

# A Gentle Introduction to R

Jonathan Whiteley

2023-08-16

# Debugging

```
\baselineskip: 12.0pt  
\parskip: 6.8ptplus2.0pt  
\itemsep: 0.0pt  
\partopsep: 0.0pt
```

```
\lineskip: 1.0pt  
\parsep: 6.0ptplus2.0ptminus2.0pt  
\topsep: 0.0pt  
\OuterFrameSep: 4.0pt
```

```
\baselineskip: 12.0pt  
\parskip: 0.0pt  
\parsep: 6.0ptplus2.0ptminus2.0pt  
\itemsep: 0.0pt  
\topsep: 0.0pt  
\partopsep: 0.0pt  
\OuterFrameSep: 4.0pt
```

```
baselineskip: 12.0pt  
parskip: 0.0pt  
parsep: 6.0pt plus 2.0pt minus 2.0pt  
itemsep: 0.0pt  
topsep: 0.0pt  
partopsep: 0.0pt  
OuterFrameSep: 4.0pt
```

pandoc version: 3.1.1

knitr version: 1.43

rmarkdown version: 2.23

# Prerequisites

- Access to a copy of the <sup>1</sup> 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.

---

<sup>1</sup>The R logo () 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/Comedywildlifephotography.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)<sup>2</sup>
  - ▶ S was created at Bell Laboratories in 1976<sup>3</sup>
  - ▶ R was based on the S syntax (mostly v3), but works very differently “under the hood”.
- 2 R was created by Ross Ihaka and Robert Gentleman — aka “R & R”<sup>4</sup> — at the University of Auckland in the early 1990s.

*Read more about the history of R on Wikipedia*<sup>5</sup>

---

<sup>2</sup><https://www.r-project.org/about.html>

<sup>3</sup>[https://en.wikipedia.org/wiki/S\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/S_(programming_language))

<sup>4</sup><https://www.r-project.org/contributors.html>

<sup>5</sup>[https://en.wikipedia.org/wiki/R\\_\(programming\\_language\)#History](https://en.wikipedia.org/wiki/R_(programming_language)#History)

## Section 2

### Interacting with R (Interface)

# The 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<sup>a</sup> editions.

---

<sup>a</sup>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'.



left:

- ① top: **Source**<sup>a</sup>
- ② bottom: **Console, Terminal, ...**

right:

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

<sup>a</sup>empty until you create or open a file

Figure 3: Screenshot of RStudio (default layout).

- 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 *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 command-line

- The command *prompt* normally looks like this<sup>6</sup>:

```
>
```

- ▶ 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.

---

<sup>6</sup>the colour of the prompt varies depending on the interface



# The 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.

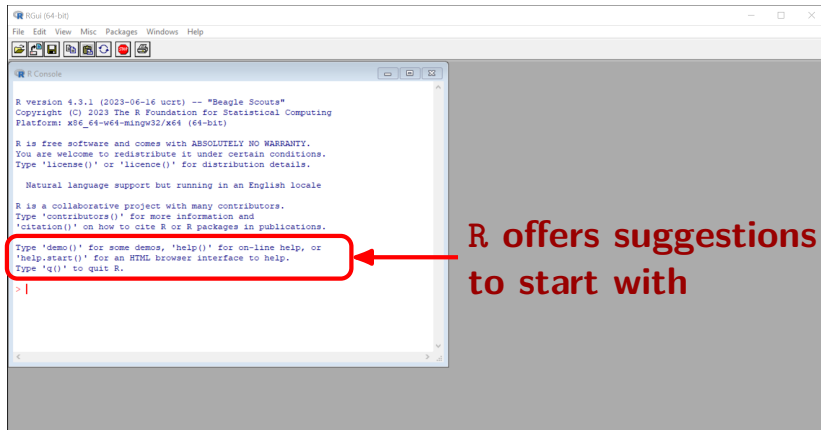


Figure 4: R offers suggestions of commands to **Type** in the console when it starts.

`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*.

# 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*)

- 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.

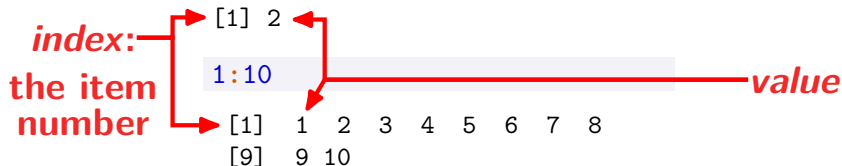
## Section 4

### 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*) and should **never be used as variable names**.

NA	“Not Available”	placeholder for unknown or missing values
NaN	“Not a Number”	placeholder for <i>undefined</i> numeric values
NULL	<i>a special object</i>	placeholder for missing <i>objects</i>
Inf	Infinity	
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	



