A Succinct Intro to R

Steve Haroz

2021

Contents

About

This book is a short introduction to the R language. It covers the basics of R that are not covered by analysis and visualization guides like R for Data Science. Consider it a quick way to get up to speed on R before diving into the analysis and visualization aspects.

This example-focused guide assumes you are familiar with programming concepts but want to learn the R language. It offers more examples than an "R cheat sheet" without the verbosity of a language spec or an introduction to programming.

Sources of inspiration

http://alyssafrazee.com/introducing-R.html

R for programmers

Acknowledgements

People who have offered helpful suggestions: @pietroppeter

Prerequisites

The prerequisites in R for Data Science are the same for this guide:

1. Install R for Windows, Mac, or your variant of Linux.
2. Install RStudio.
3. Run RStudio, and install the tidyverse by typing the following into the RStudio console: `install.packages("tidyverse")`

Chapter 1

Variables, Math, and Comparisons

1.1 Help

```r
# Hi. This is a comment.

# If you know a function's name, but not how to use it:
?t.test
```

You can also click a function and press F1 to see the help page.

If you don't know the exact name of a function or variable, you can type part of the name and press tab to autocomplete and see some info about it.

1.2 Assignment

```r
a = 6
b = 8
c = 5.44
d = TRUE
e = "hello world"
e = 'hello world' # same as double quote
```

Note: No semi colon or "var" needed

You'll sometimes see `a <- 6` instead of `a = 6`. Just use =. Some people insist on using <-. They are silly.

1.3 Names with weird characters

R allows names to have a `.`, and it's common in many built-in functions. For your own variables, avoid it if possible. If you want to have a space in a name, use an underscore (_) instead of being ridiculous.

To learn how to access object members, see the lists chapter.

```
this.is.a.variable.name = 1
better_name = 2
```

You can use any weird character like a space in a variable name by surrounding the name with '. Avoid it if you can, but sometimes it's necessary when you load data from a file.

```
`more than four (>4)` = 5
```

1.4 Console Output

Print `a` in the console

```
a
```

```
#> [1] 6
```

The `[1]` is output because it is the first element in an array. For more info, see the arrays chapter.

Another option that's useful inside functions, which don't output most results

```
print(a)
```

```
#> [1] 6
```

1.5 Math

Arithmetic

```
z = a + b
z = a - b
z = a * b
z = a / b
z = a %/% b # Integer division
z = a %% b # Note the double % for the modulo operator
z = a ^ b # exponent

1 + 2 - 3 * 4 / 5 ^ 6 # Please excuse my dear aunt, Sally
```

```
#> [1] 2.999232
```

Note: There is no ++ or +=

Functions for floats

```
floor(4.82)
```

```
#> [1] 4
```

```
ceiling(4.82)
```

```
#> [1] 5
```

Rounding

```
round(4.4) # round down
```

```
#> [1] 4
```

```
round(4.6) # round up
```

```
#> [1] 5
```

```
round(4.5) # round to even (down)
```

```
#> [1] 4
```

```
round(5.5) # round to even (up)
```

```
#> [1] 6
```

Other basic math functions

```r
sin(pi/2) + cos(0) # radians, not degrees
```

```
#> [1] 2
```

```r
log(exp(2)) # base e (like ln) is the default
```

```
#> [1] 2
```

```r
log(100, 10) # use base 10
```

```
#> [1] 2
```

1.6 Comparisons

```r
a == b
```

```
#> [1] FALSE
```

```r
a != b
```

```
#> [1] TRUE
```

```r
a > b
```

```
#> [1] FALSE
```

```r
a < b
```

```
#> [1] TRUE
```

```r
a >= b
```

```
#> [1] FALSE
```

```r
a <= b
```

```
#> [1] TRUE
```

1.7 Boolean

```
TRUE & FALSE
```

```
#> [1] FALSE
```

```
TRUE | FALSE
```

```
#> [1] TRUE
```

```
!TRUE
```

```
#> [1] FALSE
```

There's `&` and `&&`. You usually want just `&`.

Chapter 2

Arrays

In R, arrays are commonly called “vectors”. R likes to be special.

2.1 Everything is an array

In R, even single values are arrays. That’s why you see `[1]` in front of results: even single values are the first item in an array of length one.

