A Gentle Introduction to R

Jonathan Whiteley

2023-09-13

Prerequisites

- Access to a copy of the R[1] software
  - i.e., a "binary executable"
  - Go to *www.r-project.org* to get a copy, or ask your system administrator.
- Knowledge of common mathematical operations: arithmetic, logarithms, etc.
- No previous experience with R or programming required.

[1]The R logo (R) is © 2016 The R Foundation and used as-is under the terms of the **CC-BY-SA 4.0** license

Section 1

Welcome

Pop Quiz

We will review these *at the end*, so you can see how much you have learned.

- What does 'CRAN' stand for?
- Why is it named '`R`'?
- How can you use `R` *interactively*?
- How do you find out what a function does & how to use it?
- How do you store values to re-use later?
- True or False: Warnings can be ignored, but an Error means I made a mistake.
- True or False: Error messages will tell me how to fix the problem.

Answer in the chat:

What emoji best describes your current mood or state of mind?

Introductions

- Name
- Pronouns
- Job title, role
- *optional*: a hobby or activity you enjoy?

- Have you used `R` before?
- Have you used a programming language before?

Icebreaker activity

What is this?

1–3 word description, for example:

- "This is grey"
- "This looks uncomfortable"

OR caption this image?

On your turn:

1. Previous person's name
2. Their answer to the question
3. Your name
4. Your answer
5. Name of the person to go next

Figure 1: Caption this image.
© John Speirs/Comedywildlifephoto.com

Learning Objectives

- Get familiar with the  *interface*
- Use technical *terms* for R concepts
- Enter *commands*
  - use R interactively: understand input & output
  - use some common *functions*
- Get familiar with 'R objects'
  - store & retrieve values
- Understand *Errors*, *Warnings*, and *Messages*
- How to get Help

Why is it named 'R'?

1. R started as an *open-source* implementation of the S statistical computing language (S-PLUS)[2]
   - S was created at Bell Laboratories in 1976[3]
   - R was based on the S syntax (mostly v3), but works very differently "under the hood".
2. R was created by **R**oss Ihaka and **R**obert Gentleman — aka "R & R"[4] — at the University of Aukland in the early 1990s.

Read more about the history of R on Wikipedia[5]

[2] https://www.r-project.org/about.html
[3] https://en.wikipedia.org/wiki/S_(programming_language)
[4] https://www.r-project.org/contributors.html
[5] https://en.wikipedia.org/wiki/R_(programming_language)#History

Section 2

Interacting with R (Interface)

The R Interface

- 'base R' has a slightly different interface for each **O**perating **S**ystem (OS)
  - GUI = **G**raphical **U**ser **I**nterface
- R can also run inside of a terminal (no GUI) or other software (different GUI).

Integrated **D**evelopment **E**nvironment (IDE)

- An IDE is like an extra interface layer on top of 'base R'
- IDEs often add convenient tools to make writing code easier (e.g., syntax highlighting), and for developing larger projects with multiple files.
- **RStudio** is one of the most popular cross-platform IDEs for R.
  - RStudio is available in open source (free/libre) and commercial[a] editions.

[a] for organizations not able to use software licensed with AGPL

A quick tour of the 'base R GUI'

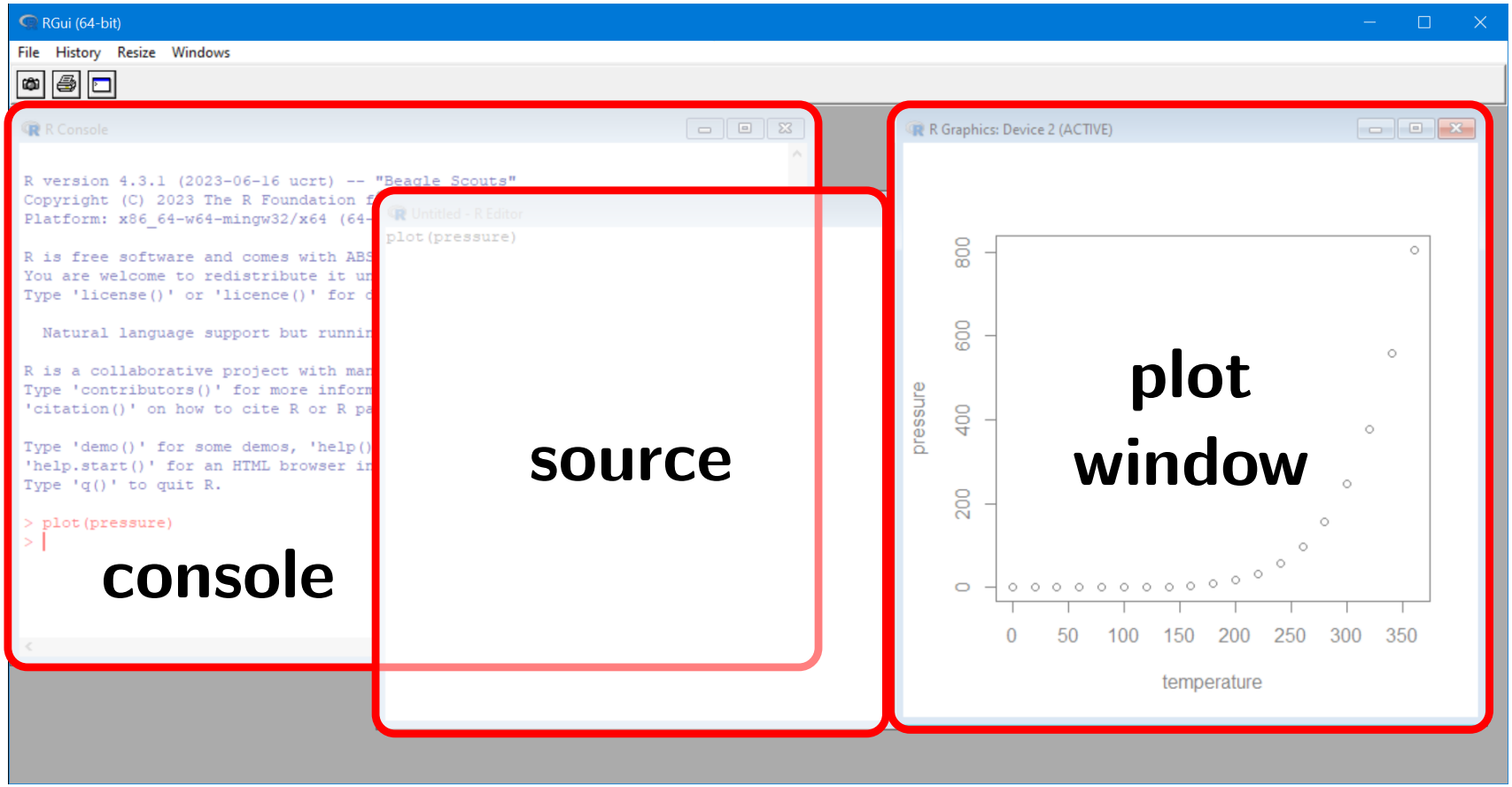


Figure 2: Screenshot of the R GUI in Windows.

A quick tour of RStudio

The RStudio GUI has 4 'panes' that contain 'tabs'.

