Shift+PageUp/Down
Select to Start/End	Shift+Alt+↑/↓
Delete Word Left	Ctrl+Backspace

Mac

Tab or Cmd+Space	↑/↓
	Enter, Tab, or →
Esc	
Cmd+Z	
Cmd+Shift+Z	
Cmd+X	
Cmd+C	
Cmd+V	
Cmd+A	
Cmd+D	
Select	Shift+[Arrow]
Select Word	Option+Shift+←/→
Select to Line Start	Cmd+Shift+←
Select to Line End	Cmd+Shift+→
Select Page Up/Down	Shift+PageUp/Down
Select to Start/End	Shift+Alt+↑/↓
Delete Word Left	Ctrl+Opt+Backspace

WHY RSTUDIO SERVER PRO?

RSP extends 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
- edit the same project at the same time as others
- see what you and others are doing on your server
- switch easily from one version of R to a different version
- integrate with your authentication, authorization, and audit practices

Download a free 45 day evaluation at

www.rstudio.com/products/rstudio-server-pro/

2 RUN CODE

Search command history
Navigate command history
Move cursor to start of line
Move cursor to end of line
Change working directory

Interrupt current command
Clear console

Quit Session (desktop only)

Restart R Session

Run current line/selection

Run current (retain cursor)

Run from current to end

Run the current function

Source a file

Source the current file

Source with echo

Windows/Linux Mac

Ctrl+↑	Cmd+↑
↑/↓	↑/↓
Home	Cmd+←
End	Cmd+→
Ctrl+Shift+H	Ctrl+Shift+H
Esc	Esc
Ctrl+L	Ctrl+L
Ctrl+Q	Cmd+Q
Ctrl+Shift+F10	Cmd+Shift+F10
Ctrl+Enter	Cmd+Enter
Alt+Enter	Option+Enter
Ctrl+Alt+E	Cmd+Option+E
Ctrl+Alt+F	Cmd+Option+F
Ctrl+Alt+G	Cmd+Option+G
Ctrl+Shift+S	Cmd+Shift+S
Ctrl+Shift+Enter	Cmd+Shift+Enter

3 NAVIGATE CODE

Goto File/Function

Fold Selected

Unfold Selected

Fold All

Unfold All

Go to line

Jump to

Switch to tab

Previous tab

Next tab

First tab

Last tab

Navigate back

Navigate forward

Jump to Brace

Select within Braces

Use Selection for Find

Find in Files

Find Next

Find Previous

Jump to Word

Jump to Start/End

Toggle Outline

Windows /Linux

Mac

Ctrl+.	Ctrl+.
Alt+L	Cmd+Option+L
Shift+Alt+L	Cmd+Shift+Option+L
Alt+O	Cmd+Option+O
Shift+Alt+O	Cmd+Shift+Option+O
Shift+Alt+G	Cmd+Shift+Option+G
Shift+Alt+J	Cmd+Shift+Option+J
Ctrl+Shift+.	Ctrl+Shift+.
Ctrl+F11	Ctrl+F11
Ctrl+F12	Ctrl+F12
Ctrl+Shift+F11	Ctrl+Shift+F11
Ctrl+Shift+F12	Ctrl+Shift+F12
Ctrl+F9	Cmd+F9
Ctrl+F10	Cmd+F10
Ctrl+P	Ctrl+P
Ctrl+Shift+Alt+E	Ctrl+Shift+Option+E
Ctrl+F3	Cmd+E
Ctrl+Shift+F	Cmd+Shift+F
Win: F3, Linux: Ctrl+G	Cmd+G
W: Shift+F3, L:	Cmd+Shift+G
Option+←/→	Option+←/→
Ctrl+↑/↓	Ctrl+↑/↓
Ctrl+Shift+O	Cmd+Shift+O

4 WRITE CODE

Attempt completion
Navigate candidates
Accept candidate
Dismiss candidates
Undo
Redo
Cut
Copy
Paste
Select All
Delete Line

Windows /Linux

Tab or Ctrl+Space	↑/↓
	Enter, Tab, or →
Esc	
Ctrl+Z	
Ctrl+Shift+Z	
Ctrl+X	
Ctrl+C	
Ctrl+V	
Ctrl+A	
Ctrl+D	
Select	Shift+[Arrow]
Select Word	Ctrl+Shift+←/→
Select to Line Start	Alt+Shift+←
Select to Line End	Alt+Shift+→
Select Page Up/Down	Shift+PageUp/Down
Select to Start/End	Shift+Alt+↑/↓
Delete Word Left	Ctrl+Backspace

Mac

Tab or Cmd+Space	↑/↓
	Enter, Tab, or →
Esc	
Cmd+Z	
Cmd+Shift+Z	
Cmd+X	
Cmd+C	
Cmd+V	
Cmd+A	
Cmd+D	
Select	Shift+[Arrow]
Select Word	Option+Shift+←/→
Select to Line Start	Cmd+Shift+←
Select to Line End	Cmd+Shift+→
Select Page Up/Down	Shift+PageUp/Down
Select to Start/End	Shift+Alt+↑/↓
Delete Word Left	Ctrl+Opt+Backspace

5 DEBUG CODE

Toggle Breakpoint
Execute Next Line
Step Into Function
Finish Function/Loop
Continue
Stop Debugging

Windows/Linux Mac

Shift+F9	Shift+F9
F10	F10
Shift+F4	Shift+F4
Shift+F6	Shift+F6
Shift+F5	Shift+F5
Shift+F8	Shift+F8

6 VERSION CONTROL

Show diff
Commit changes
Scroll diff view
Stage/Unstage (Git)
Stage/Unstage and move to next

Windows/Linux Mac

Ctrl+Alt+D	Ctrl+Option+D
Ctrl+Alt+M	Ctrl+Option+M
Ctrl+↑/↓	Ctrl+↑/↓
Spacebar	Spacebar
Enter	Enter

7 MAKE PACKAGES

Build and Reload
Load All (devtools)
Test Package (Desktop)
Test Package (Web)
Check Package
Document Package

Windows/Linux Mac

Ctrl+Shift+B	Cmd+Shift+B
Ctrl+Shift+L	Cmd+Shift+L
Ctrl+Shift+T	Cmd+Shift+T
Ctrl+Alt+F7	Cmd+Opt+F7
Ctrl+Shift+E	Cmd+Shift+E
Ctrl+Shift+D	Cmd+Shift+D

8 DOCUMENTS AND APPS

Preview HTML (Markdown, etc.)
Knit Document (knitr)
Compile Notebook
Compile PDF (TeX and Sweave)
Insert chunk (Sweave and Knitr)
Insert code section
Re-run previous region
Run current document
Run from start to current line
Run the current code section
Run previous Sweave/Rmd code
Run the current chunk
Run the next chunk
Sync Editor & PDF Preview
Previous plot
Next plot
Show Keyboard Shortcuts
Ctrl+Shift+K
Ctrl+Shift+K
Ctrl+Shift+K
Ctrl+Shift+K
Ctrl+Alt+I
Ctrl+Shift+R
Ctrl+Shift+P
Cmd+Shift+P
Ctrl+Alt+R
Ctrl+Alt+R
Ctrl+Alt+R
Ctrl+Alt+B
Ctrl+Alt+T
Ctrl+Alt+P
Cmd+Option+P
Ctrl+Alt+C
Cmd+Option+C
Ctrl+Alt+N
Cmd+Option+N
Ctrl+F8
Cmd+F8
Ctrl+Alt+F11
Ctrl+Alt+F12
Alt+Shift+K
Option+Shift+K

Shiny :: CHEAT SHEET

Basics

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).

APP TEMPLATE

Begin writing a new app with this template. Preview the app by running the code at the R command line.

```
library(shiny)
ui <- fluidPage()
server <- function(input, output){}
shinyApp(ui = ui, server = server)
```

- **ui** - nested R functions that assemble an HTML user interface for your app
- **server** - a function with instructions on how to build and rebuild the R objects displayed in the UI
- **shinyApp** - combines **ui** and **server** into an app. Wrap with **runApp()** if calling from a sourced script or inside a function.

SHARE YOUR APP

 The easiest way to share your app is to host it on shinyapps.io, a cloud based service from RStudio

1. Create a free or professional account at <http://shinyapps.io>
2. Click the **Publish** icon in the RStudio IDE or run:
`rsconnect::deployApp("<path to directory>")`

Build or purchase your own Shiny Server at www.rstudio.com/products/shiny-server/



Building an App

Add inputs to the UI with `*Input()` functions

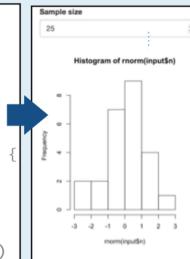
Add outputs with `*Output()` functions

Tell server how to render outputs with R in the server function. To do this:

1. Refer to outputs with `output$<id>`
2. Refer to inputs with `input$<id>`
3. Wrap code in a `render*`() function before saving to output

Complete the template by adding arguments to `fluidPage()` and a body to the server function.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```



Save your template as **app.R**. Alternatively, split your template into two files named **ui.R** and **server.R**.

```
library(shiny)
ui <- fluidPage(
  numericInput(inputId = "n",
    "Sample size", value = 25),
  plotOutput(outputId = "hist")
)
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}
shinyApp(ui = ui, server = server)
```

ui.R contains everything you would save to ui.

server.R ends with the function you would save to server.

No need to call **shinyApp()**.

Save each app as a directory that holds an **app.R** file (or a **server.R** file and a **ui.R** file) plus optional extra files.



The directory name is the name of the app
(optional) defines objects available to both ui.R and server.R
(optional) used in showcase mode
(optional) data, scripts, etc.
(optional) directory of files to share with web browsers (images, CSS, js, etc.) Must be named "www"

Launch apps with
`runApp(<path to directory>)`

Outputs - `render*`() and `*Output()` functions work together to add R output to the UI



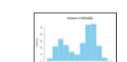
`DT::renderDataTable(expr, options, callback, escape, env, quoted)`

works with

`dataTableOutput(outputId, icon, ...)`



`renderImage(expr, env, quoted, deleteFile)`



`renderPlot(expr, width, height, res, ..., env, quoted, func)`



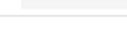
`renderPrint(expr, env, quoted, func, width)`



`renderTable(expr, ..., env, quoted, func)`



`renderText(expr, env, quoted, func)`



`renderUI(expr, env, quoted, func)`

&

`tableOutput(outputId)`

`textOutput(outputId, container, inline)`

`uiOutput(outputId, inline, container, ...)`

`htmlOutput(outputId, inline, container, ...)`



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, ...)`

Link

`actionLink(inputId, label, icon, ...)`

Choice 1

Choice 2

Choice 3

Check me

`checkboxGroupInput(inputId, label, choices, selected, inline)`

`checkboxInput(inputId, label, value)`

`dateInput(inputId, label, value, min, max, format, startview, weekstart, language)`

`dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator)`

`fileInput(inputId, label, multiple, accept)`

1

`numericInput(inputId, label, value, min, max, step)`

.....

