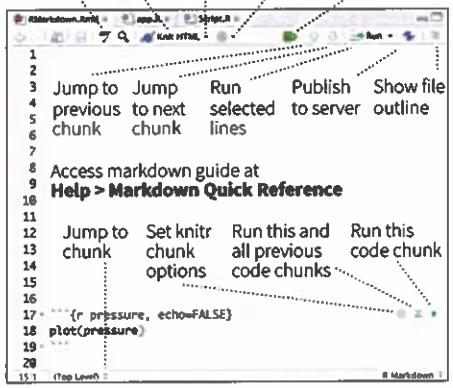
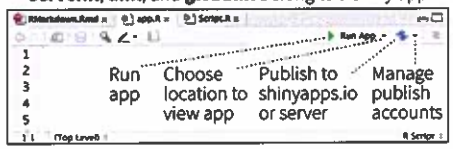
# RStudio IDE : : CHEAT SHEET

## Documents and Apps

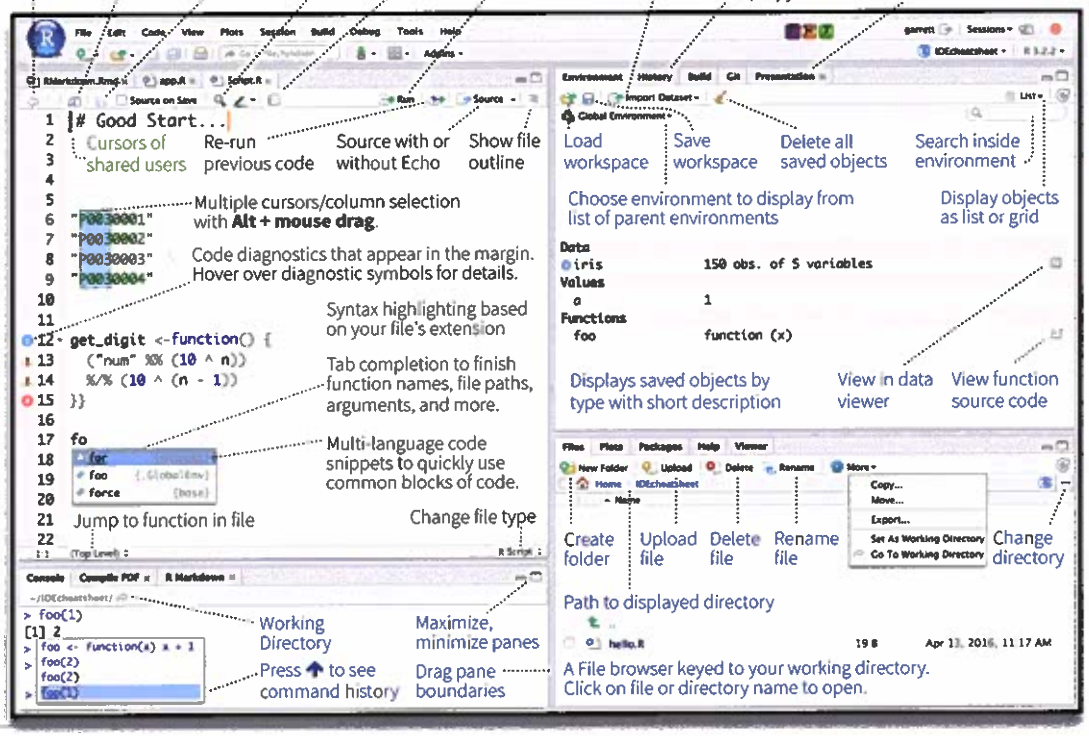
Open Shiny, R Markdown, knitr, Sweave, LaTeX, .Rd files and more in Source Pane




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



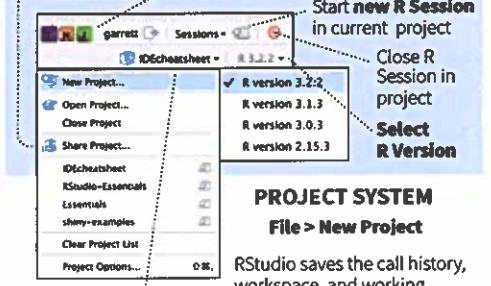

## Write Code




## R Support




## Pro Features




**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.

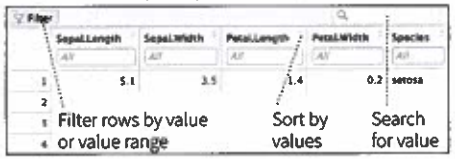
RStudio opens plots in a dedicated Plots pane