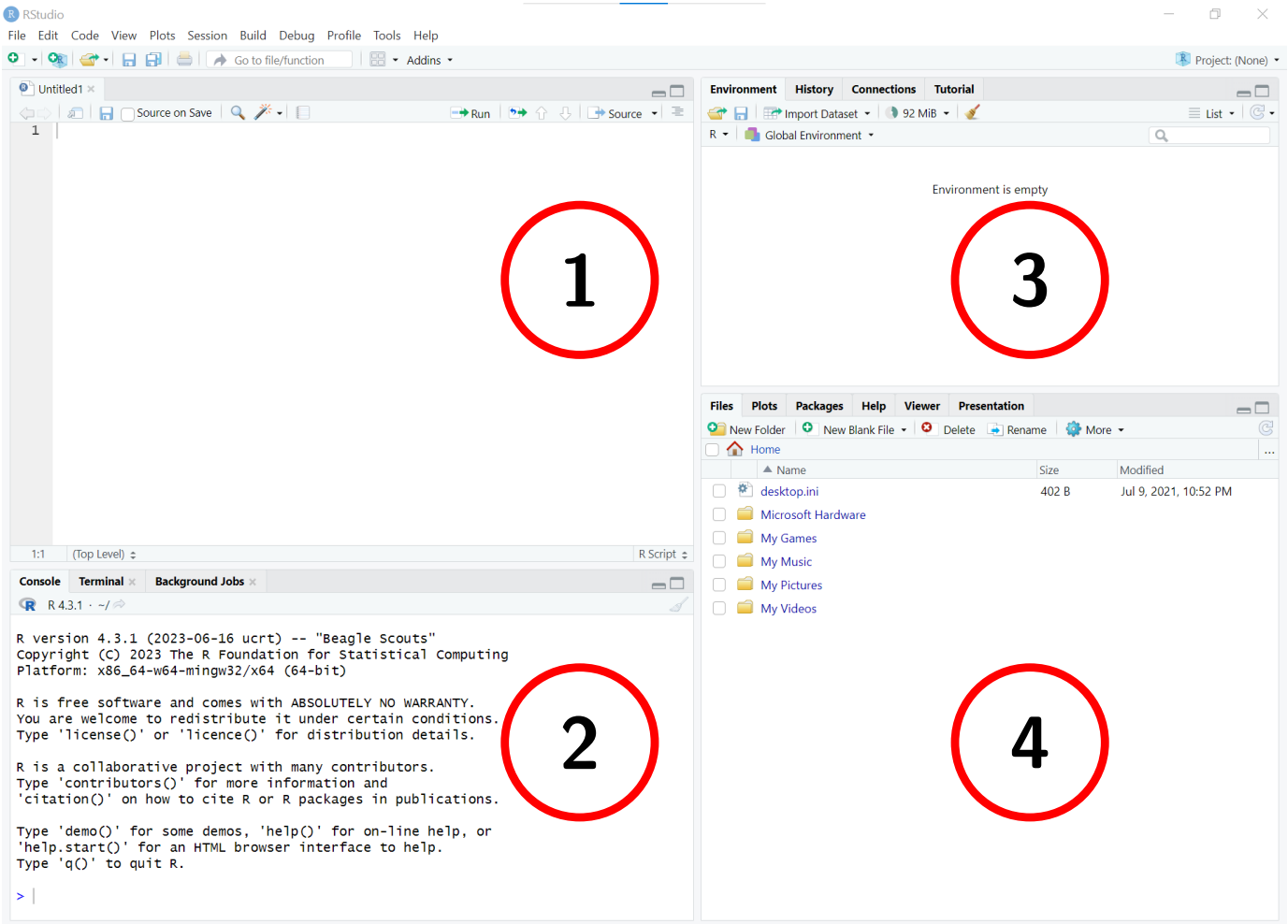


Figure 3: Screenshot of RStudio (default layout).

left:

1. top: **Source**[a]
2. bottom: **Console, Terminal, ...**

right:

3. top: **Environment, History, ...**
4. bottom: **Files, Plots, Help, ...**

[a] empty until you create or open a file

Interacting with R

- Regardless of the GUI, you interact with R primarily using a *command line*
  - aka a command line interface (cli)
  - the command line is usually in the *console*
- "Question-and-Answer Model"
  - You ask R to do something (a *command*), and R tells you the answer (*result*).
- Instructions are given to R using the *R language*.

The R console

The *console* is a window or pane where you will find:

- The *command line*
  - where you will enter commands for `R` to run
- Results of commands and other output
- `Messages`, ***Warnings***, and **Errors**

The R command-line

- The command *prompt* normally looks like this[6]:

```
>
```

  - This is R's way of saying "I am ready to accept new commands".
  - Type a new command on the line after this prompt (i.e., *input*).

- **Press `return`/`enter` to *run* the current *command***
- If you can still edit the command next to the prompt, then it has not been submitted to R to execute (it is still waiting for input).
- If the last prompt is not empty (i.e., there is text beside it) *and* you cannot edit what is beside the prompt, it means R is still running the last command and is not ready to accept a new command yet.
  - Wait for a new empty prompt to appear before entering the next command.

[6] the colour of the prompt varies depending on the interface

The R command-line (continued)

- If the prompt looks like this:

```
+
```

  it means the last command was *incomplete* and R is waiting for more input.
  R will not do anything until the command is completed or cancelled.

  - This usually means you forgot a closing
    quote ", *parenthesis* (, *bracket* [, or *brace* {

- You can *cancel* the current command at any time by pressing escape (esc)

Section 3

Warming up: some early commands

Input & Output

In this presentation,

- *commands* that can be entered in the *command-line* look like this:

```
Input (commands)
```

  - You can try these yourself!

- Expected output (results) look like this:

```
Output (results)
```

R offers suggestions

Read the opening message carefully.

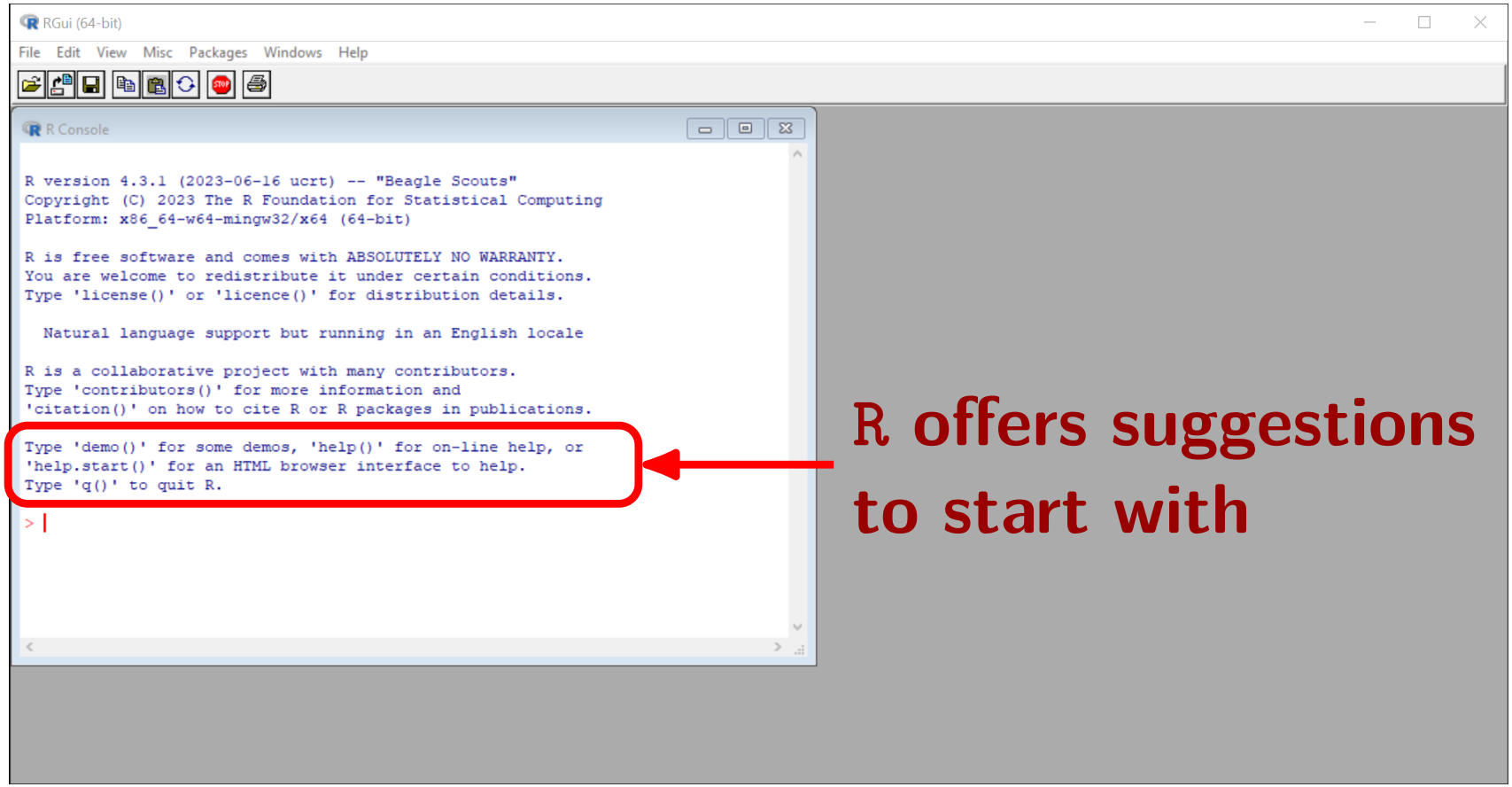


Figure 4: R offers suggestions of commands to Type in the console when it starts.