`passwordInput(inputId, label, value)`

Choice A

Choice B

Choice C

Choice 1

Choice 2

8

10

12

14

16

18

20

22

24

26

28

30

32

34

36

38

40

Apply Changes

Enter text

`radioButtons(inputId, label, choices, selected, inline)`

`selectInput(inputId, label, choices, selected, multiple, selectize, width, size) (also selectizeInput())`

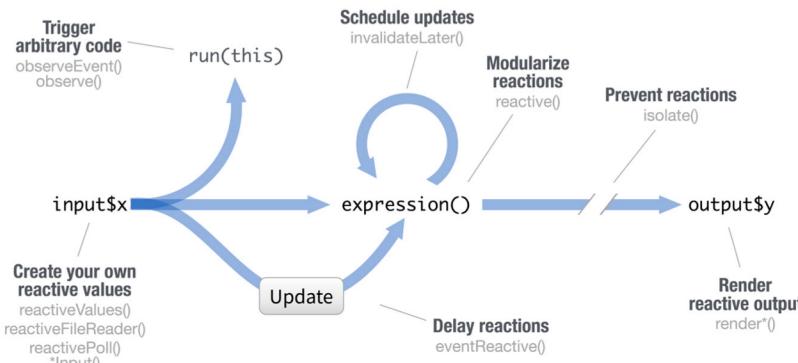
`sliderInput(inputId, label, min, max, value, step, round, format, locale, ticks, animate, width, sep, pre, post)`

`submitButton(text, icon) (Prevents reactions across entire app)`

`textInput(inputId, label, value)`

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

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

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

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

Each input function creates a reactive value stored as **input\$<inputId>**
reactiveValues() creates a list of reactive values whose values you can set.

PREVENT REACTIONS

```
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.

TRIGGER ARBITRARY CODE

```
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)

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

MODULARIZE REACTIONS

```
ui <- fluidPage(
  textInput("x","","A"),
  textInput("z","","Z"),
  textOutput("b"))

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

reactive(x, env, quoted, label, domain)
Creates a **reactive expression** that

- caches its value to reduce computation
- can be called by other code
- notifies its dependencies when it has been invalidated

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

DELAY REACTIONS

```
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)

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

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

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

```
fluidPage(
  textInput("a","",)
)
## <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>
```

Returns 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.

tags\$a	tags\$data	tags\$h6	tags\$nav	tags\$span
tags\$abbr	tags\$dataList	tags\$head	tags\$noscript	tags\$strong
tags\$address	tags\$dd	tags\$header	tags\$object	tags\$style
tags\$area	tags\$del	tags\$group	tags\$ol	tags\$sub
tags\$article	tags\$details	tags\$hr	tags\$optgroup	tags\$summary
tags\$aside	tags\$dfn	tags\$HTML	tags\$option	tags\$sup
tags\$audio	tags\$div	tags\$output	tags\$stable	
tags\$b	tags\$dl	tags\$iframe	tags\$p	tags\$tbody
tags\$base	tags\$dt	tags\$img	tags\$pre	tags\$td
tags\$bdy	tags\$em	tags\$input	tags\$progress	tags\$tfoot
tags\$blockquote	tags\$embed	tags\$ins	tags\$script	tags\$th
tags\$body	tags\$eventsSource	tags\$kbd	tags\$style	tags\$thead
tags\$fieldset	tags\$fieldset	tags\$q	tags\$tbody	
tags\$br	tags\$keygen	tags\$ruby	tags\$thead	
tags\$button	tags\$caption	tags\$rp	tags\$time	
tags\$canvas	tags\$figure	tags\$legend	tags\$title	
tags\$caption	tags\$footer	tags\$sr	tags\$tbody	
tags\$cite	tags\$form	tags\$sli	tags\$tr	
tags\$code	tags\$h1	tags\$link	tags\$tbody	
tags\$col	tags\$h2	tags\$mark	tags\$tr	
tags\$colgroup	tags\$h3	tags\$map	tags\$tbody	
tags\$command	tags\$h4	tags\$menu	tags\$tr	
tags\$h5	tags\$li	tags\$meta	tags\$tbody	
	tags\$ol	tags\$source	tags\$tbody	
	tags\$ul	tags\$tbody		

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>"))
)
```



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>"))
```



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>"))
```

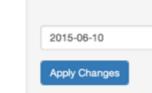


To include an image 1. Place the file in the **www** subdirectory 2. Link to it with **img(src=<file name>)**

Layouts

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

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



absolutePanel()
conditionalPanel()
fixedPanel()
headerPanel()
inputPanel()
mainPanel()

navlistPanel()
sidebarPanel()
tabPanel()
tabpanel()
titlePanel()
wellPanel()

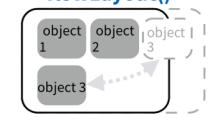
Organize panels and elements into a layout with a layout function. Add elements as arguments of the layout functions.

fluidRow()



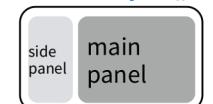
```
ui <- fluidPage(
  fluidRow(column(width = 4),
    column(width = 2, offset = 3)),
  fluidRow(column(width = 12))
)
```

flowLayout()



```
ui <- fluidPage(
  flowLayout(object1,
  object2,
  object3
)
)
```

sidebarLayout()



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

splitLayout()



```
ui <- fluidPage(
  splitLayout(object1,
  object2
)
)
```

verticalLayout()



```
ui <- fluidPage(
  verticalLayout(object1,
  object2,
  object3
)
)
```



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"))
)
```

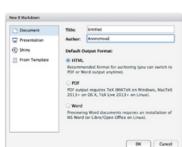


R Markdown :: CHEAT SHEET

What is R Markdown?

- .Rmd files** • An R Markdown (.Rmd) file is a record of your research. It contains the code that a scientist needs to reproduce your work along with the narration that a reader needs to understand your work.
- Reproducible Research** • At the click of a button, or the type of a command, you can rerun the code in an R Markdown file to reproduce your work and export the results as a finished report.
- Dynamic Documents** • You can choose to export the finished report in a variety of formats, including html, pdf, MS Word, or RTF documents; html or pdf based slides, Notebooks, and more.

Workflow



- ① Open a new .Rmd file at File ▶ New File ▶ R Markdown. Use the wizard that opens to pre-populate the file with a template
- ② Write document by editing template
- ③ Knit document to create report; use knit button or render() to knit
- ④ Preview Output in IDE window
- ⑤ Publish (optional) to web server
- ⑥ Examine build log in R Markdown console
- ⑦ Use output file that is saved along side .Rmd

The screenshot shows the RStudio interface with the R Markdown file 'report.Rmd' open. The file contains R code for generating a summary of the 'cars' dataset. The RStudio interface includes a toolbar with various buttons like 'set preview location', 'insert code chunk', 'run code chunk(s)', 'publish', etc. The R Markdown preview window shows the rendered HTML output with a table of car data. The R Markdown console shows the command 'render("report.Rmd", output_file = "report.html")' being run, and the resulting file 'report.html' is listed in the file browser.

Embed code with knitr syntax

INLINE CODE

Insert with `r <code>` . Results appear as text without code.

Built with `r getRVersion()` → Built with 3.2.3

CODE CHUNKS

One or more lines surrounded with `{{r}}` and `{{ }}` . Place chunk options within curly braces, after r. Insert with ↗

```
```{r echo=TRUE}
getRVersion()
```
getRVersion()
## [1] '3.2.3'
```

GLOBAL OPTIONS

Set with knitr::opts_chunk\$set(), e.g.

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

IMPORTANT CHUNK OPTIONS

cache - cache results for future knits (default = FALSE)
cache.path - directory to save cached results in (default = "cache/")
child - file(s) to knit and then include (default = NULL)
collapse - collapse all output into single block (default = FALSE)
comment - prefix for each line of results (default = '##')

dependson - chunk dependencies for caching (default = NULL)
echo - Display code in output document (default = TRUE)
engine - code language used in chunk (default = 'R')
error - Display error messages in doc (TRUE) or stop render when errors occur (FALSE) (default = FALSE)
eval - Run code in chunk (default = TRUE)

Options not listed above: R.options, aniopts, autodep, background, cache.comments, cache.lazy, cache.rebuild, cache.vars, dev, dev.args, dpi, engine.opts, engine.path, fig.asp, fig.env, fig.ext, fig.keep, fig.lp, fig.path, fig.pos, fig.process, fig.retina, fig.scap, fig.show, fig.showtext, fig.subcap, interval, out.extra, out.height, out.width, prompt, purl, reflabel, render, size, split, tidy.opts

fig.align - 'left', 'right', or 'center' (default = 'default')
fig.cap - figure caption as character string (default = NULL)
fig.height, fig.width - Dimensions of plots in inches
highlight - highlight source code (default = TRUE)
include - Include chunk in doc after running (default = TRUE)

message - display code messages in document (default = TRUE)
results (default = 'markup')
'asis' - passthrough results
'hide' - do not display results
'hold' - put all results below all code
tidy - tidy code for display (default = FALSE)
warning - display code warnings in document (default = TRUE)

.rmd Structure

rmarkdown

YAML Header
Optional section of render (e.g. pandoc) options written as key:value pairs (YAML).
At start of file
Between lines of ---
Text
Narration formatted with markdown, mixed with:
Code Chunks
Chunks of embedded code. Each chunk:
Begins with `{{r}}`
ends with `{{ }}`
R Markdown will run the code and append the results to the doc.
It will use the location of the .Rmd file as the **working directory**

Parameters

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

1. **Add parameters** • Create and set parameters in the header as sub-values of params
2. **Call parameters** • Call parameter values in code as params\$<name>
3. **Set parameters** • Set values with Knit with parameters or the params argument of render():
render("doc.Rmd", params = list(n = 1, d = as.Date("2015-01-01")))

Interactive Documents

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 markdown::run or click Run Document in RStudio IDE

The screenshot shows a Shiny application interface. On the left, there is a text input field with the value '5'. On the right, there is a table with columns 'speed' and 'dist'. The table has five rows with data: (4.00, 2.00), (4.00, 10.00), (7.00, 4.00), (7.00, 22.00), and (8.00, 16.00). A blue arrow points from the text input field to the table, indicating the flow of data from the user input to the reactive output.