GUI Package manager lists every installed package


RStudio opens documentation in a dedicated Help pane

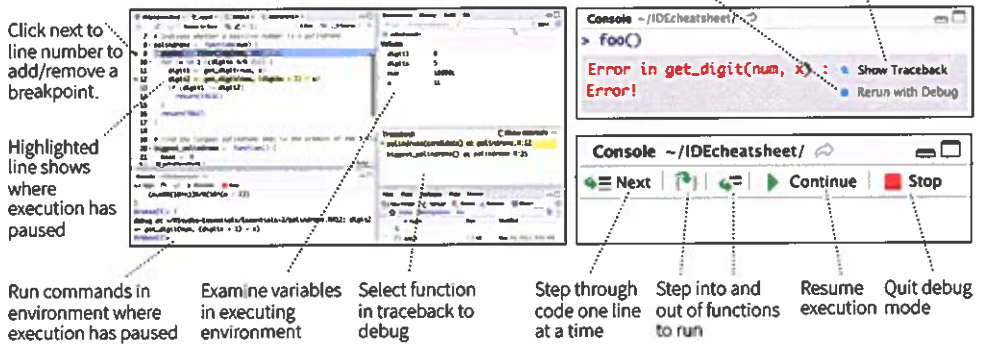

Viewer Pane displays HTML content, such as Shiny apps, RMarkdown reports, and interactive visualizations


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




## Debug Mode

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




## Version Control with Git or SVN

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


## 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**

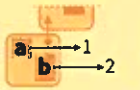
RStudio

# 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.

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

`abs ( 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))*

`e` `a + b` **Expression** - an object that stores quoted code without evaluating it. is_*expression(expr(a + b))*

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

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

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).

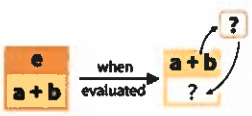
A quosure can be evaluated later in the stored environment to return a predictable result.

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.

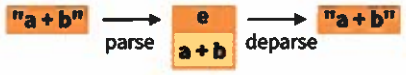
A quoted expression can be evaluated later to return a result that will depend on the environment it is evaluated in

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)*

## 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)*

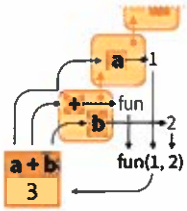
`log` (x = `4`, base = `2` )

`2`

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

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

## Evaluation



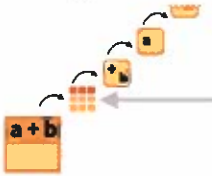

To evaluate an expression, R :

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.

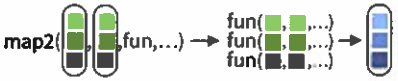
Use the pronoun **.data$** to force a symbol to be matched in data, and **!!** (see back) to force a symbol to be matched in the environments.

# Apply functions with purrr : : **CHEAT SHEET**

## Apply Functions

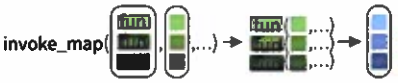
Map functions apply a function iteratively to each element of a list or vector.

**map**(.x, .f, ...) Apply a function to each element of a list or vector. *map(x, is.logical)*



**map2**(.x, .y, .f, ...) Apply a function to pairs of elements from two lists, vectors. *map2(x, y, sum)*

**pmap**(.l, .f, ...) Apply a function to groups of elements from list of lists, vectors. *pmap(list(x, y, z), sum, na.rm = TRUE)*



**invoke_map**(.f, .x = list(NULL), ..., .env=NULL) Run each function in a list. Also **invoke**. *l <- list(var, sd); invoke_map(l, x = 1:9)*

**lmap**(.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.

| function | returns |
|---|---|
| **map** | list |
| **map_chr** | character vector |
| **map_dbl** | double (numeric) vector |
| **map_dfc** | data frame (column bind) |
| **map_dfr** | data frame (row bind) |
| **map_int** | integer vector |
| **map_lgl** | logical vector |
| **walk** | triggers side effects, returns the 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, ~ 2 +.x)* becomes *map(l, function(x) 2 + x )*

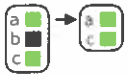
**~ .x .y** becomes **function(.x, .y) .x .y**, e.g. *map2(l, p, ~ .x +.y )* becomes *map2(l, p, function(l, p) l + p )*

**~ ..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)*

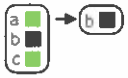
## Work with Lists

### FILTER LISTS

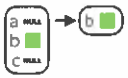
**pluck**(.x, ..., .default=NULL) Select an element by name or index, *pluck(x,"b")* ,or its attribute with **attr_getter**. *pluck(x,"b",attr_getter("n"))*



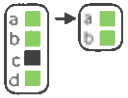
**keep**(.x, .p, ...) Select elements that pass a logical test. *keep(x, is.na)*