2.2 Creation

`c()` is some sort of legacy nonsense from the S language. I think it means *character array* even though it can hold things other than characters.

I pronounce it “CAW”. Like the sound a crow makes.

Simple array

```
c(8, 6, 7, 5)
```

```
#> [1] 8 6 7 5
```

For multiple types, R converts elements to the most complex type (usually a string). For a real multi-typed collection, see lists

```
c(9, 'hello', 7)
```

```
#> [1] "9"     "hello" "7"
```

2.3 Array generators

R has a cultural fear of complete words. Many terms are shortcuts or acronyms.

Repeat

```
rep(0, 4)
```

```
#> [1] 0 0 0 0
```

```
rep(c(1,2,3), 4) # repeate the whole array
```

```
#>  [1] 1 2 3 1 2 3 1 2 3 1 2 3
```

```
rep(c(1,2,3), each=4) # repeat each item in the array before moving to the next
```

```
#>  [1] 1 1 1 1 2 2 2 2 3 3 3 3
```

Sequence

```
#increment by 1
4:10
```

```
#> [1]  4  5  6  7  8  9 10
```

```
#increment by any other value
seq(from=10, to=50, by=5)
```

```
#> [1] 10 15 20 25 30 35 40 45 50
```

Randomly sample from a given distribution

```
# uniform distribution (not 'run if')
runif(n=5, min=0, max=1)
```

```
#> [1] 0.1594836 0.4781883 0.7647987 0.7696877 0.2685485
```

```
# normal distribution
rnorm(n=5, mean=0, sd=1)
```

```
#> [1]  0.4483395  1.0208067 -0.1378989  0.2103863 -0.6428271
```

2.4 Concatenate arrays

An array made up of smaller arrays combines them. R doesn't seem to allow for an array of arrays.

```
x = 1:3
y = c(10, 11)
z = 500

c(x, y, z)
```

```
#> [1]   1   2   3  10  11 500
```

Note: z is technically an array of length 1

Collapse an array into a string

```
paste(1:5, collapse=", ")
```

```
#> [1] "1, 2, 3, 4, 5"
```

2.5 Indexing

```
a = 10:20
```

Get the first value - **Indices start at 1, not 0**

```
a[1]
```

```
#> [1] 10
```

2nd and 6th values

```
a[c(2,6)]
```

```
#> [1] 11 15
```

Exclude the 2nd and 6th values

```r
a[c(-2,-6)]
```

```
#> [1] 10 12 13 14 16 17 18 19 20
```

Range of values

```r
a[2:6]
```

```
#> [1] 11 12 13 14 15
```

Any order or number of repetitions

```r
a[c(2, 4, 6, 6, 6)]
```

```
#> [1] 11 13 15 15 15
```

specify values using booleans (keep this in mind for the "Array operations" section)

```r
a[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)]
```

```
#> [1] 10 12 14 16 18 20
```

2.6 Sampling from an Array

Randomly sample from an array. Elements may repeat.

```r
sample(1:3, size=10, replace=TRUE)
```

```
#>  [1] 1 1 2 3 2 2 2 2 3 1
```

`replace` means "sample with replacement", so an element can be sampled more than once

Sample without replacement. Elements will not repeat.

```r
sample(1:5, size=4, replace=FALSE)
```

```
#> [1] 4 3 1 5
```

Shuffle the order of an array

```
sample(a, size=length(a), replace=FALSE)
```

```
#>  [1] 15 13 17 20 18 10 12 16 14 11 19
```

Make sure you have enough elements

```
sample(1:5, size=10, replace=FALSE)
```

```
#> Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the popu
```

2.7 Array constants

The `letters` and `LETTERS` constants hold lower and upper case letters

```
letters[1:5]
```

```
#> [1] "a" "b" "c" "d" "e"
```

```
LETTERS[1:5]
```

```
#> [1] "A" "B" "C" "D" "E"
```

2.8 Array operations

Many functions in R are *vectorized*, so they apply to arrays.

```
a * 2
```

```
#>  [1] 20 22 24 26 28 30 32 34 36 38 40
```

Compare individual elements

```
a > 15
```

```
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Compare each element across arrays

```
a == c(10, 9, 12, 13, 14, 15, 16, 17, 18, 19, 20)
```

```
#>  [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Select elements using boolean array

```
a[a>15]
```

```
#> [1] 16 17 18 19 20
```