R is a show-off

`demo(graphics)`	• some plots and graphs that can be made with R
`demo(image)`	• image-like graphics and maps that can be produced with R
`demo(lm.glm)`	• a demonstration of linear modelling & GLMs
`demo()`	• a list of available demos
`help.start()`	← A great place to start, especially if you are comfortable reading documentation for a programming language. *More on this later.*

Note

R will not only show the output, but also *the code used to produce it*.

R is a calculator

```
1 + 1
```

```
[1] 2
```

```
2 * 2
```

```
[1] 4
```

```
2 ^ 3
```

```
[1] 8
```

```
10 - 1
```

```
[1] 9
```

```
8 / 2
```

```
[1] 4
```

```
sqrt(9)
```

```
[1] 3
```

- These are *expressions*
- *Expressions* are *evaluated*, and the *value* (result) is *returned* (sometimes *invisibly*)

command-line tips

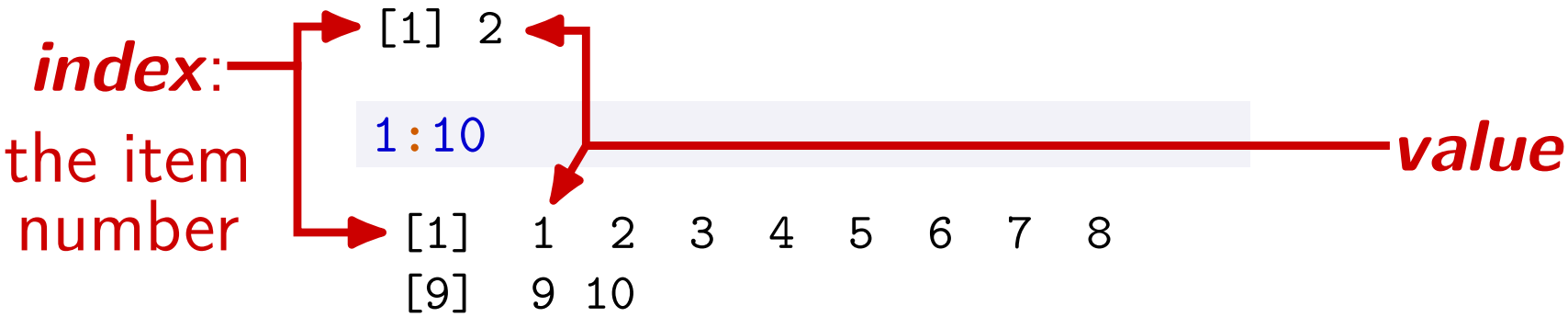
- With the cursor next to the empty prompt (>),
  use the up & down arrow keys (↑↓) to re-produce previous commands.
- This lets you "scroll through your *command history*".
- Press up (↑) once, and you get the last command you entered
  without having to copy & paste.

Section 4

Simple R objects

Vectors

- The most basic kind of *object* in R is a *vector*
- Think of a vector as a list of related values (data), which are *all the same type*
- A single value is an "*atomic vector*" (a vector with a length of 1)



Using vectors

- Vectors can be used in calculations
- Operations are applied to each item (*element-wise*)

```
sum( c(1, 2, 3, 4, 5) )
1:10 + 2
1:5 * 5:1
```

- Vectors can be used to plot data in a graph

```
plot( rnorm(1000) )
hist( rnorm(1000) )
```

Some data *types* (of *atomic vectors*)

numeric

- Includes *integers*, *real* (decimal / *double*), and *complex* numbers.
- `1.23`

character (*string*)

- in single `'` or double `"` quotes.
- `'hello world'`
- `"1.23"`

logical

- `TRUE` or `FALSE`

```
class(1.23)
class('hello')
class("1.23")
class(FALSE)

typeof(1.23)
typeof(1:10)

as.character(c(1,2,NA,4))
```

`as.*()`: converting from one type to another = *coercion*

Section 5

Storing & retrieving values

Symbolic *variables*

- You can store values (*objects*) in symbolic variables (*names*) using an *assignment operator*:

`<-`	assign the *value* on the **right** to the *name* on the **left**

- Names can include:

letters	`a-z A-Z`
numbers	`0-9`
periods	`.`
underscores	`_`

```
A <- 10
B <- 10 * 10
A_log <- log(A)
B.seq <- 1:B

assign('x', 3)
```

- Names *should begin with a* ***letter***.

Retrieve values

When a variable *name* is evaluated, it returns the stored *value*.

```
A
```

```
[1] 10
```

```
B
```

```
[1] 100
```

```
A_log
```

```
[1] 2.303
```

```
x
```

```
[1] 3
```

```
B.seq
```

```
 [1]   1   2   3   4   5   6   7   8   9  10  11  12  13
[14]  14  15  16  17  18  19  20  21  22  23  24  25  26
[27]  27  28  29  30  31  32  33  34  35  36  37  38  39
[40]  40  41  42  43  44  45  46  47  48  49  50  51  52
[53]  53  54  55  56  57  58  59  60  61  62  63  64  65
[66]  66  67  68  69  70  71  72  73  74  75  76  77  78
[79]  79  80  81  82  83  84  85  86  87  88  89  90  91
[92]  92  93  94  95  96  97  98  99 100
```

Built-in variables

Some words and letters already have values in R and should **never be used as variable names**.

```
pi
```

```
[1] 3.142
```

```
version
```

```
... information about
this version of R ...
```

```
letters
```

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

```
LETTERS
```

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

Reserved words

Some words and letters already have special meaning in the R language (*keywords*)[7] and should **never be used as variable names**.

`NA`	"Not Available"	placeholder for unknown or missing values
`NaN`	"Not a Number"	placeholder for *undefined* numeric values
`NULL`	*a special object*	placeholder for missing *objects*
`Inf`	Infiniti	
`TRUE`	Logical value	
`FALSE`	Logical value	
`T`	short for `TRUE`	
`F`	short for `FALSE`	
`c,q,t,C,D,I`	`R` functions	
`diff, df, pt`	`R` functions	

[7] See section 10.3.3 of "The R Language Definition" for a list of *reserved words*.

R is case-sensitive

`R.version`	a variable	`pi`
`R.Version()`	a *function*	*PI*
`letters`	a-z	`NA`
`LETTERS`	A-Z	*na*

Use variables in calculations

```
A +5
```

```
[1] 15
```

```
B/A
```

```
[1] 10
```

```
Weight <- c(60 , 72 , 57 , 90 , 95 , 72 )
Height <- c(1.7, 1.8, 1.6, 1.9, 1.7, 1.9)
BMI <- Weight / Height^2
BMI
```

```
[1] 20.76 22.22 22.27 24.93 32.87 19.94
```

```
plot(Height, Weight)
```

Housekeeping

`ls()`	List all variables you have created
`rm(x)`	Remove the variable 'x' from memory
`rm(list=ls())`	Remove *all variables* from memory (clear memory)

```
pi
pi <- "pie"
pi
rm(pi)
pi
```

Section 6

Operators

Operators

Operators are special symbols that go between two values, to perform an *operation* on both values (the *operands*) and return the *result*.

- For example: `2 * 3` is a way of saying "*multiply* 2 and 3 together"
- Operations are evaluated one pair at a time, according to precedence (*order of operations*).

Arithmetic Operators

The usual math symbols:
`+`, `-`, `*`, `/`, `^`, etc.