**discard**(.x, .p, ...) Select elements that do not pass a logical test. *discard(x, is.na)*

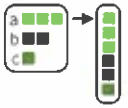


**compact**(.x, .p = identity) Drop empty elements. *compact(x)*

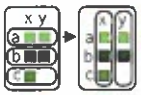


**head_while**(.x, .p, ...) Return head elements until one does not pass. Also **tail_while**. *head_while(x, is.character)*

### RESHAPE LISTS



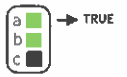
**flatten**(.x) Remove a level of indexes from a list. Also **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. *flatten(x)*



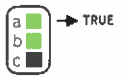
**transpose**(.l, .names = NULL) Transposes the index order in a multi-level list. *transpose(x)*

### SUMMARISE LISTS

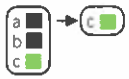
**every**(.x, .p, ...) Do all elements pass a test? *every(x, is.character)*



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

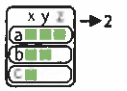


**has_element**(.x, .y) Does a list contain an element? *has_element(x, "foo")*



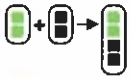
**detect**(.x, .f, ..., .right=FALSE, .p) Find first element to pass. *detect(x, is.character)*

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

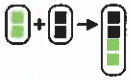


**vec_depth**(x) Return depth (number of levels of indexes). *vec_depth(x)*

### JOIN (TO) 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")*

### TRANSFORM LISTS

**modify**(.x, .f, ...) Apply function to each element. Also **map, map_chr, map_dbl, map_dfc, map_dfr, map_int, map_lgl**. *modify(x, ~.+ 2)*

**modify_at**(.x, .at, .f, ...) Apply function to elements by name or index. Also **map_at**. *modify_at(x, "b", ~.+ 2)*

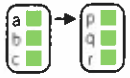
**modify_if**(.x, .p, .f, ...) Apply function to elements that pass a test. Also **map_if**. *modify_if(x, is.numeric,~.+2)*

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

### WORK WITH LISTS

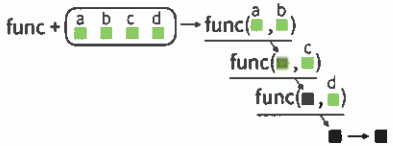
**array_tree**(array, margin = NULL) Turn array into list. Also **array_branch**. *array_tree(x, margin = 3)*

**cross2**(.x, .y, .filter = NULL) All combinations of .x and .y. Also **cross, cross3, cross_df**. *cross2(1:3, 4:6)*

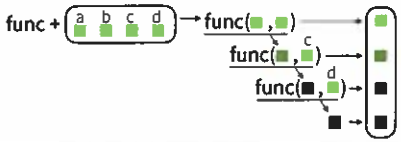


**set_names**(x, nm = x) Set the names of a vector/list directly or with a function. *set_names(x, c("p", "q", "r"))* *set_names(x, tolower)*

## Reduce Lists



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



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

## Modify function behavior

**compose**() Compose multiple functions.

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

**rerun**() Rerun expression n times.

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

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

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

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

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

# 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*
*1,2,3*
*4,5,NA*

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 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")

*a;b;c*
*1;2;3*
*4;5;NA*

**Semi-colon Delimited Files**
**read_csv2("file2.csv")**
write_file(x = "a;b;c\n1;2;3\n4;5;NA", path = "file2.csv")

*a|b|c*
*1|2|3*
*4|5|NA*

**Files with Any Delimiter**
**read_delim("file.txt", delim = "|")**
write_file(x = "a|b|c\n1|2|3\n4|5|NA", path = "file.txt")

*a b c*
*1 2 3*
*4 5 NA*

**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*
*4,5,NA*

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

**No header**
read_csv(f, **col_names = FALSE**)

**Provide header**
read_csv(f, **col_names = c("x", "y", "z")**)

**Skip lines**
read_csv(f, **skip = 1**)

**Read in a subset**
read_csv(f, **n_max = 1**)

**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()))*

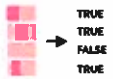
3. Else, read in as character vectors then parse with a parse_ function.
- **parse_guess()**
- **parse_character()**
- **parse_datetime()** Also **parse_date()** and **parse_time()**
- **parse_double()**
- **parse_factor()**
- **parse_integer()**
- **parse_logical()**
- **parse_number()**

*x$A <- parse_number(x$A)*

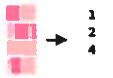
# String manipulation with stringr : : CHEAT SHEET