Embed a complete app into your document with shiny::shinyAppDir()
NOTE: Your report will be rendered as a Shiny app, which means you must choose an html output format, like **html_document**, and serve it with an active R Session.

Data Import :: CHEAT SHEET



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



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



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

OTHER TYPES OF DATA

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

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

Save Data

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

Comma delimited file

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

File with arbitrary delimiter

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

CSV for excel

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

String to file

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

String vector to file, one element per line

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

Object to RDS file

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

Tab delimited files

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



Read Tabular Data

 - These functions share the common arguments:

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

| a,b,c | A | B | C |
|--------|---|---|----|
| 1,2,3 | 1 | 2 | 3 |
| 4,5,NA | 4 | 5 | NA |

| a;b;c | A | B | C |
|--------|---|---|----|
| 1;2;3 | 1 | 2 | 3 |
| 4;5;NA | 4 | 5 | NA |

| a b c | A | B | C |
|--------|---|---|----|
| 1 2 3 | 1 | 2 | 3 |
| 4 5 NA | 4 | 5 | NA |

| a b c | A | B | C |
|--------|---|---|----|
| 1 2 3 | 1 | 2 | 3 |
| 4 5 NA | 4 | 5 | NA |

Comma Delimited Files

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

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

Semi-colon Delimited Files

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

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

Files with Any Delimiter

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

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

Fixed Width Files

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

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

Tab Delimited Files

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

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

USEFUL ARGUMENTS

| a,b,c | 1 | 2 | 3 |
|--------|---|---|----|
| 1,2,3 | 4 | 5 | NA |
| 4,5,NA | | | |

Example file

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

| A | B | C |
|---|---|----|
| 1 | 2 | 3 |
| 4 | 5 | NA |

No header

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

| x | y | z |
|---|---|---|
| A | B | C |
| 1 | 2 | 3 |

Provide header

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

| 1 | 2 | 3 |
|---|---|----|
| 4 | 5 | NA |
| | | |

Skip lines

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

| A | B | C |
|---|---|----|
| 1 | 2 | 3 |
| 4 | 5 | NA |

Read in a subset

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

| A | B | C |
|----|---|----|
| NA | 2 | 3 |
| 4 | 5 | NA |

Missing Values

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

Read Non-Tabular Data

Read a file into a single string

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

Read each line into its own string

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

Read Apache style log files

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

Read a file into a raw vector

```
read_file_raw(file)
```

Read each line into a raw vector

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

Data types

readr functions guess the types of each column and convert types when appropriate (but will NOT convert strings to factors automatically). A message shows the type of each column in the result.

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

age is an integer
sex is a character
earn is a double (numeric)

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

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

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

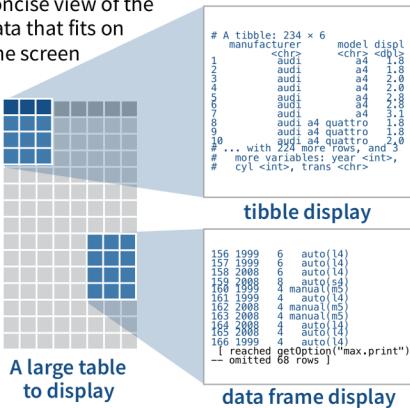
3. Else, read in as character vectors then parse with a **parse_** function.

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

Tibbles - an enhanced data frame

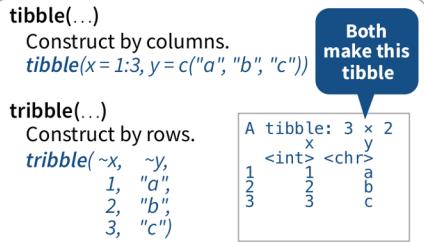
The **tibble** package provides a new S3 class for storing tabular data, the **tibble**. Tibbles inherit the data frame class, but improve three behaviors:

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



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

CONSTRUCT A TIBBLE IN TWO WAYS



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

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

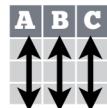
`is_tibble(x)` Test whether x is a tibble.

R Studio

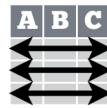
Tidy Data with `tidyr`

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

A table is tidy if:



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

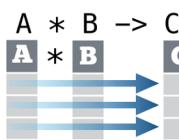


Each **observation**, or **case**, is in its own **row**

Tidy data:



Makes variables easy to access as vectors



Preserves cases during vectorized operations

Reshape Data - change the layout of values in a table

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

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

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

table4a

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

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

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 |

key

value

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

key

value

`spread(table2, type, count)`

Handle Missing Values

fill(data, ...)

Fill in NA's in ... columns with most recent non-NA values.

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

`drop_na(x, x2)`

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

`fill(x, x2)`

replace_na(data, replace = list(...))

Replace NA's by column.

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

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

Expand Tables - quickly create tables with combinations of values

complete(data, ...)

Adds to the data missing combinations of the values of the variables listed in ...

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

expand(data, ...)

Create new tibble with all possible combinations of the values of the variables listed in ...

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

Split Cells

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



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

Separate each cell in a column to make several columns.

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

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

separate_rows(data, ...)

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

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

`separate_rows(table3, rate)`

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

Collapse cells across several columns to make a single column.

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

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

Data Transformation with dplyr :: CHEAT SHEET



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



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



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



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

Summarise Cases

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

summary function



summarise(.data, ...)

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



count(x, ..., wt = NULL, sort = FALSE)

Count number of rows in each group defined by the variables in ... Also **tally()**.
`count(iris, Species)`

VARIATIONS

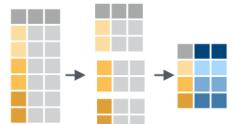
summarise_all() - Apply funs to every column.

summarise_at() - Apply funs to specific columns.

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

Group Cases

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



`mtcars %>%`

`group_by(cyl) %>%`

`summarise(avg = mean(mpg))`

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

ungroup(x, ...)

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



Manipulate Cases

EXTRACT CASES

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



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



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



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



sample_n(tbl, size, replace = FALSE, weight = NULL, .env = parent.frame()) Randomly select size rows.
`sample_n(iris, 10, replace = TRUE)`



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



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

Manipulate Variables

EXTRACT VARIABLES

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



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



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

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

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

MAKE NEW VARIABLES

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

vectorized function



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



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



mutate_all(.tbl, .funs, ...) Apply funs to every column. Use with **funs()**. Also **mutate_if()**.
`mutate_all(faithful, funs(log10, log2(.)))`
`mutate_if(iris, is.numeric, funs(log(.)))`



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



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



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



Vector Functions

TO USE WITH MUTATE ()

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

vectorized function

OFFSETS

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

CUMULATIVE AGGREGATES

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

RANKINGS

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

MATH

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

MISC

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

Summary Functions

TO USE WITH SUMMARISE ()

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

summary function

COUNTS

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

LOCATION

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

LOGICALS

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

POSITION/ORDER

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

RANK

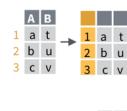
quantile() - nth quantile
min() - minimum value
max() - maximum value

SPREAD

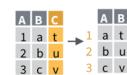
IQR() - Inter-Quartile Range
mad() - median absolute deviation
sdi() - standard deviation
var() - variance

Row Names

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



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



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

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

Combine Tables

COMBINE VARIABLES

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

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

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

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

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

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

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

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

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

A B X C B Y D
a t 1 t 3
b u 2 u 2
c v 3 N A N 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"))

A1 B1 C1 A2 B2
a t 1 d w
b u 2 b u
c v 3 a t
Use **suffix** to specify the suffix to give to unmatched columns that have the same name in both tables.
left_join(x, y, by = c("C" = "D"), suffix = c("1", "2"))

COMBINE CASES

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

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

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

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

EXTRACT ROWS

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

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

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

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

Data Visualization with ggplot2 :: CHEAT SHEET



Basics

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



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



Complete the template below to build a graph.

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

required

Not required, sensible defaults supplied

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

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

last_plot() Returns the last plot

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



Geoms

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

GRAPHICAL PRIMITIVES

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

a + geom_blank()
  (Useful for expanding limits)

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

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

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

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

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

LINE SEGMENTS

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

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

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

ONE VARIABLE continuous

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

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

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

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

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

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

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

discrete

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

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

TWO VARIABLES

continuous x , continuous y

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

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

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

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

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

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

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

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

discrete x , continuous y

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

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

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

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

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

discrete x , discrete y

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

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

THREE VARIABLES

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

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

continuous bivariate distribution

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

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

h + geom_density2d()
  x, y, alpha, colour, group, linetype, size

h + geom_hex()
  x, y, alpha, colour, fill, size
```

continuous function

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

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

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

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

visualizing error

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

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

j + geom_errorbar()
  x, y, 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 = USAArrests$Murder,
state = tolower(rownames(USAArrests)))
map <- map_data("state")
k <- ggplot(data, aes(fill = murder))

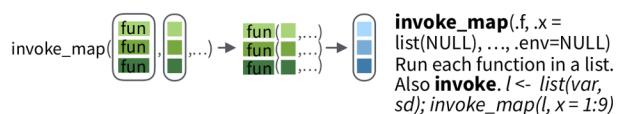
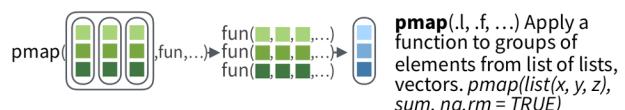
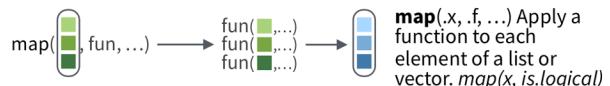
k + geom_map(aes(map_id = state), map = map)
  + expand_limits(x = map$long, y = map$lat),
  map_id, alpha, color, fill, group, linetype, size
```


Apply functions with purrr :: CHEAT SHEET



Apply Functions

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



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

OUTPUT

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

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

SHORTCUTS - within a purrr function:

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

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

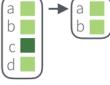
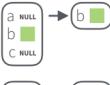
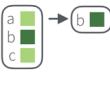
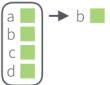
`~.x.y` becomes **function(x, y)** *x.y*, e.g. `map2(l, p, ~.x+y)` becomes `map2(l, p, function(l, p) l + p)`

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



Work with Lists

FILTER LISTS



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)`

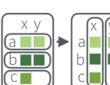
has_element(x, .y) Does a list contain an element? `has_element(x, "foo")`

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

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

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

RESHAPE LISTS

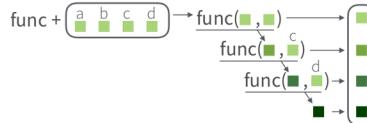
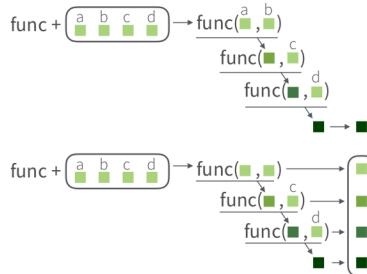


append(x, values, after = length(x)) Add to end of list. `append(x, list(d = 1))`

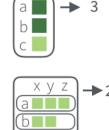
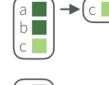
prepend(x, values, before = 1) Add to start of list. `prepend(x, list(d = 1))`

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

Reduce Lists



SUMMARISE LISTS



JOIN (TO) LISTS



Modify function behavior

compose() Compose multiple functions.