Assignment Operators

Assign values to symbolic variables:
`<-`, `->`, `=`, etc.

Comparison (*Relational*) Operators

For comparing two values:
`==`, `!=`, `>`, `<`, etc.

Boolean Operators

Combining logical values (`TRUE`, `FALSE`): `!`, `&`, `|`, etc.

Comparisons

Comparison of 2 values results in *logical values*: `TRUE` or `FALSE`

`==`	"equal" — Note the **two** equals signs. Not to be confused with a single equals sign (used to *assign* values).
`!=`	"not equal"
`>`	"greater than"
`<`	"less than"
`>=`	"greater than or equal to"
`<=`	"less than or equal to"

Comparisons: examples

```
1 == 2
```

```
[1] FALSE
```

```
1 <= 2
```

```
[1] TRUE
```

```
1 < "a"
```

```
[1] TRUE
```

```
1 < 2
```

```
[1] TRUE
```

```
1 != "foo"
```

```
[1] TRUE
```

```
0 == FALSE
```

```
[1] TRUE
```

Comparing decimals ('floating point' arithmetic)

Computers can't represent *all* values accurately, and there is often some rounding that occurs (even at 50+ decimal places).
As a result, 'floating point' values may not be *reliably equal*. [8] [9]

This is a common source of confusion, but it is a fact of how computers handle floating point arithmetic, and not specific to R.

```
a <- sqrt(2)
a * a == 2  # should be TRUE
```

```
[1] FALSE
```

```
a * a - 2
```

```
[1] 4.441e-16
```

Two common solutions:

1. `round()` decimal values when comparing them
2. use a function with a tolerance for small differences, such as `all.equal()`

```
round(a * a, 8) == 2  #(1)
```

```
[1] TRUE
```

```
all.equal(a * a, 2)   #(2)
```

```
[1] TRUE
```

[8] R FAQ: "Why doesn't R think these numbers are equal?"
[9] See Stackoverflow: "Why are these numbers not equal?" for other solutions

Section 7

Functions

Functions

- *Functions* are special commands that can do more than simple operators[10].
- They are the main instructions you give to R.
- To use (or *call*) a function, the command must be structured properly, following the "grammar rules" of the R language (*syntax*).

```
log( 8 , base = 2 )
```

[10] technically, operators are special functions with exactly 1 (*unary*) or 2 (*binary*) *arguments*. See section 3.1.4 "Operators" in the R Language Definition.

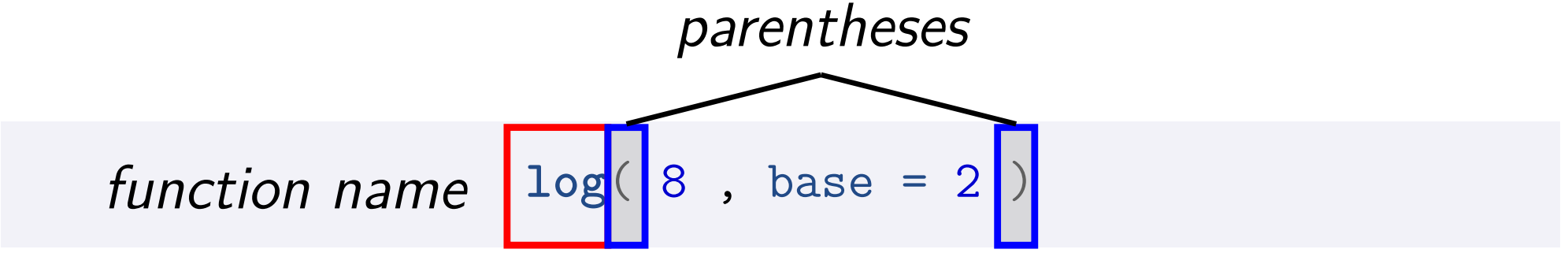
Functions

- *Functions* are special commands that can do more than simple operators[10].
- They are the main instructions you give to R.
- To use (or *call*) a function, the command must be structured properly, following the "grammar rules" of the R language (*syntax*).

function name `log( 8 , base = 2 )`

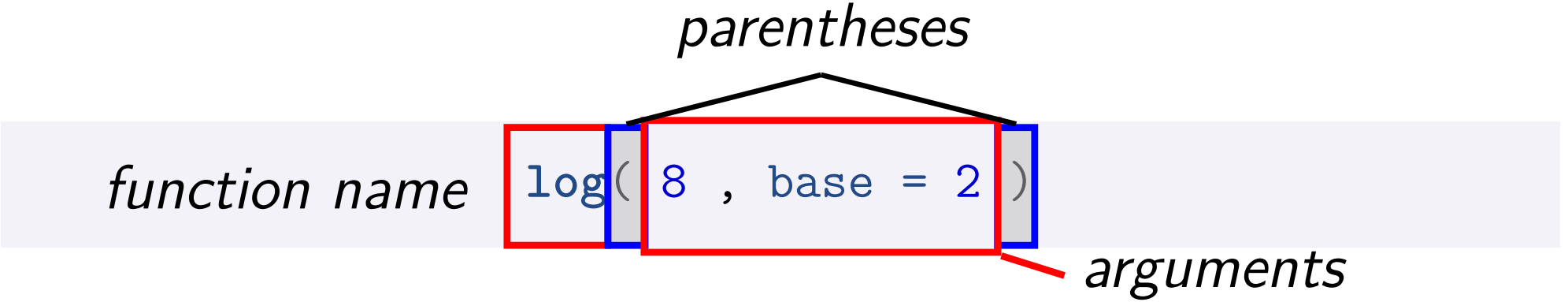
Functions

- *Functions* are special commands that can do more than simple operators[10].
- They are the main instructions you give to R.
- To use (or *call*) a function, the command must be structured properly, following the "grammar rules" of the R language (*syntax*).



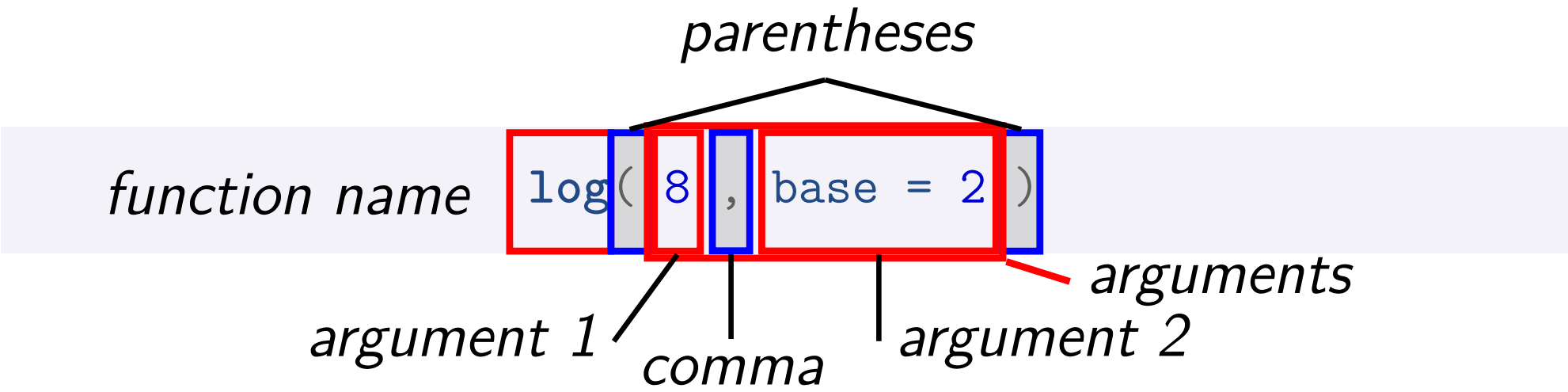
Functions

- *Functions* are special commands that can do more than simple operators[10].
- They are the main instructions you give to R.
- To use (or *call*) a function, the command must be structured properly, following the "grammar rules" of the R language (*syntax*).



Functions

- *Functions* are special commands that can do more than simple operators[10].
- They are the main instructions you give to R.
- To use (or *call*) a function, the command must be structured properly, following the "grammar rules" of the R language (*syntax*).