# is case-sensitive

---

<code>R.version</code>	a variable	<code>pi</code>
------------------------	------------	-----------------

<code>R.Version()</code>	a <i>function</i>	<i>PI</i>
--------------------------	-------------------	-----------

<code>letters</code>	a-z	<code>NA</code>
----------------------	-----	-----------------

<code>LETTERS</code>	A-Z	<i>na</i>
----------------------	-----	-----------

---

# 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**

---

<b>==</b>	“equal” — Note the <b>two</b> equals signs. Not to be confused with a single equals sign (used to <i>assign</i> values).
<b>!=</b>	“not equal”
<b>&gt;</b>	“greater than”
<b>&lt;</b>	“less than”
<b>&gt;=</b>	“greater than or equal to”
<b>&lt;=</b>	“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*.<sup>7 8</sup>

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
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

Two common solutions:

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

---

<sup>7</sup>R FAQ: "Why doesn't R think these numbers are equal?"

<sup>8</sup>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<sup>9</sup>.
- 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 )
```

---

<sup>9</sup>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<sup>9</sup>.
- 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<sup>9</sup>.
- 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*).

*parentheses*

*function name*

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

# Functions

- *Functions* are special commands that can do more than simple operators<sup>9</sup>.
- 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*).

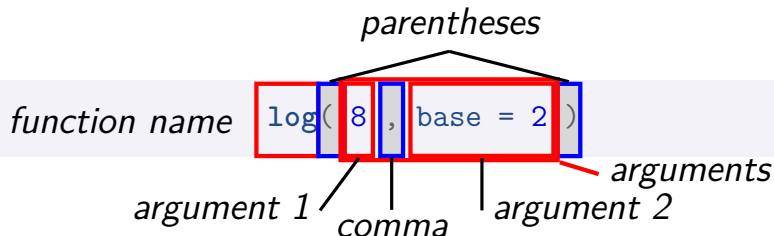
*parentheses*

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

The diagram illustrates the syntax of an R function call. The function name 'log' is enclosed in a red box. The opening parenthesis '(' is enclosed in a blue box. The arguments '8 , base = 2' are enclosed in a red box. The closing parenthesis ')' is enclosed in a blue box. A black line labeled 'parentheses' connects the two parentheses. A red line labeled 'arguments' points to the argument box.

# Functions

- *Functions* are special commands that can do more than simple operators<sup>9</sup>.
- 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*).



<sup>9</sup>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*):  
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, error=  
      if (inherits(pos, "error")  
        name <- substitute(na  
        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](http://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

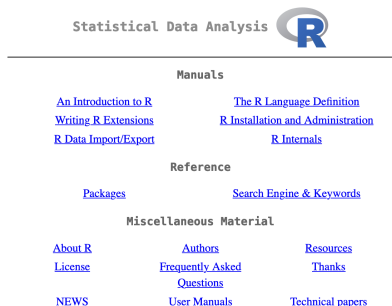
- R web site: [www.r-project.org](http://www.r-project.org)
  - ▶ especially the “Documentation” section
  - ▶ e-mail lists
- RStudio Education: [education.rstudio.com](http://education.rstudio.com)
- R-bloggers.com [www.r-cookbok.com](http://www.r-cookbok.com)
- Stack Overflow ([stackoverflow.com](http://stackoverflow.com))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

# HELP: Reference cards / cheat sheets

- <https://cran.r-project.org/doc/contrib/Short-refcard.pdf>
- <https://cran.r-project.org/doc/contrib/refcard.pdf>
- RStudio IDE cheat sheet
- Search the internet for
  - ▶ “R cheat sheet”
  - ▶ “R reference card”

# 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
```

```
?c
```

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

```
?seq
```

```
?help.search
```

```
help.search("t-test")
```

```
?? 't-test'
```

- Documentation about documentation, and how to search it
- read about the often-used 'combine' *function*
- read about a function for making a *sequence*
- 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* → seq {base} ← *package*  
(or topic)

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 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

- ... 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

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 on how  
the function works*

## 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()` 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

`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.

### 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](#).

[i.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)
```



## 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

# Section 11

## Installing packages

## Section 12

### Saving code (files)

# Saving code (files)

# Section 13

## Backmatter

# Quiz Review





# 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](https://education.rstudio.com))
  - ▶ tutorials, workshop materials, and other resources.
-  Manuals (<https://cran.r-project.org/manuals.html>)
-  Contributed Documentation
  - ▶ e.g., <http://cran.r-project.org/doc/contrib/usingR.pdf>
- Internet search
  - ▶ Stack Overflow ([stackoverflow.com](https://stackoverflow.com))
  - ▶ Cookbook for R ([www.cookbook-r.com](http://www.cookbook-r.com))