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

rerun() Rerun expression n times.

safely() Modify func to return list of results and errors.

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

partial() Create a version of a function that has some args preset to values.

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

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

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)`



Nested Data

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

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

n_iris\$data[[1]]

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

n_iris

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

n_iris\$data[[2]]

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

n_iris\$data[[3]]

Use a nested data frame to:

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

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

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

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

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

tidyverse::nest(data, ..., key = data)

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

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

n_iris %>% **unnest()**

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

Unnests a nested data frame.

List Column Workflow

1 Make a list column

Species S.L S.W P.L P.W

setosa 5.1 3.5 1.4 0.2

setosa 4.9 3.0 1.4 0.2

setosa 4.7 3.2 1.3 0.2

setosa 4.6 3.1 1.5 0.2

setosa 5.0 3.6 1.4 0.2

versi 7.0 3.2 4.7 1.4

versi 6.4 3.2 4.5 1.5

versi 6.9 3.1 4.9 1.5

versi 5.5 2.3 4.0 1.3

versi 6.5 2.8 4.6 1.5

virgini 6.3 3.3 6.0 2.5

virgini 5.8 2.7 5.1 1.9

virgini 7.1 3.0 5.9 2.1

virgini 6.3 2.9 5.6 1.8

virgini 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

setos 7.0 3.2 4.7 1.4

setos 6.4 3.2 4.5 1.5

setos 6.9 3.1 4.9 1.5

setos 5.5 2.3 4.0 1.3

setos 6.5 2.8 4.6 1.5

setos 6.3 3.3 6.0 2.5

setos 5.8 2.7 5.1 1.9

setos 7.1 3.0 5.9 2.1

setos 6.3 2.9 5.6 1.8

setos 6.5 3.0 5.8 2.2

Species <tibble [50 x 4]>

versi <tibble [50 x 4]>

virgini <tibble [50 x 4]>

Species S.L S.W P.L P.W

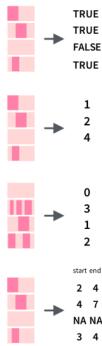
setos 7.0 3.2 4.7 1.4

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.

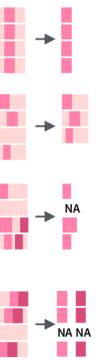


Detect Matches



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

Subset Strings



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



`str_subset(string, pattern)` Return only the strings that contain a pattern match.
`str_subset(fruit, "b")`



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



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

Manage Lengths



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



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

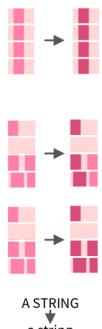


`str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...")` Truncate the width of strings, replacing content with ellipsis.
`str_trunc(fruit, 3)`



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

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.
`str_replace(fruit, "a", "-")`



`str_replace_all(string, pattern, replacement)` Replace all matched patterns in each string.
`str_replace_all(fruit, "a", "-")`

A STRING
↓
a string

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

a string
↓
A STRING

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

a string
↓
A String

`str_to_title(string, locale = "en")` Convert strings to title case.
`str_to_title(sentences)`

Join and Split



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



`str_c(..., sep = "", collapse = "")` Collapse a vector of strings into a single string.
`str_c(letters, collapse = "")`



`str_dup(string, times)` Repeat strings times times.
`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.
`str_split_fixed(fruit, " ", n=2)`



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



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

Order Strings



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



`str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)` Sort a character vector.
`str_sort(x)`

Helpers

apple
banana
pear

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

apple
banana
pear

`str_view(string, pattern, match = NA)` View HTML rendering of first regex match in each string.
`str_view(fruit, "[aeiou]")`

apple
banana
pear

`str_view_all(string, pattern, match = NA)` View HTML rendering of all regex matches.
`str_view_all(fruit, "[aeiou])`

apple
banana
pear

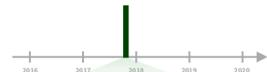
`str_wrap(string, width = 80, indent = 0, exdent = 0)` Wrap strings into nicely formatted paragraphs.
`str_wrap(sentences, 20)`

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

Dates and times with lubridate :: CHEAT SHEET



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"
```

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 wide variety of input formats.

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(), dyd(). dmy("4th of July '99")
```

2001: Q3

```
yq() Q for quarter. yq("2001: Q3")
```

2:01

hms::hms() Also lubridate::hms(), hm() and ms(), which return periods.* hms::hms(sec = 0, min = 1, hours = 2)

2017.5

```
date_decimal(decimal, tz = "UTC")
date_decimal(2017.5)
```



R Studio

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
```

GET AND SET COMPONENTS

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

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

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

2018-01-31 11:59:59

quarter(x, with_year = FALSE)
Quarter. *quarter(dt)*

2018-01-31 11:59:59

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

2018-01-31 11:59:59

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

2018-01-31 11:59:59

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

2018-01-31 11:59:59

leap_year(x) Is it a leap year?
leap_year(dt)

2018-01-31 11:59:59

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")

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

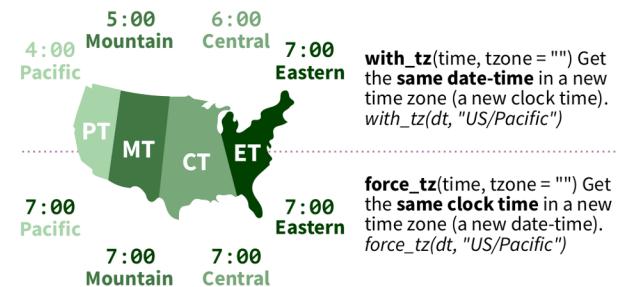
Tip: use a date with day > 12

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()*





Math with Date-times

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

A normal day

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



The start of daylight savings (spring forward)

```
gap <- ymd_hms("2018-03-11 01:30:00", tz="US/Eastern")
```



The end of daylight savings (fall back)

```
lap <- ymd_hms("2018-11-04 00:30:00", tz="US/Eastern")
```



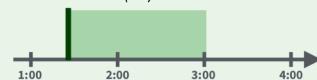
Leap years and leap seconds

```
leap <- ymd("2019-03-01")
```

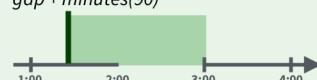


Periods track changes in clock times, which ignore time line irregularities.

`nor + minutes(90)`



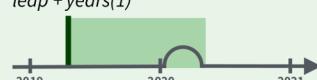
`gap + minutes(90)`



`lap + minutes(90)`

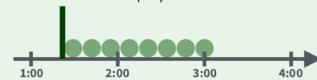


`leap + years(1)`



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

`nor + dminutes(90)`



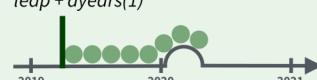
`gap + dminutes(90)`



`lap + dminutes(90)`



`leap + dyears(1)`



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

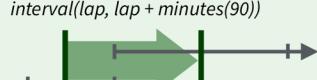
`interval(nor, nor + minutes(90))`



`interval(gap, gap + minutes(90))`



`interval(lap, lap + minutes(90))`



`interval(leap, leap + years(1))`



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

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

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

```
jan31 <- ymd(20180131)
```

```
jan31 + months(1)
```

```
## NA
```

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

```
jan31 %m%+ months(1)
```

```
## "2018-02-28"
```

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

```
add_with_rollback(jan31, months(1), roll_to_first = TRUE)
```

```
## "2018-03-01"
```

PERIODS

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

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

```
p <- months(3) + days(12)
```

```
"3m 12d 0H 0M 0S"
```

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)"
```

`dyears(x = 1) 31536000x 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 × 10-3 seconds.`