[10] technically, operators are special functions with exactly 1 (*unary*) or 2 (*binary*) *arguments*. See section 3.1.4 "Operators" in the R Language Definition.

Function arguments

- *arguments* are the values passed to a function when it is *called*
  - these are values the function needs to do its thing
  - some change *how* the function operates (these are usually optional)
- arguments are separated by a comma (`,`)
- arguments can be *passed by order* or *passed by name*
  - *passed by order* means the arguments are specified in the correct order, without a name
  - *passed by name* means the arguments can be in any order, but must be declared by name: `argument = value`

!

Note the **single** equals sign (=), used to assign values to function arguments by name

Calling Functions

- Some functions can be called without arguments.
- **You still need the parentheses() !**
- The same word without () refers to an *object* (*variable*) *name*: adding the () specifies a *function call*
- Typing a function name without brackets usually outputs the *raw code* for that function (unless another object has been defined with the same name).
  - i.e., the *value* of the function object itself.

```
ls()
```

```
[1] "a"
```

```
ls
```

```
function (name, pos = -1L, envir =
    pattern, sorted = TRUE)
{
    if (!missing(name)) {
        pos <- tryCatch(name, erro
        if (inherits(pos, "error")
            name <- substitute(nam
            if (!is.character(name
```

A complex example

```
Var <- sum( ((x <- 1:20) - mean(x))^2 / (length(x) -1) )
```

- Try breaking this up and run each piece one at a time to see all the steps.

A complex example

```
Var <- sum( ((x <- 1:20) - mean(x))^2 / (length(x) -1) )
```

- Try breaking this up and run each piece one at a time to see all the steps.
- The shorter version:

```
var(1:20)
```

```
[1] 35
```

Section 8

Errors, Warnings, and Messages

Errors

- When R receives a command it does not understand, or cannot execute, it outputs an ***error*** to the *console*.
  - This is a message that begins with the word "`Error`".
- A command that produces an *error* is **not** executed.

```
Fail <- 1 + "2"
```

```
Error in 1 + "2" : non-numeric argument to binary operator
```

```
Fail
```

```
Error in eval(expr, envir, enclos) : object 'Fail' not found
```

- Error messages tell you what went wrong, not how to fix it: that's up to you to figure out.
- When an error occurs, `R` **stops running** commands and returns to the command-line.
  - Your *session* is still active: `R` didn't quit, and you can enter more commands.

Warnings

- Some commands still work, but did not run exactly as R (or the developers) think is "ideal", and may produce a ***warning*** instead.
  - This is a message that begins with the word "*Warning*".
- These do not interrupt what R is doing: it will keep running, but tell you that there were warnings.
  - *It is up to you to review the warnings and decide if they are important*.
  - Use the `warnings()` command to review them.

```
oops <- log(-1)
```

```
Warning in log(-1): NaNs produced
```

Errors, Warnings, and Messages

- ***Errors*** indicate something is wrong, and R had to stop. You'll have to figure out what caused the error, fix it, and try again.
  - Think of errors as a red traffic light: stop — something is wrong!
- ***Warnings*** indicate something unusual happened, but R is able to continue. You'll have to assess if it's worth worrying about.
  - Think of warnings as a yellow traffic light: you can go, but be careful and pay attention, in case there is a problem.
- Other ***Messages*** are for information, and a sign that things are working fine (at least, according to the programmers who created the function).
  - Think of messages as a green traffic light: you are safe to continue.

Section 9

Help & documentation

HELP

- R *documentation* (help files)
- Books
- Web sites
- Cheat sheets / Reference cards
- Each Other

HELP: Books

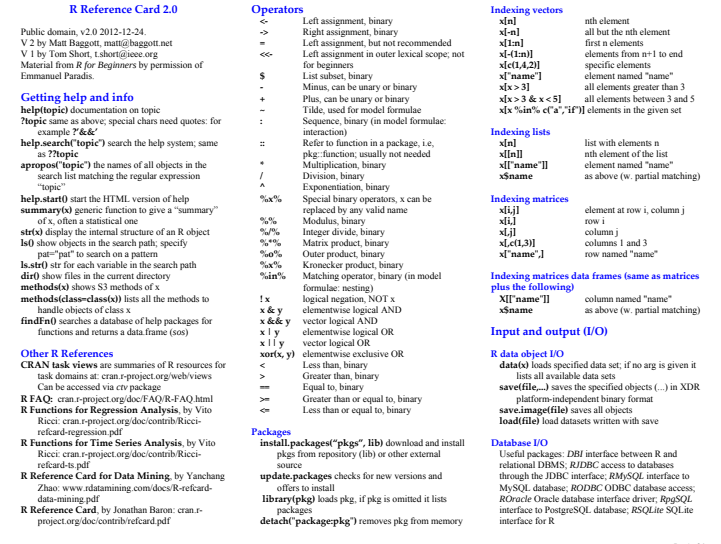
- Springer publishing: "**Use R!**" series
  - Some older: A Beginner's Guide to R (2009)
  - Some more recent: Data Wrangling with R (2016)
  - Some focus on specific methods, e.g.:
    Numerical Ecology with R (2018)
    Applied Spatial Data Analysis with R (2013)
- Other suggestions on the R web site:
  www.r-project.org/doc/bib/R-books.html
- R packages can change quickly: be careful if older content refers to old versions of packages, or packages that are deprecated.
  - Concepts or general methods may still be relevant.
- Many are available in physical or digital formats (or both)

HELP: Web Sites

- R web site: www.r-project.org
  - especially the "Documentation" section
  - e-mail lists
- RStudio Education: education.rstudio.com
- R-bloggers.com www.r-cookbok.com
- Stack Overflow (stackoverflow.com)
  - Q&A site: search for your question, or ask your own
- Cookbook for R (www.cookbook-r.com)
- Your preferred search engine ...

HELP: Reference cards / cheat sheets

- CRAN Contributed Documentation
  - "Short Documents and Reference Cards"
  - https://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf
  - https://cran.r-project.org/doc/contrib/refcard.pdf
- Cheat sheets from RStudio/Posit: posit.co/resources/cheatsheets
  - Contributed Cheatsheets, including non-English translations
  - RStudio IDE cheat sheet
- Search the internet for
  - "R cheat sheet"
  - "R reference card"



R Reference Card 2.0

Public domain, v2.0 2012-12-24.
V 2 by Matt Baggott, matt@baggott.net
V 1 by Tom Short, t.short@ieee.org
Material from *R for Beginners* by permission of Emmanuel Paradis.

Getting help and info
help(topic) documentation on topic
?topic same as above; special chars need quotes: for example **?"&&"**
help.search("topic") search the help system; same as **??topic**
apropos("topic") the names of all objects in the search list matching the regular expression "topic"
help.start() start the HTML version of help
summary(x) generic function to give a "summary" of x, often a statistical one
str(x) display the internal structure of an R object
ls() show objects in the search path; specify pat="pat" to search on a pattern
ls.str() str for each variable in the search path
dir() show files in the current directory
methods(x) shows S3 methods of x
methods(class=class(x)) lists all the methods to handle objects of class x
findFn() searches a database of help packages for functions and returns a data.frame (sos)

Other R References
CRAN task views are summaries of R resources for task domains at: cran.r-project.org/web/views Can be accessed via *ctv* package
R FAQ: cran.r-project.org/doc/FAQ/R-FAQ.html
R Functions for Regression Analysis, by Vito Ricci: cran.r-project.org/doc/contrib/Ricci-refcard-regression.pdf
R Functions for Time Series Analysis, by Vito Ricci: cran.r-project.org/doc/contrib/Ricci-refcard-ts.pdf
R Reference Card for Data Mining, by Yanchang Zhao: www.rdatamining.com/docs/R-refcard-data-mining.pdf
R Reference Card, by Jonathan Baron: cran.r-project.org/doc/contrib/refcard.pdf

