R for Reproducible Scientific Analysis (./): Reference

Key Points

Introduction to R and RStudio (./01-rstudio-intro/index.html)	• Use RStudio to write and run R programs. • R has the usual arithmetic operators and mathematical functions. • Use `<-` to assign values to variables. • Use `ls()` to list the variables in a program. • Use `rm()` to delete objects in a program. • Use `install.packages()` to install packages (libraries).
Project Management With RStudio (./02-project-intro/index.html)	• Use RStudio to create and manage projects with consistent layout. • Treat raw data as read-only. • Treat generated output as disposable. • Separate function definition and application.
Seeking Help (./03-seeking-help/index.html)	• Use `help()` to get online help in R.
Data Structures (./04-data-structures-part1/index.html)	• Use `read.csv` to read tabular data in R. • The basic data types in R are double, integer, complex, logical, and character. • Use factors to represent categories in R.

Exploring Data Frames (./05-data-structures-part2/index.html)	• Use `cbind()` to add a new column to a data frame. • Use `rbind()` to add a new row to a data frame. • Remove rows from a data frame. • Use `na.omit()` to remove rows from a data frame with `NA` values. • Use `levels()` and `as.character()` to explore and manipulate factors. • Use `str()`, `nrow()`, `ncol()`, `dim()`, `colnames()`, `rownames()`, `head()`, and `typeof()` to understand the structure of a data frame. • Read in a csv file using `read.csv()`. • Understand what `length()` of a data frame represents.
Subsetting Data (./06-data-subsetting/index.html)	• Indexing in R starts at 1, not 0. • Access individual values by location using `[]`. • Access slices of data using `[low:high]`. • Access arbitrary sets of data using `[c(...)]`. • Use logical operations and logical vectors to access subsets of data.
Control Flow (./07-control-flow/index.html)	• Use `if` and `else` to make choices. • Use `for` to repeat operations.
Creating Publication-Quality Graphics with ggplot2 (./08-plot-ggplot2/index.html)	• Use `ggplot2` to create plots. • Think about graphics in layers: aesthetics, geometry, statistics, scale transformation, and grouping.
Vectorization (./09-vectorization/index.html)	• Use vectorized operations instead of loops.
Functions Explained (./10-functions/index.html)	• Use `function` to define a new function in R. • Use parameters to pass values into functions. • Use `stopifnot()` to flexibly check function arguments in R. • Load functions into programs using `source()`.
Writing Data (./11-writing-data/index.html)	• Save plots from RStudio using the ‘Export’ button. • Use `write.table` to save tabular data.

Splitting and Combining Data Frames with plyr (./12-plyr/index.html)	• Use the `plyr` package to split data, apply functions to subsets, and combine the results.
Dataframe Manipulation with dplyr (./13-dplyr/index.html)	• Use the `dplyr` package to manipulate dataframes. • Use `select()` to choose variables from a dataframe. • Use `filter()` to choose data based on values. • Use `group_by()` and `summarize()` to work with subsets of data. • Use `mutate()` to create new variables.
Dataframe Manipulation with tidyr (./14-tidyr/index.html)	• Use the `tidyr` package to change the layout of dataframes. • Use `gather()` to go from wide to long format. • Use `spread()` to go from long to wide format.
Producing Reports With knitr (./15-knitr-markdown/index.html)	• Mix reporting written in R Markdown with software written in R. • Specify chunk options to control formatting. • Use `knitr` to convert these documents into PDF and other formats.
Writing Good Software (./16-wrap-up/index.html)	• Keep your project folder structured, organized and tidy. • Document what and why, not how. • Break programs into short single-purpose functions. • Write re-runnable tests. • Don’t repeat yourself. • Don’t repeat yourself. • Be consistent in naming, indentation, and other aspects of style.

Reference

Introduction to R and RStudio (./01-rstudio-intro/)

- Use the escape key to cancel incomplete commands or running code (Ctrl+C) if you’re using R from the shell.
- Basic arithmetic operations follow standard order of precedence:
  - Brackets: `(`, `)`
  - Exponents: `^` or `**`
  - Divide: `/`
  - Multiply: `*`
  - Add: `+`
  - Subtract: `-`

- Scientific notation is available, e.g: `2e-3`
- Anything to the right of a `#` is a comment, R will ignore this!
- Functions are denoted by `function_name()`. Expressions inside the brackets are evaluated before being passed to the function, and functions can be nested.
- Mathematical functions: `exp`, `sin`, `log`, `log10`, `log2` etc.
- Comparison operators: `<`, `<=`, `>`, `>=`, `==`, `!=`
- Use `all.equal` to compare numbers!
- `<-` is the assignment operator. Anything to the right is evaluate, then stored in a variable named to the left.
- `ls` lists all variables and functions you've created
- `rm` can be used to remove them
- When assigning values to function arguments, you *must* use `=`.

Project management with RStudio (./02-project-intro/)

- To create a new project, go to File -> New Project
- Install the `packrat` package to create self-contained projects
- `install.packages` to install packages from CRAN
- `library` to load a package into R
- `packrat::status` to check whether all packages referenced in your scripts have been installed.