`dmicroseconds(x = 1) x × 10-6 seconds.`

`dnanoseconds(x = 1) x × 10-9 seconds.`

`dpicoseconds(x = 1) x × 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.

```
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())`

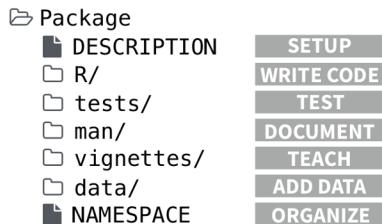
Package Development: : CHEAT SHEET



Package Structure

A package is a convention for organizing files into directories.

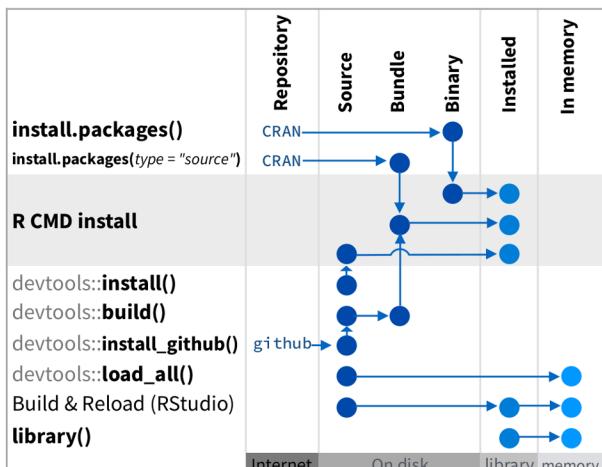
This sheet shows how to work with the 7 most common parts of an R package:



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

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

Or installed into an R library (loaded into memory during an R session) or archived online in a repository. Use the functions below to move between these states.



`devtools::use_build_ignore("file")`

Adds file to `.Rbuildignore`, a list of files that will not be included when package is built.



Setup (DESCRIPTION)

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

- You must have a DESCRIPTION file
- Add the packages that yours relies on with `devtools::use_package()`
Adds a package to the Imports or Suggests field

| CC0 | MIT | GPL-2 |
|----------------------|--|--|
| No strings attached. | MIT license applies to your code if re-shared. | GPL-2 license applies to your code, and all code anyone bundles with it, if re-shared. |

Package: mypackage

Title: Title of Package

Version: 0.1.0

Authors@R: person("Hadley", "Wickham", email = "hadley@me.com", role = c("aut", "cre"))

Description: What the package does (one paragraph)

Depends: R (>= 3.1.0)

License: GPL-2

LazyData: true

Imports:

dplyr (>= 0.4.0),

ggvis (>= 0.2)

Suggests:

knitr (>= 0.1.0)

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

Suggest packages that are not very essential to yours. Users can install them manually, or not, as they like.

Write Code (R/)

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

- Create a new package project with `devtools::create("path/to/name")`
Create a template to develop into a package.
- Save your code in R/ as scripts (extension .R)

WORKFLOW

1. Edit your code.
 2. Load your code with one of
`devtools::load_all()`
Re-loads all saved files in R/ into memory.
Ctrl/Cmd + Shift + L (keyboard shortcut)
Saves all open files then calls `load_all()`.
 3. Experiment in the console.
 4. Repeat.
- Use consistent style with r-pkgs.had.co.nz/r.html#style
 - Click on a function and press **F2** to open its definition
 - Search for a function with **Ctrl + .**



Visit r-pkgs.had.co.nz to learn much more about writing and publishing packages for R

Test (tests/)

Use tests/ to store tests that will alert you if your code breaks.

- Add a tests/ directory

- Import **testthat** with `devtools::use_testthat()`, which sets up package to use automated tests with testthat

- Write tests with `context()`, `test()`, and expect statements

- Save your tests as .R files in tests/testthat/

WORKFLOW

1. Modify your code or tests.
2. Test your code with one of
`devtools::test()`
Runs all tests in tests/
Ctrl/Cmd + Shift + T (keyboard shortcut)
3. Repeat until all tests pass

Example Test

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

Expect statement Tests

| | |
|---------------------------------|--|
| <code>expect_equal()</code> | is equal within small numerical tolerance? |
| <code>expect_identical()</code> | is exactly equal? |
| <code>expect_match()</code> | matches specified string or regular |
| <code>expect_output()</code> | prints specified output? |
| <code>expect_message()</code> | displays specified message? |
| <code>expect_warning()</code> | displays specified warning? |
| <code>expect_error()</code> | throws specified error? |
| <code>expect_is()</code> | output inherits from certain class? |
| <code>expect_false()</code> | returns FALSE? |
| <code>expect_true()</code> | returns TRUE? |

Document (□ man/)

□ man/ contains the documentation for your functions, the help pages in your package.

- Use roxygen comments to document each function beside its definition
- Document the name of each exported data set
- Include helpful examples for each function

WORKFLOW

1. Add roxygen comments in your .R files
2. Convert roxygen comments into documentation with one of:

devtools::document()

Converts roxygen comments to .Rd files and places them in □ man/. Builds NAMESPACE.

Ctrl/Cmd + Shift + D (Keyboard Shortcut)

3. Open help pages with ? to preview documentation
4. Repeat

.Rd FORMATTING TAGS

| | |
|-----------------------|--|
| \emph{italic text} | \email{name@foo.com} |
| \strong{bold text} | \href{url}{display} |
| \code{function(args)} | \url{url} |
| \pkg{package} | |
| \dontrun{code} | \link[=dest]{display} |
| \dontshow{code} | \link4class{class} |
| \donttest{code} | \code{\link{function}} |
| \deqn{a + b (block)} | \code{\link[package]{function}} |
| \eqn{a + b (inline)} | \tabular{lcr}{
left \tab centered \tab right \cr
cell \tab cell \tab cell \cr} |

Teach (□ vignettes/)

□ vignettes/ holds documents that teach your users how to solve real problems with your tools.

- Create a □ vignettes/ directory and a template vignette with devtools::use_vignette()
Adds template vignette as vignettes/my-vignette.Rmd.
- Append YAML headers to your vignettes (like right)
- Write the body of your vignettes in R Markdown (rmarkdown.rstudio.com)

ROXYGEN2

The **roxygen2** package lets you write documentation inline in your .R files with a shorthand syntax. devtools implements roxygen2 to make documentation.



- Add roxygen documentation as comment lines that begin with #’.
- Place comment lines directly above the code that defines the object documented.
- Place a roxygen @ tag (right) after #’ to supply a specific section of documentation.
- Untagged lines will be used to generate a title, description, and details section (in that order)

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

COMMON ROXYGEN TAGS

| | | | |
|------------------|----------------|-----------------|------|
| @aliases | @inheritParams | @seealso | |
| @concepts | @keywords | @format | |
| @describeln | @param | @source | data |
| @examples | @rdname | @include | |
| @export | @return | @slot | S4 |
| @family | @section | @field | RC |

Add Data (□ data/)

The □ data/ directory allows you to include data with your package.

- Save data as .Rdata files (suggested)
- Store data in one of **data/**, **R/Sysdata.rda**, **inst/extdata**
- Always use **LazyData: true** in your DESCRIPTION file.



devtools::use_data()

Adds a data object to data/ (R/Sysdata.rda if **internal = TRUE**)

devtools::use_data_raw()

Adds an R Script used to clean a data set to data-raw/. Includes data-raw/ on .Rbuildignore.

Store data in

- **data/** to make data available to package users
- **R/sysdata.rda** to keep data internal for use by your functions.
- **inst/extdata** to make raw data available for loading and parsing examples. Access this data with **system.file()**

Organize (□ NAMESPACE)

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

- Export functions for users by placing **@export** in their roxygen comments
- Import objects from other packages with **package::object** (recommended) or **@import**, **@importFrom**, **@importClassesFrom**, **@importMethodsFrom** (not always recommended)

WORKFLOW

1. Modify your code or tests.
2. Document your package (devtools::document())
3. Check NAMESPACE
4. Repeat until NAMESPACE is correct

SUBMIT YOUR PACKAGE

r-pkgs.had.co.nz/release.html

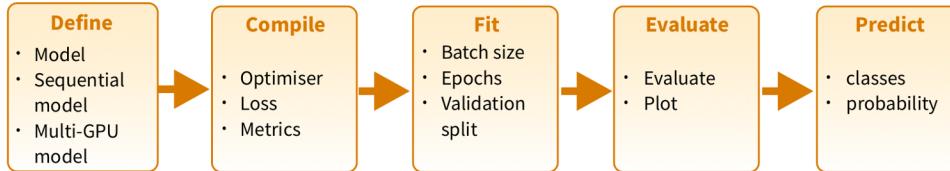
Deep Learning with Keras :: CHEAT SHEET



Intro

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation. It supports multiple backends, 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.



<https://keras.rstudio.com>

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

The “Hello, World!”
of deep learning

See ?install_keras
for GPU instructions

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()
```