Operators
<- Left assignment, binary
-> Right assignment, binary
= Left assignment, but not recommended
<<- Left assignment in outer lexical scope; not for beginners
$ List subset, binary
- Minus, can be unary or binary
+ Plus, can be unary or binary
~ Tilde, used for model formulae
: Sequence, binary (in model formulae: interaction)
:: Refer to function in a package, i.e, pkg::function; usually not needed
***** Multiplication, binary
/ Division, binary
^ Exponentiation, binary
%x% Special binary operators, x can be replaced by any valid name
%% Modulus, binary
%/% Integer divide, binary
%*% Matrix product, binary
%o% Outer product, binary
%x% Kronecker product, binary
%in% Matching operator, binary (in model formulae: nesting)
! x logical negation, NOT x
x & y elementwise logical AND
x && y vector logical AND
x | y elementwise logical OR
x || y vector logical OR
xor(x, y) elementwise exclusive OR
< Less than, binary
> Greater than, binary
== Equal to, binary
>= Greater than or equal to, binary
<= Less than or equal to, binary

Packages
install.packages("pkgs", lib) download and install pkgs from repository (lib) or other external source
update.packages checks for new versions and offers to install
library(pkg) loads pkg, if pkg is omitted it lists packages
detach("package:pkg") removes pkg from memory

Indexing vectors
x[n] nth element
x[-n] all but the nth element
x[1:n] first n elements
x[-(1:n)] elements from n+1 to end
x[c(1,4,2)] specific elements
x["name"] element named "name"
x[x > 3] all elements greater than 3
x[x > 3 & x < 5] all elements between 3 and 5
x[x %in% c("a","if")] elements in the given set

Indexing lists
x[n] list with elements n
x[[n]] nth element of the list
x[["name"]] element named "name"
x$name as above (w. partial matching)

Indexing matrices
x[i,j] element at row i, column j
x[i,] row i
x[,j] column j
x[,c(1,3)] columns 1 and 3
x["name",] row named "name"

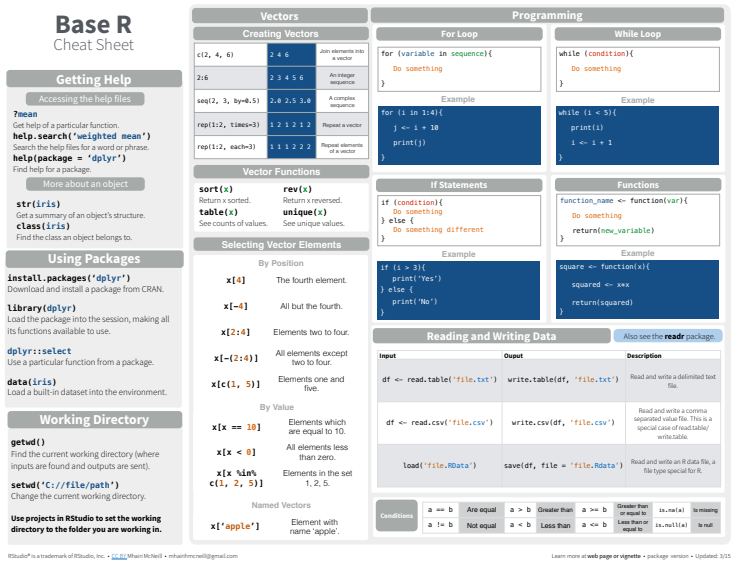
Indexing matrices data frames (same as matrices plus the following)
X[["name"]] column named "name"
x$name as above (w. partial matching)

Input and output (I/O)

R data object I/O
data(x) loads specified data set; if no arg is given it lists all available data sets
save(file,...) saves the specified objects (...) in XDR platform-independent binary format
save.image(file) saves all objects
load(file) load datasets written with save

Database I/O
Useful packages: *DBI* interface between R and relational DBMS; *RJDBC* access to databases through the JDBC interface; *RMySQL* interface to MySQL database; *RODBC* ODBC database access; *ROracle* Oracle database interface driver; *RpgSQL* interface to PostgreSQL database; *RSQLite* SQLite interface for R

R p 1 of 6



Base R
Cheat Sheet

Getting Help
Accessing the help files
`?mean` Get help of a particular function.
`help.search('weighted mean')` Search the help files for a word or phrase.
`help(package = 'dplyr')` Find help for a package.
More about an object
`str(iris)` Get a summary of an object's structure.
`class(iris)` Find the class an object belongs to.

Using Packages
`install.packages('dplyr')` Download and install a package from CRAN.
`library(dplyr)` Load the package into the session, making all its functions available to use.
`dplyr::select` Use a particular function from a package.
`data(iris)` Load a built-in dataset into the environment.

Working Directory
`getwd()` Find the current working directory (where inputs are found and outputs are sent).
`setwd('C://file/path')` Change the current working directory.
Use projects in RStudio to set the working directory to the folder you are working in.

Vectors
Creating Vectors
`c(2, 4, 6)` 2 4 6 — Join elements into a vector
`2:6` 2 3 4 5 6 — An integer sequence
`seq(2, 3, by=0.5)` 2.0 2.5 3.0 — A complex sequence
`rep(1:2, times=3)` 1 2 1 2 1 2 — Repeat a vector
`rep(1:2, each=3)` 1 1 1 2 2 2 — Repeat elements of a vector

Vector Functions
`sort(x)` Return x sorted.
`rev(x)` Return x reversed.
`table(x)` See counts of values.
`unique(x)` See unique values.

Selecting Vector Elements
By Position
`x[4]` The fourth element.
`x[-4]` All but the fourth.
`x[2:4]` Elements two to four.
`x[-(2:4)]` All elements except two to four.
`x[c(1, 5)]` Elements one and five.
By Value
`x[x == 10]` Elements which are equal to 10.
`x[x < 0]` All elements less than zero.
`x[x %in% c(1, 2, 5)]` Elements in the set 1, 2, 5.
Named Vectors
`x['apple']` Element with name 'apple'.

Programming
For Loop
```
for (variable in sequence){
  Do something
}
```
Example
```
for (i in 1:4){
  j <- i + 10
  print(j)
}
```
While Loop
```
while (condition){
  Do something
}
```
Example
```
while (i < 5){
  print(i)
  i <- i + 1
}
```
If Statements
```
if (condition){
  Do something
} else {
  Do something different
}
```
Example
```
if (i > 3){
  print('Yes')
} else {
  print('No')
}
```
Functions
```
function_name <- function(var){
  Do something
  return(new_variable)
}
```
Example
```
square <- function(x){
  squared <- x*x
  return(squared)
}
```
Reading and Writing Data — Also see the **readr** package.

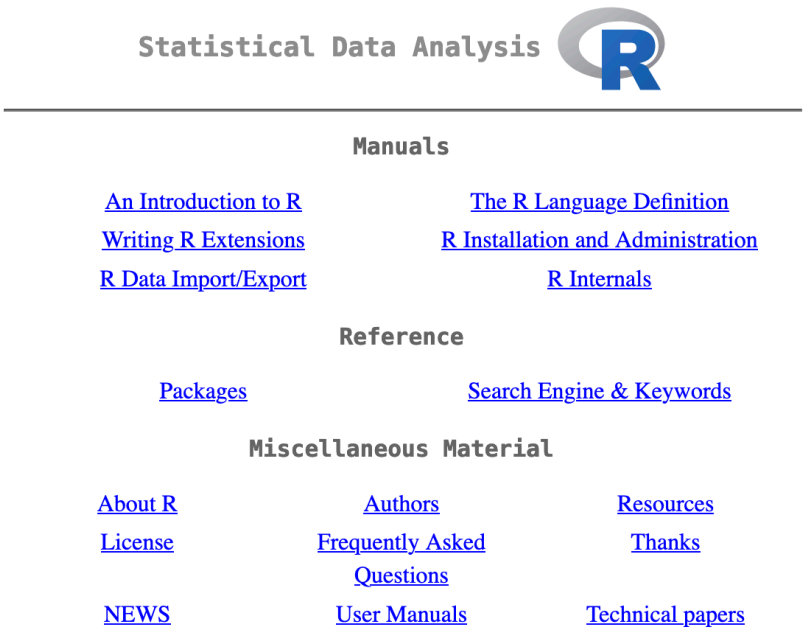
Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

Conditions: `a == b` Are equal; `a > b` Greater than; `a >= b` Greater than or equal to; `is.na(a)` Is missing
`a != b` Not equal; `a < b` Less than; `a <= b` Less than or equal to; `is.null(a)` Is null

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com
Learn more at **web page or vignette** • package version • Updated: 3/15

R Documentation

```
help.start()
```



- A great place to start
- HTML documentation with tutorials, concepts, and examples.
- Browse or search for something specific, or just explore.
- Click on "packages" to see a list of installed packages,
  - documentation about each package (e.g., "vignettes"),
  - functions included in a package

R Documentation: find it

```
?help
```

- Documentation about documentation, and how to search it

```
?c
help.search("c")
```

- read about the often-used 'combine' *function*

```
?seq
```

- read about a function for making a *sequence*

```
?help.search
help.search("t-test")
??'t-test'
```

- use `help.search("")` or `??` to search for a term when you don't know the name of the function, but you know what you want to do.