Seeking help (./03-seeking-help/)

- To access help for a function type `?function_name` or `help(function_name)`
- Use quotes for special operators e.g. `?"+"`
- Use fuzzy search if you can't remember a name '??search_term'
- CRAN task views (http://cran.at.r-project.org/web/views) are a good starting point.
- Stack Overflow (http://stackoverflow.com/) is a good place to get help with your code.
    - `?dput` will dump data you are working from so others can load it easily.
    - `sessionInfo()` will give details of your setup that others may need for debugging.

Data structures (./04-data-structures-part1/)

Individual values in R must be one of 5 **data types**, multiple values can be grouped in **data structures**.

Data types

- `typeof(object)` gives information about an items data type.
- There are 5 main data types:
    - `?numeric` real (decimal) numbers
    - `?integer` whole numbers only
    - `?character` text
    - `?complex` complex numbers
    - `?logical` TRUE or FALSE values

  Special types:

    - `?NA` missing values
    - `?NaN` "not a number" for undefined values (e.g. `0/0`).
    - `?Inf`, `-Inf` infinity.
    - `?NULL` a data structure that doesn't exist

`NA` can occur in any atomic vector. `NaN`, and `Inf` can only occur in complex, integer or numeric type vectors. Atomic vectors are the building blocks for all other data structures. A `NULL` value will occur in place of an entire data structure (but can occur as list elements).

Basic data structures in R:

- atomic `?vector` (can only contain one type)
- `?list` (containers for other objects)
- `?data.frame` two dimensional objects whose columns can contain different types of data
- `?matrix` two dimensional objects that can contain only one type of data.
- `?factor` vectors that contain predefined categorical data.
- `?array` multi-dimensional objects that can only contain one type of data

Remember that matrices are really atomic vectors underneath the hood, and that data.frames are really lists underneath the hood (this explains some of the weirder behaviour of R).

Vectors (./04-data-structures-part1/)

- `?vector()` All items in a vector must be the same type.
- Items can be converted from one type to another using *coercion*.
- The concatenate function 'c()' will append items to a vector.
- `seq(from=0, to=1, by=1)` will create a sequence of numbers.
- Items in a vector can be named using the `names()` function.

Factors (./04-data-structures-part1/)

- `?factor()` Factors are a data structure designed to store categorical data.
- `levels()` shows the valid values that can be stored in a vector of type factor.

Lists (./04-data-structures-part1/)

- `?list()` Lists are a data structure designed to store data of different types.

Matrices (./04-data-structures-part1/)

- `?matrix()` Matrices are a data structure designed to store 2-dimensional data.

Data Frames (./05-data-structures-part2/)

- `?data.frame` is a key data structure. It is a `list` of `vectors`.
- `cbind()` will add a column (vector) to a data.frame.
- `rbind()` will add a row (list) to a data.frame.

Useful functions for querying data structures:

- `?str` structure, prints out a summary of the whole data structure
- `?typeof` tells you the type inside an atomic vector
- `?class` what is the data structure?
- `?head` print the first `n` elements (rows for two-dimensional objects)
- `?tail` print the last `n` elements (rows for two-dimensional objects)
- `?rownames`, `?colnames`, `?dimnames` retrieve or modify the row names and column names of an object.
- `?names` retrieve or modify the names of an atomic vector or list (or columns of a data.frame).
- `?length` get the number of elements in an atomic vector
- `?nrow`, `?ncol`, `?dim` get the dimensions of a n-dimensional object (Won't work on atomic vectors or lists).

Exploring Data Frames (./05-data-structures-part2/)

- `read.csv` to read in data in a regular structure
  - `sep` argument to specify the separator
    - "," for comma separated
    - "\t" for tab separated
  - Other arguments:
    - `header=TRUE` if there is a header row

Subsetting data (./06-data-subsetting/)

- Elements can be accessed by:
  - Index
  - Name
  - Logical vectors
- `[` single square brackets:
  - *extract* single elements or *subset* vectors
  - e.g. `x[1]` extracts the first item from vector x.
  - *extract* single elements of a list. The returned value will be another `list()`.
  - *extract* columns from a data.frame
- `[` with two arguments to:
  - *extract* rows and/or columns of
    - matrices
    - data.frames
    - e.g. `x[1,2]` will extract the value in row 1, column 2.
    - e.g. `x[2,:]` will extract the entire second column of values.
- `[[` double square brackets to extract items from lists.
- `$` to access columns or list elements by name
- negative indices skip elements

Control flow (./07-control-flow/)

- Use `if` condition to start a conditional statement, `else if` condition to provide additional tests, and `else` to provide a default
- The bodies of the branches of conditional statements must be indented.
- Use `==` to test for equality.
- `X && Y` is only true if both X and Y are `TRUE`.
- `X || Y` is true if either X or Y, or both, are `TRUE`.
- Zero is considered `FALSE`; all other numbers are considered `TRUE`
- Nest loops to operate on multi-dimensional data.