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

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

`layer_repeat_vector()` Repeats the input n times

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

`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

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)
```



More layers

CONVOLUTIONAL LAYERS

| | |
|---|---|
|  | <code>layer_conv_1d()</code> 1D, e.g. temporal convolution |
|  | <code>layer_conv_2d_transpose()</code> Transposed 2D (deconvolution) |
|  | <code>layer_conv_2d()</code> 2D, e.g. spatial convolution over images |
|  | <code>layer_conv_3d_transpose()</code> Transposed 3D (deconvolution) |
|  | <code>layer_conv_3d()</code> 3D, e.g. spatial convolution over volumes |
|  | <code>layer_conv_lstm_2d()</code> Convolutional LSTM |
|  | <code>layer_separable_conv_2d()</code> Depthwise separable 2D |
|  | <code>layer_upsampling_1d()</code>
<code>layer_upsampling_2d()</code>
<code>layer_upsampling_3d()</code> Upsampling layer |
|  | <code>layer_zero_padding_1d()</code>
<code>layer_zero_padding_2d()</code>
<code>layer_zero_padding_3d()</code> Zero-padding layer |
|  | <code>layer_cropping_1d()</code>
<code>layer_cropping_2d()</code>
<code>layer_cropping_3d()</code> Cropping layer |

POOLING LAYERS

| | |
|---|---|
|  | <code>layer_max_pooling_1d()</code>
<code>layer_max_pooling_2d()</code>
<code>layer_max_pooling_3d()</code> Maximum pooling for 1D to 3D |
|  | <code>layer_average_pooling_1d()</code>
<code>layer_average_pooling_2d()</code>
<code>layer_average_pooling_3d()</code> Average pooling for 1D to 3D |
|  | <code>layer_global_max_pooling_1d()</code>
<code>layer_global_max_pooling_2d()</code>
<code>layer_global_max_pooling_3d()</code> Global maximum pooling |
|  | <code>layer_global_average_pooling_1d()</code>
<code>layer_global_average_pooling_2d()</code>
<code>layer_global_average_pooling_3d()</code> Global average pooling |

ACTIVATION LAYERS

| | |
|---|---|
|  | <code>layer_activation()</code> object, activation
Apply an activation function to an output |
|  | <code>layer_activation_leaky_relu()</code> Leaky version of a rectified linear unit |
|  | <code>layer_activation_parametric_relu()</code> Parametric rectified linear unit |
|  | <code>layer_activation_thresholded_relu()</code> Thresholded rectified linear unit |
|  | <code>layer_activation_elu()</code> Exponential linear unit |

DROPOUT LAYERS

| | |
|---|---|
|  | <code>layer_dropout()</code> Applies dropout to the input |
|  | <code>layer_spatial_dropout_1d()</code>
<code>layer_spatial_dropout_2d()</code>
<code>layer_spatial_dropout_3d()</code> Spatial 1D to 3D version of dropout |

RECURRENT LAYERS

| | |
|---|---|
|  | <code>layer_simple_rnn()</code> Fully-connected RNN where the output is to be fed back to input |
|  | <code>layer_gru()</code> Gated recurrent unit - Cho et al |

| | |
|---|--|
|  | <code>layer_cudnn_gru()</code> Fast GRU implementation backed by CuDNN |
|  | <code>layer_cudnn_lstm()</code> Fast LSTM implementation backed by CuDNN |

LOCALLY CONNECTED LAYERS

| | |
|---|--|
|  | <code>layer_locally_connected_1d()</code>
<code>layer_locally_connected_2d()</code> Similar to convolution, but weights are not shared, i.e. different filters for each patch |
|---|--|

Preprocessing

SEQUENCE PREPROCESSING

| | |
|------------------------------------|--|
| <code>pad_sequences()</code> | Pads each sequence to the same length (length of the longest sequence) |
| <code>skipgrams()</code> | Generates skipgram word pairs |
| <code>make_sampling_table()</code> | Generates word rank-based probabilistic sampling table |

TEXT PREPROCESSING

| | |
|---|--|
| <code>text_tokenizer()</code> | Text tokenization utility |
| <code>fit_text_tokenizer()</code> | Update tokenizer internal vocabulary |
| <code>save_text_tokenizer(); load_text_tokenizer()</code> | Save a text tokenizer to an external file |
| <code>texts_to_sequences(); texts_to_sequences_generator()</code> | Transforms each text in texts to sequence of integers |
| <code>texts_to_matrix(); sequences_to_matrix()</code> | Convert a list of sequences into a matrix |
| <code>text_one_hot()</code> | One-hot encode text to word indices |
| <code>text_hashing_trick()</code> | Converts a text to a sequence of indexes in a fixed-size hashing space |
| <code>text_to_word_sequence()</code> | Convert text to a sequence of words (or tokens) |

IMAGE PREPROCESSING

| | |
|---|---|
| <code>image_load()</code> | Loads an image into PIL format. |
| <code>flow_images_from_data()</code> | |
| <code>flow_images_from_directory()</code> | Generates batches of augmented/normalized data from images and labels, or a directory |
| <code>image_data_generator()</code> | Generate minibatches of image data with real-time data augmentation. |
| <code>fit_image_data_generator()</code> | Fit image data generator internal statistics to some sample data |
| <code>generator_next()</code> | Retrieve the next item |
| <code>image_to_array(); image_array_resize()</code> | |
| <code>image_array_save()</code> | 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

Data Science in Spark with sparklyr :: CHEAT SHEET

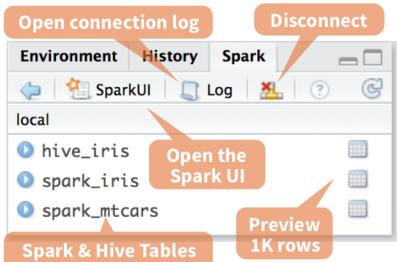


Intro

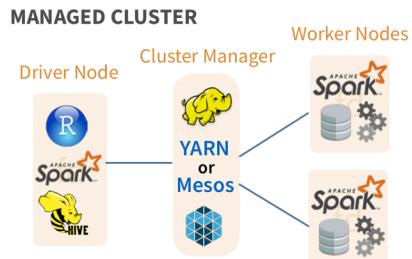
sparklyr is an R interface for Apache Spark™, it provides a complete **dplyr** backend and the option to query directly using **Spark SQL** statement. With sparklyr, you can orchestrate distributed machine learning using either **Spark's MLlib** or **H2O** Sparkling Water.

Starting with **version 1.044**, **RStudio Desktop, Server and Pro** include integrated support for the **sparklyr package**. You can create and manage connections to Spark clusters and local Spark instances from inside the IDE.

RStudio Integrates with sparklyr

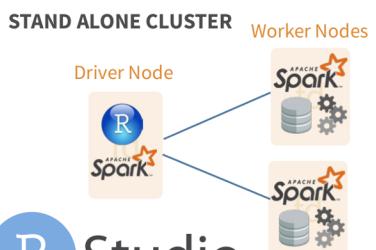


Cluster Deployment



USING LIVY (Experimental)

1. The Livy REST application should be running on the cluster
2. Connect to the cluster
`sc <- spark_connect(method = "livy", master = "http://host:port")`



Tuning Spark

EXAMPLE CONFIGURATION

```
config <- spark_config()
config$spark.executor.cores <- 2
config$spark.executor.memory <- "4G"
sc <- spark_connect(master="yarn-client",
                     config = config, version = "2.0.1")
```

IMPORTANT TUNING PARAMETERS with defaults

- `spark.yarn.am.cores`
- `spark.yarn.am.memory 512m`
- `spark.network.timeout 120s`
- `spark.executor.memory 1g`
- `spark.executor.cores 1`
- `spark.executor.instances`
- `spark.executor.extraJavaOptions`
- `spark.executor.heartbeatInterval 10s`
- `sparklyr.shell.executor-memory`
- `sparklyr.shell.driver-memory`

Using sparklyr

A brief example of a data analysis using Apache Spark, R and sparklyr in local mode

```
library(sparklyr); library(dplyr); library(ggplot2);
library(tidyr);
set.seed(100)
```

Install Spark locally

```
spark_install("2.0.1")
```

Connect to local version

```
sc <- spark_connect(master = "local")
```

```
import_iris <- copy_to(sc, iris, "spark_iris",
                      overwrite = TRUE)
```

Copy data to Spark memory

```
partition_iris <- sdf_partition(
  import_iris, training=0.5, testing=0.5)
```

Partition data

```
sdf_register(partition_iris,
  c("spark_iris_training", "spark_iris_test"))
```

Create a hive metadata for each partition

```
tidy_iris <- tbl(sc, "spark_iris_training") %>%
  select(Species, Petal_Length, Petal_Width)
```

Spark ML Decision Tree Model

```
model_iris <- tidy_iris %>%
  ml_decision_tree(response = "Species",
                    features = c("Petal_Length", "Petal_Width"))
```

Create reference to Spark table

```
test_iris <- tbl(sc, "spark_iris_test")
```

```
pred_iris <- sdf_predict(
  model_iris, test_iris) %>%
  collect
```

Bring data back into R memory for plotting

```
pred_iris %>%
  inner_join(data.frame(prediction = 0:2,
                        lab = model_iris$model.parameters$labels)) %>%
  ggplot(aes(Petal_Length, Petal_Width, col = lab)) +
  geom_point()
```

```
spark_disconnect(sc)
```

Disconnect

Reactivity

COPY A DATA FRAME INTO SPARK

```
sdf_copy_to(sc, iris, "spark_iris")
```

```
sdf_copy_to(sc, x, name, memory, repartition,  
overwrite)
```

IMPORT INTO SPARK FROM A FILE

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()`

SPARK SQL COMMANDS

```
DBI::dbWriteTable(sc, "spark_iris", iris)  
DBI::dbWriteTable(conn, name,  
value)
```

FROM A TABLE IN HIVE

```
my_var <- tbl_cache(sc, name=  
"hive_iris")  
tbl_cache(sc, name, force = TRUE)  
Loads the table into memory  
  
my_var <- dplyr::tbl(sc,  
name = "hive_iris")  
dplyr::tbl(scr, ...)  
Creates a reference to the table  
without loading it into memory
```

Visualize & Communicate

DOWNLOAD DATA TO R MEMORY

```
r_table <- collect(my_table)  
plot(Petal_Width ~ Petal_Length, data = r_table)  
dplyr::collect(x)
```

Download a Spark DataFrame to an R DataFrame

```
sdf_read_column(x, column)
```

Returns contents of a single column to R

SAVE FROM SPARK TO FILE SYSTEM

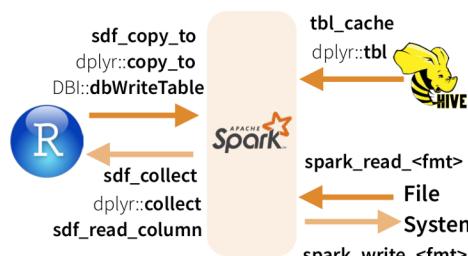
Arguments that apply to all functions: `x, path`

CSV `spark_read_csv(header = TRUE,
delimiter = "", quote = "", escape = "\\",
charset = "UTF-8", null_value = NULL)`

JSON `spark_read_json(mode = NULL)`

PARQUET `spark_read_parquet(mode = NULL)`

Reading & Writing from Apache Spark



Extensions

Create an R package that calls the full Spark API & provide interfaces to Spark packages.

CORE TYPES

`spark_connection()` Connection between R and the

Spark shell process

`spark_jobj()` Instance of a remote Spark object

`spark_dataframe()` Instance of a remote Spark DataFrame object

CALL SPARK FROM R

`invoke()` Call a method on a Java object

`invoke_new()` Create a new object by invoking a constructor

`invoke_static()` Call a static method on an object

MACHINE LEARNING EXTENSIONS

`ml_create_dummy_variables()` ml_options()

`ml_prepare_dataframe()` ml_model()

`ml_prepare_response_features_intercept()`

Model (MLlib)

```
ml_decision_tree(my_table,  
response = "Species", features =  
c("Petal_Length", "Petal_Width"))
```

```
ml_als_factorization(x, user.column = "user",  
rating.column = "rating", item.column = "item",  
rank = 10L, regularization.parameter = 0.1, iter.max = 10L,  
ml.options = ml_options())
```

```
ml_decision_tree(x, response, features, max.bins = 32L, max.depth  
= 5L, type = c("auto", "regression", "classification"), ml.options =  
ml_options()) Same options for: ml_gradient_boosted_trees
```

```
ml_generalized_linear_regression(x, response, features,  
intercept = TRUE, family = gaussian(link = "identity"), iter.max =  
100L, ml.options = ml_options())
```

```
ml_kmeans(x, centers, iter.max = 100, features = dplyr::tbl_vars(x),  
compute.cost = TRUE, tolerance = 1e-04, ml.options = ml_options())
```

```
ml_lda(x, features = dplyr::tbl_vars(x), k = length(features), alpha =  
(50/k) + 1, beta = 0.1 + 1, ml.options = ml_options())
```

```
ml_linear_regression(x, response, features, intercept = TRUE,  
alpha = 0, lambda = 0, iter.max = 100L, ml.options = ml_options())  
Same options for: ml_logistic_regression
```

```
ml_multilayer_perceptron(x, response, features, layers, iter.max =  
100, seed = sample(.Machine$integer.max, 1), ml.options =  
ml_options())
```

```
ml_naive_bayes(x, response, features, lambda = 0, ml.options =  
ml_options())
```

```
ml_one_vs_rest(classifier, response, features, ml.options =  
ml_options())
```

```
ml_pca(x, features = dplyr::tbl_vars(x), ml.options = ml_options())
```

```
ml_random_forest(x, response, features, max.bins = 32L,  
max.depth = 5L, num.trees = 20L, type = c("auto", "regression",  
"classification"), ml.options = ml_options())
```