R Documentation: find it

```
?help
```

- Documentation about documentation, and how to search it

```
?c
help.search("c")
```

- read about the often-used 'combine' *function*

```
?seq
```

- read about a function for making a *sequence*

```
?help.search
help.search("t-test")
??'t-test'
```

- use `help.search("")` or `??` to search for a term when you don't know the name of the function, but you know what you want to do.

function name (or topic) → seq {base} ← package

R Documentation

Sequence Generation — title

Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

arguments

default values

arguments followed by '=' have a *default value*: if you don't include these in your function call, they are automatically assigned the default value shown here, after the '='.

Arguments

`...` arguments passed to or from methods.

`from, to` the starting and (maximal) end values of the sequence. Of length `1` unless just `from` is supplied as an unnamed argument.

`by` number: increment of the sequence.

`length.out` desired length of the sequence. A non-negative number, which for `seq` and `seq.int` will be rounded up if fractional.

`along.with` take the length from the length of this argument.

details on how the function works

Details

Numerical inputs should all be finite (that is, not infinite, `NaN` or `NA`).

The interpretation of the unnamed arguments of `seq` and `seq.int` is *not* standard, and it is recommended always to name the

details

The fourth form generates the integer sequence `1, 2, ..., length(along.with)`. (`along.with` is usually abbreviated to `along`, and `seq_along` is much faster.)

The fifth form generates the sequence `1, 2, ..., length(from)` (as if argument `along.with` had been specified), *unless* the argument is numeric of length 1 when it is interpreted as `1:from` (even for `seq(0)` for compatibility with S). Using either `seq_along` or `seq_len` is much preferred (unless strict S compatibility is essential).

The final form generates the integer sequence `1, 2, ..., length.out` unless `length.out = 0`, when it generates `integer(0)`.

Very small sequences (with `from - to` of the order of 10^{-14} times the larger of the ends) will return `from`.

For `seq` (only), up to two of `from`, `to` and `by` can be supplied as complex values provided `length.out` or `along.with` is specified. More generally, the default method of `seq` will handle classed objects with methods for the `Math`, `Ops` and `Summary` group generics.

`seq.int`, `seq_along` and `seq_len` are primitive.

***value** returned*

Value

`seq.int` and the default method of `seq` for numeric arguments return a vector of type `"integer"` or `"double"`: programmers should not rely on which.

`seq_along` and `seq_len` return an integer vector, unless it is a *long vector* when it will be double.

publications that describe the function or algorithm (theory & concepts)

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

The methods `seq.Date` and `seq.POSIXt`.

`:`, `rep`, `sequence`, `row`, `col`.

Examples

Run examples

```
seq(0, 1, length.out = 11)
seq(stats::rnorm(20)) # effectively 'along'
seq(1, 9, by = 2)     # matches 'end'
seq(1, 9, by = pi)    # stays below 'end'
seq(1, 6, by = 3)
seq(1.575, 5.125, by = 0.05)
seq(17) # same as 1:17, or even better seq_len(17)
```

Copy & paste **Examples** into the console to try them out. Try changing the example code to do what you want.

You can also click the "Run examples" link or use '`example(seq)`' in the console to run all the example code in this section.

[Package *base* version 4.3.1 Index]

Help: example

- Create an unsorted vector of numbers
- Find out how to sort it

```
unsorted_vector <- c(1, 6, -2, 9.5, 4)
help.search("sort")
```

Help: example

- Create an unsorted vector of numbers
- Find out how to sort it

```
unsorted_vector <- c(1, 6, -2, 9.5, 4)
help.search("sort")
```

- Now try including a character string in the vector
  - Sort again
- Try to sort it in reverse order

Section 10

Working with objects

Some *object* types

An "object" is a way of packaging information in R.

vector

- a collection of *values*, all of the **same** *type*.

list

- a collection of **different** *types* of values, or even *objects*.

factor

- a collection of values (*vector*)
  from a finite list of possible values (*levels*)

data frame

- a list of *vectors*, *factors*, or other objects of the same length (# rows)
- *columns* = "variables" ; *rows* = "cases"

Working with *objects*

the object *class*	`class(pi)`
the object *type*	`typeof(pi)`
the object's *structure*	`str(pi)`
the object's *attributes*	`attributes(pi)`
is it a *vector*?	`is.vector(pi)`

Replace 'pi' in the above statements with one of these examples

```
letters
1:10
version
```

Working with a *data frame*

load a built-in data file	`data(CO2)`
peek at the first few rows	`head(CO2)`
the object's ***str****ucture*	`str(CO2)`
names of items in the object	`names(CO2)`
the object *class*	`class(CO2)`
the object *type*	`typeof(CO2)`
the object's *attributes*	`attributes(CO2)`
summary statistics	`summary(CO2)`
plot of all variable combinations	`plot(CO2)`

Structure of a *data frame*

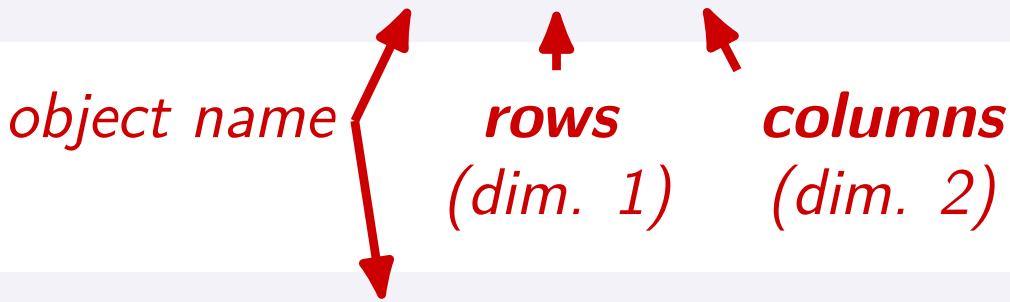
```
str(CO2)
```

```
Classes 'nfnGroupedData', 'nfGroupedData', 'groupedData' and 'data.frame':  84 obs. of  5 variables:
 $ Plant    : Ord.factor w/ 12 levels "Qn1"<"Qn2"<"Qn3"<..: 1 1 1 1 1 1 1 2 2 2 ...
 $ Type     : Factor w/ 2 levels "Quebec","Mississippi": 1 1 1 1 1 1 1 1 1 1 ...
 $ Treatment: Factor w/ 2 levels "nonchilled","chilled": 1 1 1 1 1 1 1 1 1 1 ...
 $ conc     : num  95 175 250 350 500 675 1000 95 175 250 ...
 $ uptake   : num  16 30.4 34.8 37.2 35.3 39.2 39.7 13.6 27.3 37.1 ...
 - attr(*, "formula")=Class 'formula'  language uptake ~ conc | Plant
  .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
 - attr(*, "outer")=Class 'formula'  language ~Treatment * Type
  .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
 - attr(*, "labels")=List of 2
  ..$ x: chr "Ambient carbon dioxide concentration"
  ..$ y: chr "CO2 uptake rate"
 - attr(*, "units")=List of 2
  ..$ x: chr "(uL/L)"
  ..$ y: chr "(umol/m^2 s)"
```