Creating publication quality graphics (./08-plot-ggplot2/)

- figures can be created with the grammar of graphics:
  - `library(ggplot2)`
  - `ggplot` to create the base figure

- `aes`thetics specify the data axes, shape, color, and data size
- `geom`etry functions specify the type of plot, e.g. `point`, `line`, `density`, `box`
- `geom`etry functions also add statistical transforms, e.g. `geom_smooth`
- `scale` functions change the mapping from data to aesthetics
- `facet` functions stratify the figure into panels
- `aes`thetics apply to individual layers, or can be set for the whole plot inside `ggplot`.
- `theme` functions change the overall look of the plot
- order of layers matters!
- `ggsave` to save a figure.

Vectorization (./09-vectorization/)

- Most functions and operations apply to each element of a vector
- `*` applies element-wise to matrices
- `%*%` for true matrix multiplication
- `any()` will return `TRUE` if any element of a vector is `TRUE`
- `all()` will return `TRUE` if *all* elements of a vector are `TRUE`

Functions explained (./10-functions/)

- `?"function"`
- Put code whose parameters change frequently in a function, then call it with different parameter values to customize its behavior.
- The last line of a function is returned, or you can use `return` explicitly
- Any code written in the body of the function will preferably look for variables defined inside the function.
- Document Why, then What, then lastly How (if the code isn't self explanatory)

Writing data (./11-writing-data/)

- `write.table` to write out objects in regular format
- set `quote=FALSE` so that text isn't wrapped in `"` marks

Split-apply-combine (./12-plyr/)

- Use the `xxply` family of functions to apply functions to groups within some data.
- the first letter, `a`rray, `d`ata.frame or `l`ist corresponds to the input data
- the second letter denotes the output data structure
- Anonymous functions (those not assigned a name) are used inside the `plyr` family of functions on groups within data.

Dataframe manipulation with dplyr (./13-dplyr/)

- `library(dplyr)`
- `?select` to extract variables by name.
- `?filter` return rows with matching conditions.
- `?group_by` group data by one of more variables.
- `?summarize` summarize multiple values to a single value.
- `?mutate` add new variables to a data.frame.

- Combine operations using the `?"%>%"` pipe operator.

Dataframe manipulation with tidyr (./14-tidyr/)

- `library(tidyr)`
- '?gather' convert data from *wide* to *long* format.
- '?spread' convert data from *long* to *wide* format.
- '?separate' split a single value into multiple values.
- '?unite' merge multiple values into a single value.

Producing reports with knitr (./15-knitr-markdown/)

- Value of reproducible reports
- Basics of Markdown
- R code chunks
- Chunk options
- Inline R code
- Other output formats

Best practices for writing good code (./16-wrap-up/)

- Program defensively, i.e., assume that errors are going to arise, and write code to detect them when they do.
- Write tests before writing code in order to help determine exactly what that code is supposed to do.
- Know what code is supposed to do before trying to debug it.
- Make it fail every time.
- Make it fail fast.
- Change one thing at a time, and for a reason.
- Keep track of what you've done.
- Be humble

Glossary

argument
: A value given to a function or program when it runs. The term is often used interchangeably (and inconsistently) with parameter.

assign
: To give a value a name by associating a variable with it.

body
: (of a function): the statements that are executed when a function runs.

comment
: A remark in a program that is intended to help human readers understand what is going on, but is ignored by the computer. Comments in Python, R, and the Unix shell start with a `#` character and run to the end of the line; comments in SQL start with `--`, and other languages have other conventions.

comma-separated values
: (CSV) A common textual representation for tables in which the values in each row are separated by commas.

delimiter
: A character or characters used to separate individual values, such as the commas between columns in a CSV file.

documentation
Human-language text written to explain what software does, how it works, or how to use it.

floating-point number
A number containing a fractional part and an exponent. See also: integer.

for loop
A loop that is executed once for each value in some kind of set, list, or range. See also: while loop.

index
A subscript that specifies the location of a single value in a collection, such as a single pixel in an image.

integer
A whole number, such as -12343. See also: floating-point number.

library
In R, the directory(ies) where packages are stored.

package
A collection of R functions, data and compiled code in a well-defined format. Packages are stored in a library and loaded using the library() function.

parameter
A variable named in the function's declaration that is used to hold a value passed into the call. The term is often used interchangeably (and inconsistently) with argument.

return statement
A statement that causes a function to stop executing and return a value to its caller immediately.

sequence
A collection of information that is presented in a specific order.

shape
An array's dimensions, represented as a vector. For example, a 5×3 array's shape is `(5,3)` .

string
Short for "character string", a sequence of zero or more characters.

syntax error
A programming error that occurs when statements are in an order or contain characters not expected by the programming language.

type
The classification of something in a program (for example, the contents of a variable) as a kind of number (e.g. floating-point, integer), string, or something else. In R the command typeof() is used to query a variables type.

while loop
A loop that keeps executing as long as some condition is true. See also: for loop.