```
ml_survival_regression(x, response, features, intercept =  
TRUE, censor = "censor", iter.max = 100L, ml.options = ml_options())
```

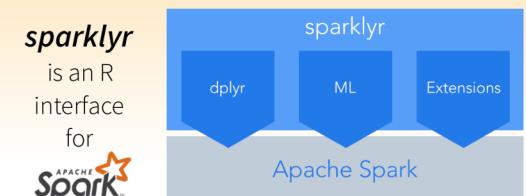
```
ml_binary_classification_eval(predicted_tbl_spark, label, score,  
metric = "areaUnderROC")
```

```
ml_classification_eval(predicted_tbl_spark, label, predicted_lbl,  
metric = "f1")
```

```
ml_tree_feature_importance(sc, model)
```

sparklyr

is an R
interface
for

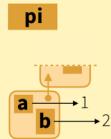




Tidy evaluation with rlang :: CHEAT SHEET

Vocabulary

Tidy Evaluation (Tidy Eval) is not a package, but a framework for doing non-standard evaluation (i.e. delayed evaluation) that makes it easier to program with tidyverse functions.



Symbol - a name that represents a value or object stored in R. `is_symbol(expr(pi))`

Environment - a list-like object that binds symbols (names) to objects stored in memory. Each env contains a link to a second, **parent** env, which creates a chain, or search path, of environments. `is_environment(current_env())`

`rlang::caller_env(n = 1)` Returns calling env of the function it is in.

`rlang::child_env(.parent, ...)` Creates new env as child of .parent. Also **env**.

`rlang::current_env()` Returns execution env of the function it is in.

1

abs (1)

Constant - a bare value (i.e. an atomic vector of length 1). `is_bare_atomic(1)`

Call object - a vector of symbols/constants/calls that begins with a function name, possibly followed by arguments. `is_call(expr(abs(1)))`

pi — code
3.14 — result

Code - a sequence of symbols/constants/calls that will return a result if evaluated. Code can be:

1. Evaluated immediately (**Standard Eval**)
2. Quoted to use later (**Non-Standard Eval**)

`is_expression(expr(pi))`

Expression - an object that stores quoted code without evaluating it. `is_expression(expr(a + b))`

Quosure- an object that stores both quoted code (without evaluating it) and the code's environment. `is_quosure(quo(a + b))`

a `rlang::quo_get_env(quo)` Return the environment of a quosure.

a `rlang::quo_set_env(quo, expr)` Set the environment of a quosure.

a + b `rlang::quo_get_expr(quo)` Return the expression of a quosure.

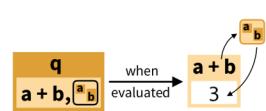
Expression Vector - a list of pieces of quoted code created by base R's `expression` and `parse` functions. Not to be confused with **expression**.



Quoting Code

Quote code in one of two ways (if in doubt use a quosure):

QUOSURES



Quosure- An expression that has been saved with *an environment* (aka a closure).

A quosure can be evaluated later in the stored environment to return a predictable result.

`rlang::quo(expr)` Quote contents as a quosure. Also **quos** to quote multiple expressions. `a <- 1; b <- 2; q <- quo(a + b); qs <- quos(a, b)`

`rlang::enquo(arg)` Call from within a function to quote what the user passed to an argument as a quosure. Also **enquos** for multiple args.
`quote_this <- function(x) enquo(x)`
`quote_these <- function(...) enquos(...)`

`rlang::new_quosure(expr, env = caller_env())` Build a quosure from a quoted expression and an environment.
`new_quosure(expr(a + b), current_env())`



EXPRESSION



Quoted Expression - An expression that has been saved by itself.

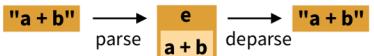
A quoted expression can be evaluated later to return a result that will depend on the environment it is evaluated in

`rlang::expr(expr)` Quote contents. Also **exprs** to quote multiple expressions. `a <- 1; b <- 2; e <- expr(a + b); es <- exprs(a, b, a + b)`

`rlang::enexpr(arg)` Call from within a function to quote what the user passed to an argument. Also **enexprs** to quote multiple arguments.
`quote_that <- function(x) enexpr(x)`
`quote_those <- function(...) enexprs(...)`

`rlang::ensym(x)` Call from within a function to quote what the user passed to an argument as a symbol, accepts strings. Also **ensyms**.
`quote_name <- function(name) ensym(name)`
`quote_names <- function(...) ensyms(...)`

Parsing and Deparsing



Parse - Convert a string to a saved expression.

• • •

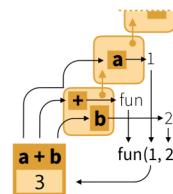
`rlang::parse_expr(x)` Convert a string to an expression. Also **parse_exprs**, **sym**, **parse_quo**, **parse_quos**. `e <- parse_expr("a + b")`

Deparse - Convert a saved expression to a string.

• • •

`rlang::expr_text(expr, width = 60L, nlines = Inf)` Convert expr to a string. Also **quo_name**, **expr_text(e)**

Evaluation



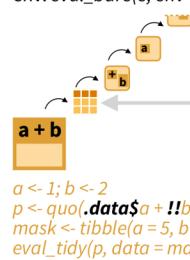
To evaluate an expression, R :

1. Looks up the symbols in the expression in the active environment (or a supplied one), followed by the environment's parents
2. Executes the calls in the expression

The result of an expression depends on which environment it is evaluated in.

QUOTED EXPRESSION

`rlang::eval_bare(expr, env = parent.frame())` Evaluate expr in env. `eval_bare(e, env = GlobalEnv)`



`a <- 1; b <- 2`
`p <- quo(.data$a + !!b)`
`mask <- tibble(a = 5, b = 6)`
`eval_tidy(p, data = mask)`

QUOSURES (and quoted exprs)

`rlang::eval_tidy(expr, data = NULL, env = caller_env())` Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment. `eval_tidy(q)`

Data Mask - If data is non-NULL, `eval_tidy` inserts data into the search path before env, matching symbols to names in data.

Use the pronoun **.data\$** to force a symbol to be matched in data, and **!!** (see back) to force a symbol to be matched in the environments.

Building Calls

`rlang::call2(fn, ..., .ns = NULL)` Create a call from a function and a list of args. Use `exec` to create and then evaluate the call. (See back page for **!!!**) `args <- list(x = 4, base = 2)`

`log(x = 4, base = 2)`

2

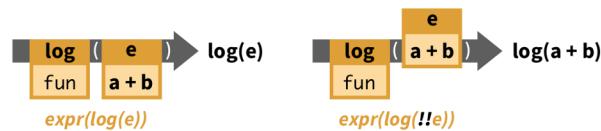
`call2("log", x = 4, base = 2)`
`call2("log", !!!args)`
`exec("log", x = 4, base = 2)`
`exec("log", !!!args)`



Quasiquotation (!! , !!!, :=)

QUOTATION

Storing an expression without evaluating it.
`e <- expr(a + b)`

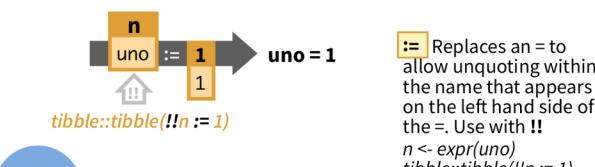
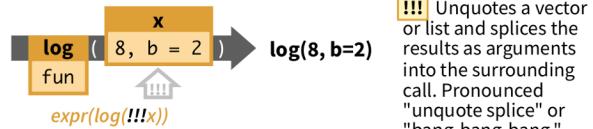
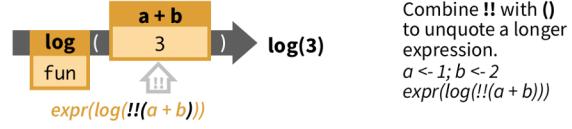
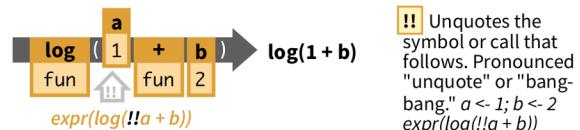


`rlang` provides `!!`, `!!!`, and `:=` for doing quasiquotation.

`!!`, `!!!`, and `:=` are not functions but syntax (symbols recognized by the functions they are passed to). Compare this to how

- . is used by `magrittr::%>%()`
- . is used by `stats::lm()`
- .x is used by `purrr::map()`, and so on.

`!!`, `!!!`, and `:=` are only recognized by some `rlang` functions and functions that use those functions (such as tidyverse functions).



Programming Recipes

Quoting function- A function that quotes any of its arguments internally for delayed evaluation in a chosen environment. You must take **special steps to program safely** with a quoting function.

How to spot a quoting function?
A function quotes an argument if the argument returns an error when run on its own.

Many tidyverse functions are quoting functions: e.g. `filter`, `select`, `mutate`, `summarise`, etc.

```
dplyr::filter(cars, speed == 25)
          speed dist
          1    25   85
```

```
speed == 25
          Error!
```

PROGRAM WITH A QUOTING FUNCTION

```
data_mean <- function(data, var) {
  require(dplyr)
  var <- rlang::enquo(var)
  data %>%
    summarise(mean = mean (!!var)) 2
}
```

1. Capture user argument that will be quoted with `rlang::enquo`.
2. Unquote the user argument into the quoting function with `!!`.

PASS MULTIPLE ARGUMENTS TO A QUOTING FUNCTION

```
group_mean <- function(data, var, ...) {
  require(dplyr)
  var <- rlang::enquo(var)
  group_vars <- rlang::enquos(...) 1
  data %>%
    group_by (!!group_vars) %>%
    summarise(mean = mean (!!var))
}
```

1. Capture user arguments that will be quoted with `rlang::enquo`.
2. Unquote splice the user arguments into the quoting function with `!!!`.

MODIFY USER ARGUMENTS

```
my_do <- function(f, v, df) {
  f <- rlang::enquo(f)
  v <- rlang::enquo(v)
  todo <- rlang::quo (!!f) (!!v))
  rlang::eval_tidy(todo, df)
}
```

1. Capture user arguments with `rlang::enquo`.
2. **Unquote** user arguments into a new expression or quoture to use
3. **Evaluate** the new expression/ quoture instead of the original argument

APPLY AN ARGUMENT TO A DATA FRAME

```
subset2 <- function(df, rows) {
  rows <- rlang::enquo(rows)
  vals <- rlang::eval_tidy(rows, data = df)
  df[vals, , drop = FALSE]
}
```

1. Capture user argument with `rlang::enquo`.
2. Evaluate the argument with `rlang::eval_tidy`. Pass the data frame to `data` to use as a data mask.
3. **Suggest** in your documentation that your users use the `.data` and `.env` pronouns.

WRITE A FUNCTION THAT RECOGNIZES QUASIQUOTATION (!! , !!!, :=)

1. Capture the quasiquotation-aware argument with `rlang::enquo`.
2. Evaluate the arg with `rlang::eval_tidy`.