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

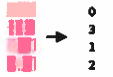
## Detect Matches




**str_detect**(string, **pattern**) Detect the presence of a pattern match in a string.
*str_detect(fruit, "a")*



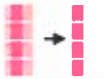

**str_which**(string, **pattern**) Find the indexes of strings that contain a pattern match.
*str_which(fruit, "a")*




**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")*

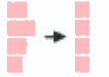

**str_extract**(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*


**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)*

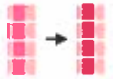
**str_pad**(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. *str_pad(fruit, 17)*



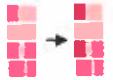
**str_trunc**(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis.
*str_trunc(fruit, 3)*

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

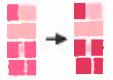
## Mutate Strings



**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", "-")*

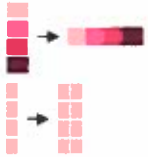

**str_to_lower**(string, locale = "en")[1] Convert strings to lower case.
*str_to_lower(sentences)*


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


**str_to_title**(string, locale = "en")[1] Convert strings to title case. *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, ...)[1] 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, ...)[1] Sort a character vector.
*str_sort(x)*

## Helpers

```
apple
banana
pear

apple
banana
pear
```

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

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

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

**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.

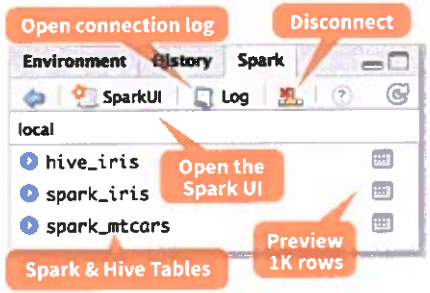
# Data Science in Spark with Sparklyr : : CHEAT SHEET

## 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.

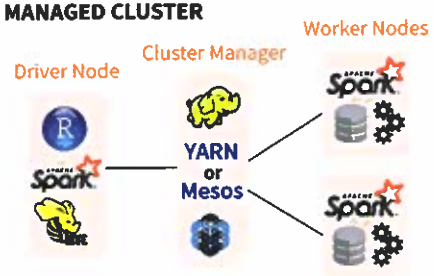
Starting with **version 1.044, RStudio Desktop, Server and Pro include integrated support for the sparklyr package**. You can create and manage connections to Spark clusters and local Spark instances from inside the IDE.

**RStudio Integrates with sparklyr**

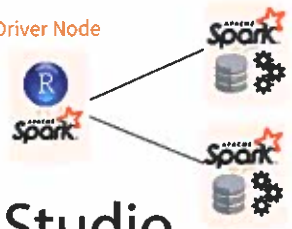



## Cluster Deployment

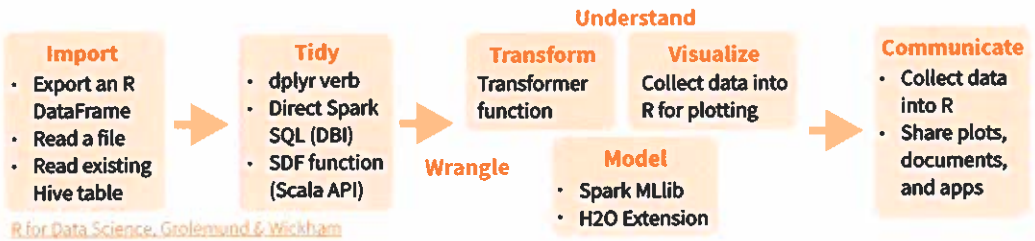
**MANAGED CLUSTER**




**STAND ALONE CLUSTER**




## Data Science Toolchain with Spark + sparklyr




R for Data Science, Grolemund & Wickham

## Getting Started

**LOCAL MODE** (No cluster required)

1. Install a local version of Spark: **spark_install ("2.0.1")**
2. Open a connection **sc <- spark_connect (master = "local")**

**ON A MESOS MANAGED CLUSTER**

1. Install RStudio Server or Pro on one of the existing nodes
2. Locate path to the cluster's Spark directory
3. Open a connection **spark_connect(master="[mesos URL]", version = "1.6.2", spark_home = [Cluster's Spark path])**

**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")**

**ON A YARN MANAGED CLUSTER**

1. Install RStudio Server or RStudio Pro on one of the existing nodes, preferably an edge node
2. Locate path to the cluster's Spark Home Directory, it normally is "/usr/lib/spark"
3. Open a connection **spark_connect(master="yarn-client", version = "1.6.2", spark_home = [Cluster's Spark path])**

**ON A SPARK STANDALONE CLUSTER**

1. Install RStudio Server or RStudio Pro on one of the existing nodes or a server in the same LAN
2. Install a local version of Spark: **spark_install (version = "2.0.1")**
3. Open a connection **spark_connect(master="spark://host:port", version = "2.0.1", spark_home = spark_home_dir())**

## 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)

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

Spark ML Decision Tree Model

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

Create reference to Spark table

```
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