Indexing

- You can refer to parts of an object by their *index* or *name* (if they have one)

```
CO2[1:6, 3]
```




```
CO2$Treatment
```

Indexing examples

```
names(CO2)
```
- available *names*

```
CO2$Treatment
CO2[["Treatment"]]
```
- "Treatment" column (name)

```
CO2[,3]
```
- all rows, column 3

```
CO2[3,]
```
- row 3, all columns

```
CO2[3]
```
- element 3

```
CO2[1:6,]
```
- rows 1-6, all columns

```
CO2[c(1,2,3,4,5,6),3]
```
- rows 1-6, column 3

```
CO2$Treatment[1:6]
```
- elements 1-6 of Treatment column

```
CO2[CO2$conc>100,]
```
- rows where conc $>$ 100

```
CO2[CO2$Treatment=="chilled",]
```
- rows where Treatment $==$ "chilled"

```
CO2[sample(nrow(CO2), 10),]
```
- 10 random rows

Section 11

Installing packages

Intalling *packages*

- R is extensible & modular
  - you can add features (*functions*) and capabilities by *installing* and *loading* extra components called "*packages*"
- You can do this within R, using simple functions
- This keeps R lean, fast, and avoids "feature bloat": load only the packages you need.

Try a function that does not exist (yet)

```
?recode
recode(CO2$Type, "'Quebec'='QC'; else='MS'")
```

```
Error in recode(CO2$Type, "'Quebec'='QC'; else='MS'") :
  could not find function "recode"
```

- This often happens when you forget to *load* a *package* that contains the *function* you are trying to use.
- or there's a typo in the *function name*

Trying to load a *package* that is not installed

- If you try to load a package that is not installed, you will get an **Error**

```
library(car)
```

```
Error in library(car) : there is no package called 'car'
```

- This often happens when you run R code written on a different computer, or written by someone else (on a different computer).
- You have to *install* the package before you can *load* it.

Installing the 'car' package

- car = "Companion to Applied Regression"
  - includes the `recode()` function

```
install.packages("car")
```

- Download & install the package
- You may have to select a "CRAN mirror"[11] (server) to use.

You only need to do this *once* per R installation.

```
library(car)
```

- *Load* the package, to make all contained *functions*, data, and *documentation* available.

This is all you need to do when you want to use it.

```
help(package="car")
```

- Information about your new package.
- See also `help.start()`, and click on the "Packages" link.

---

[11] Comprehensive R Archive Network: repositories of R software, packages and

Try out your new package

```
?recode
recode(CO2$Type, "'Quebec'='QC'; else='MS'")
```

```
 [1] QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC
[19] QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC QC
[37] QC QC QC QC QC QC MS MS MS MS MS MS MS MS MS MS MS MS
[55] MS MS MS MS MS MS MS MS MS MS MS MS MS MS MS MS MS MS
[73] MS MS MS MS MS MS MS MS MS MS MS MS
Levels: MS QC
```

Section 12

Review

Let's play "**Command-R**"

- 2 (or more) players
  - If you have 2 computers, use one to play and the other to consult the help files.
- **Start with:**

```
x <- 0
```

- Take turns using the variable 'x' as an *argument* in a *function* or *expression*
- Assign the *result* to the same variable 'x'
- You may not use the same command twice in a chain
- You may use the same *function* or *mathematical operator*, but you have to use more, fewer, or different *arguments* each time.
- How long can you keep the chain going without getting errors or breaking the rules?

"Command-R"

Challenges

- Change the object *type* of `x` into a :
  - vector of multiple items
  - data frame
- Use `x` in a graph or `plot()`
  - Careful! Some plotting functions do not return a result and may break your chain.

Sample commands

```
x <- x + 1
x <- x * (x+10)
x <- log(x)
x <- exp(x)
x <- 1:x
x <- seq(from=x[1], to=100,
         by=2)
x <- rnorm(x)
x <- x[1:3]
x <- x[2]
x <- data.frame(
  foo = rnorm(length(x)),
  x
)
```

Today's Commands

```
+ - * / ^          min       max       ls    rm
<- =      assign   sum       mean      install.packages
== != < > <= >=    rnorm     var       library
:                  summary
seq                                    plot
c                  head                hist
                   length
round              str                 demo
all.equal          names               example

sqrt               typeof              help          ?
log                class               help.search   ??
exp                attributes          help.start
                   is.vector
```

Section 13

Saving code (files)

Scripts

An `R` *script* is a file that stores `R` code in *plain text*

- They have a `.R` file extension
- They are plain text files
  - so any text editor can read & write them
  - they also work well with **v**ersion **c**ontrol **s**ystems, like `git`, GitHub, and GitLab)
- All the code in a script can be run in order
  - i.e., a *program*
- They make it easy to re-use code
- Scripts provide a record of the steps in a program or analysis
  - results are more *reproducible*
  - scripts are a form of documentation

Note

There is a script file containing all the code shown in the slides for this workshop: '`R-intro.R`' in the '`source`' folder of the workshop files.

Run R code in scripts

Most IDEs have a shortcut to send portions of R code (a line or statement over multiple lines) to an R session:

- R GUI: `CTL+Return` (mac: `CMD+Return`)
- RStudio: `CTL+Return` (mac: `CMD+Return`)

You can run *all* the code in a script in different ways:

- The `source()` function, with a path to the script file as an argument
  - The code will run in the current session.
  - `?source`
- Run R in "*batch mode*"
  - "*batch mode*" is **not** interactive (no prompt)
  - It is usually invoked from a terminal or other command-line (outside an R GUI)
  - The code in the script will run in a new session
  - You can capture output in a separate file
  - `?BATCH`

R Markdown

- **R Markdown** is another file type that can store reusable R code.
- R Markdown combines R code with text and other content, written in *Markdown*.
- Markdown is a simple plain text format that is both human-readable and machine-readable.
  - Markdown can be easily converted to formatted document types (especially ones based on a mark-up language), including HTML (web pages), pdf, or even Word and PowerPoint files.
- Combining R code and other content supports reproducible research, and can allow for automated reports.
- "R Markdown: The Definitive Guide" contains much more information about R Markdown, itself written in R Markdown.

R Markdown & RStudio

- R Markdown files have a '`.rmd`' file extenstion, but they are also a plain text format.
  - They can be edited in any text editor, and compiled using the `rmarkdown` package in any flavour of `R` or IDE.
- RStudio makes working with R Markdown files easy, with features like a 'visual' editing mode, and built-in tools to compile (*render*) documents.

Notes

The slides for this workshop were also made using R Markdown!
You can find the `.rmd` file in the `'source'` folder.

R Notebooks

- RStudio lets you use any R Markdown file like a *notebooks*, where you can run code chunks and see the output, either in the document or in your console.
  - But the results will not be saved if you close the file.
- Results and output can be saved by producing an "html_notebook" output, which can be shared along with the rmd file for reproducible research and analysis.
- More information on R Notebooks is available in the Notebook chapter of the R Markdown book.

Notes

1. You can open the "`.rmd`" file used to produce this document in RStudio, and use it as an 'R Notebook' to run each 'code chunk' and see the results, without having to copy & paste code into the console.
2. We'll explore script files and notebooks more in the next workshop.

Section 14

Backmatter

Quiz Review

- What does 'CRAN' stand for?
- Why is it named '`R`'?
- How can you use `R` *interactively*?
- How do you find out what a function does & how to use it?
- How do you store values to re-use later?
- True or False: Warnings can be ignored, but an Error means I made a mistake.
- True or False: Error messages will tell me how to fix the problem.

References & More Information

```
help.start()
```

Accessible from the screen above (offline):

- An Introduction to R
- The R Language Definition

Online:

- RStudio Education (education.rstudio.com)
-  Manuals (https://cran.r-project.org/manuals.html)
-  Contributed Documentation
  - e.g., http://cran.r-project.org/doc/contrib/usingR.pdf
- Stack Overflow (stackoverflow.com)

Books (now old, but helped me learn):

- A Beginner's Guide to R
- Introductory Statistics with R