```
add1 <- function(x) {
  q <- rlang::enquo(x)
  rlang::eval_tidy(q) + 1
}
```

PASS TO ARGUMENT NAMES OF A QUOTING FUNCTION

```
named_mean <- function(data, var) {
  require(dplyr)
  var <- rlang::ensym(var)
  data %>%
    summarise (!!name := mean (!!var)) 2
}
```

1. Capture user argument that will be quoted with `rlang::ensym`.
2. Unquote the name into the quoting function with `!!` and `:=`.

PASS CRAN CHECK

```
#'@importFrom rlang .data
  mutate_y <- function(df) {
    dplyr::mutate(df, y = .data$a + 1)
  }
```

Quoted arguments in tidyverse functions can trigger an **R CMD check** NOTE about undefined global variables. To avoid this:

1. Import `rlang::data` to your package, perhaps with the `roxygen2` tag `@importFrom rlang .data`
2. Use the `.data$` pronoun in front of variable names in tidyverse functions

caret Package

Cheat Sheet

Specifying the Model

Possible syntaxes for specifying the variables in the model:

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

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

Remember to:

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

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

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

Parallel Processing

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

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

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

Preprocessing

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

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

Methods include:

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

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

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

Adding Options

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

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

Resampling Options

`trainControl` is used to choose a resampling method:

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

Methods and options are:

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

Performance Metrics

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

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

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

Grid Search

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

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

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

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

Random Search

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

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

Subsampling

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

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

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

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.

A screenshot of the RStudio interface showing an R Markdown document. It contains several R code chunks and Python code chunks. The Python code includes importing modules like seaborn, loading datasets, and creating plots. The RStudio interface shows the R Console and R Markdown panes. A legend at the bottom left maps R types to Python equivalents.

```
1 ```{r setup, include = FALSE}
2 library(reticulate)
3 virtualenv_create("fmri-proj")
4 py_install("seaborn", envname = "fmri-proj")
5 use_virtualenv("fmri-proj")
6
7
8 ```{python, echo = FALSE}
9 import seaborn as sns
10 fmri = sns.load_dataset("fmri")
11
12
13 ```{r}
14 f1 <- subset(py$fmri, region == "parietal")
15
16
17 ```{python}
18 import matplotlib as mpl
19 sns.lmplot("timepoint", "signal", data=r.f1)
20 mpl.pyplot.show()
21
```

A screenshot of the RStudio interface showing an R Script file. It uses the `py` object to run Python code, such as `py_run_file` and `py_run_string`, to execute Python scripts and print results.

```
1 library(reticulate)
2 py_install("seaborn")
3 use_virtualenv("r-reticulate")
4
5 sns <- import("seaborn")
6
7 fmri <- sns.load_dataset("fmri")
8 dim(fmri)
9
10 # creates tips
11 source_python("python.py")
12 dim(tips)
13
14 # creates tips in main
15 py_run_file("python.py")
16 dim(py$tips)
17
18 py_run_string("print(tips.shape)")
19
```

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 |
| NUL, 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(object)` Wrap an R function in a Python function with the same signature. `py_func(xor)`

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

`iterate(..., convert = FALSE)` 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!"))`

Python in R code

Call Python from R 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()`.

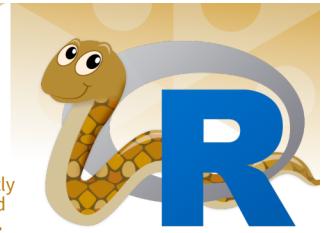
- `py_run_string(code, local = FALSE, convert = TRUE)` Run Python code (passed as a string) in the main module. `py_run_string("x = 10"); py$x`

- `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`



Python in the IDE

Requires reticulate plus RStudio v1.2 or higher.

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**)

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

matplotlib plots display in plots pane.

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

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**).
2. Type commands at **>>>** prompt
3. Press **Enter** to run code
4. Type **exit** to close and return to R console

```

Console Terminal Jobs
~/Documents/cheatsheets/
> reticulate::repl_python()
Python 2.7.10 (/Users/garrettgrolemond/.virtualenvs/r-reticulate/bin/python)
Reticulate 1.12 REPL -- A Python interpreter in R.

>>> import pandas as pd
>>> import matplotlib as mpl
>>> 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

- **py_discover_config()** Return all detected versions of Python. Use **py_config** to check which version has been loaded. **py_config()**
- **py_available(initialize = FALSE)** Check if Python is available on your system. Also **py_module_available**, **py_numpy_module**, **py_available()**

Create a Python env

- **virtualenv_create(envname)** Create a new virtualenv. **virtualenv_create("r-pandas")**
- **conda_create(envname, packages = NULL, conda = "auto")** Create a new Conda env. **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 = "r-reticulate", method = c("auto", "virtualenv", "conda"), conda = "auto", ...)** Installs Python packages into a Python env named "r-reticulate". **py_install("pandas")**
- **virtualenv_install(envname, packages, ignore_installed = FALSE)** Install a package within a virtualenv. **virtualenv_install("r-pandas", packages = "pandas")**
- **virtualenv_remove(envname, packages = NULL, confirm = interactive())** Remove individual packages or an entire virtualenv. **virtualenv_remove("r-pandas", packages = "pandas")**
- **conda_install(envname, packages, forge = TRUE, pip = FALSE, pip_ignore_installed = TRUE, conda = "auto")** Install a package within a Conda env. **conda_install("r-pandas", packages = "plotly")**
- **conda_remove(envname, packages = NULL, conda = "auto")** Remove individual packages or an entire Conda env. **conda_remove("r-pandas", packages = "plotly")**

Suggest an env to use

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.
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**...

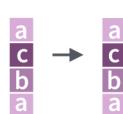


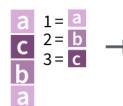
Factors withforcats :: CHEAT SHEET

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

Factors

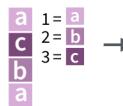
R represents categorical data with factors. A **factor** is an integer vector with a **levels** attribute that stores a set of mappings between integers and categorical values. When you view a factor, R displays not the integers, but the values associated with them.

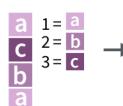
 **Create a factor with factor()**
`factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA)` Convert a vector to a factor. Also **as_factor**.
`f <- factor(c("a", "c", "b", "a"), levels = c("a", "b", "c"))`

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

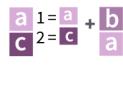
Use unclass() to see its structure

Inspect Factors

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

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

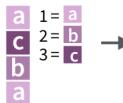
Combine Factors

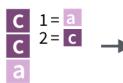
 **fct_c(...)** Combine factors with different levels.
`f1 <- factor(c("a", "c"))`
`f2 <- factor(c("b", "a"))`
`fct_c(f1, f2)`

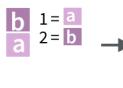
 **fct_unify(fs, levels = lvs_union(fs))** Standardize levels across a list of factors. `fct_unify(list(f2, f1))`

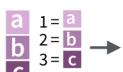


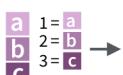
Change the order of levels

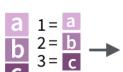
 **fct_relevel(.f, ..., after = 0L)** Manually reorder factor levels.
`fct_relevel(f, c("b", "c", "a"))`

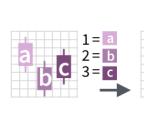
 **fct_infreq(f, ordered = NA)** Reorder levels by the frequency in which they appear in the data (highest frequency first).
`f3 <- factor(c("c", "c", "a"))`
`fct_infreq(f3)`

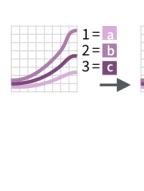
 **fct_inorder(f, ordered = NA)** Reorder levels by order in which they appear in the data. `fct_inorder(f2)`

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

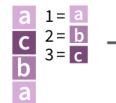
 **fct_shift(f)** Shift levels to left or right, wrapping around end. `fct_shift(f4)`

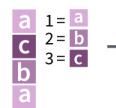
 **fct_shuffle(f, n = 1L)** Randomly permute order of factor levels. `fct_shuffle(f4)`

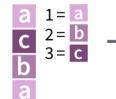
 **fct_reorder(.f, .x, .fun=median, ..., .desc = FALSE)** Reorder levels by their relationship with another variable.
`boxplot(data = iris, Sepal.Width ~ fct_reorder(Species, Sepal.Width))`

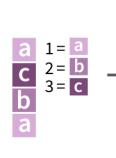
 **fct_reorder2(.f, .x, .y, .fun = last2, ..., .desc = TRUE)** Reorder levels by their final values when plotted with two other variables.
`ggplot(data = iris,`
`aes(Sepal.Width, Sepal.Length,`
`color = fct_reorder2(Species,`
`Sepal.Width, Sepal.Length))) +`
`geom_smooth()`

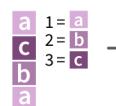
Change the value of levels

 **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))`

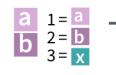
 **fct_anon(f, prefix = "")** Anonymize levels with random integers. `fct_anon(f)`

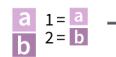
 **fct_collapse(f, ...)** Collapse levels into manually defined groups. `fct_collapse(f, x = c("a", "b"))`

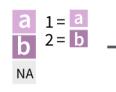
 **fct_lump(f, n, prop, w = NULL, other_level = "Other", ties.method = c("min", "average", "first", "last", "random", "max"))** Lump together least/most common levels into a single level. Also **fct_lump_min**.
`fct_lump(f, n = 1)`

 **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)))`



Orange County R Users Group (OCRUG)



2020-04 Hackathon

We would like to thank our sponsors
Data Science Go
R Consortium
RStudio
Taylor and Francis
UCI Paul Merage School of Business

OCRUG Website: <https://ocrug.com>

OCRUG Meetup.com: <https://www.meetup.com/OC-RUG/>

GitHub: <https://github.com/ocrug>

Hackathon repo: <https://github.com/ocrug/hackathon-2020-04>

Slack Signup: <https://tinyurl.com/socalrug-slack-signup>

Slack Channel: <https://solcalrug.slack.com>

Youtube Channel: <https://tinyurl.com/ocrug-youtube>

Twitter: <https://tinyurl.com/ocrug-twitter>

Instagram: <https://tinyurl.com/ocrug-instagram>