You can perform operations on the elements of two arrays **even if they are different sizes**. The smaller one wraps around.

```
a = 1:5
b = rep(1, 8)
a + b
```

```
#> Warning in a + b: longer object length is not a multiple of shorter object
#> length
```

```
#> [1] 2 3 4 5 6 2 3 4
```

2.9 Array functions

Length

```
length(20:50)
```

```
#> [1] 31
```

Reverse

```
rev(1:5)
```

```
#> [1] 5 4 3 2 1
```

Math

```
min(1:5)
```

```
#> [1] 1
```

```
max(1:5)
```

```
#> [1] 5
```

```
sum(1:5)
```

```
#> [1] 15
```

2.10 Array sorting

Sort

```
a = c(70, 20, 80, 20, 10, 40)
sort(a)
```

```
#> [1] 10 20 20 40 70 80
```

Reverse

```
sort(a, decreasing=TRUE)
```

```
#> [1] 80 70 40 20 20 10
```

Get the indices of the sorted values

```
order(a)
```

```
#> [1] 5 2 4 6 1 3
```

2.11 Test membership

To see if an item is in an array, use `%in%`

```r
9 %in% 1:10
```

```
#> [1] TRUE
```

```r
9:11 %in% 1:10
```

```
#> [1]  TRUE  TRUE FALSE
```

Chapter 3

Types

For this chapter, you'll need to load the tidyverse library. More on libraries

R is dramatic and gives "conflict" messages that look like errors. Ignore them.

```
library(tidyverse)
```

3.1 Numbers

R has integers but defaults all numbers to `numeric` which is a double precision float.

```
x = 5 # no decimal but still a double
y = x + 1
```

Good ol' float point comparison

```
x = .58
y = .08
x - y == 0.5
```

```
#> [1] FALSE
```

```
round(x-y, digits=1) == round(0.5, digits=1)
```

```
#> [1] TRUE
```

Numeric division returns a double

```
9 / 2 # double precision float
```

```
#> [1] 4.5
```

```
9 %/% 2 # drop the part after the decimal
```

```
#> [1] 4
```

3.2 Strings

Single and double quotes are the same in R, but a given string needs the same in the beginning and end

```
"hello world"
```

```
#> [1] "hello world"
```

```
'hello world'
```

```
#> [1] "hello world"
```

```
"single quote ' in a string"
```

```
#> [1] "single quote ' in a string"
```

```
'double quote " in a string'
```

```
#> [1] "double quote \" in a string"
```

Strings are not character arrays in R, so array techniques may not work as expected.

String length

```
length('hello world') # R sees this is one string, not many charaters
```

```
#> [1] 1
```

```
str_length('hello world') # the actual length of the string
```

```
#> [1] 11
```

Substring

```
str_sub('hello world', 2, 10)
```

```
#> [1] "ello worl"
```

Comparison

```
'hello' == "hello"
```

```
#> [1] TRUE
```

3.2.1 Strings with special characters

If you want to use special characters in a string, you need to "escape it" by adding \

```
"string with backslashes \\, double quote \", and unicode \u263A"
```

```
#> [1] "string with backslashes \\, double quote \", and unicode <U+263A>"
```

Or you can use the literal `r"(text)"` which is useful for a Windows path or regular expression

```
r"(c:\hello\world)"
```

```
#> [1] "c:\\hello\\world"
```

3.2.2 String Concatenation

Concatenate with a space in between

```
paste('hello', 'world')
```

```
#> [1] "hello world"
```

Use a different separator

```
paste('hello', 'world', sep='_')
```

```
#> [1] "hello_world"
```

No separator

```
paste('hello', 'world', sep='')
```

```
#> [1] "helloworld"
```

```
paste0('hello', 'world') # a shortcut function for no separator
```

```
#> [1] "helloworld"
```

See the strings chapter in R4DS for more.

3.3 Dates

See the dates chapter in R4DS.

3.4 Finding the type of a variable

```
class(c(5, 'hi', TRUE))
```

```
#> [1] "character"
```

3.5 Checking the type

What's the type?

```
class(5)
```

```
#> [1] "numeric"
```

Remember, arrays are the same as single values.

```
class(1:5)
```

```
#> [1] "integer"
```

Test if numeric

```
is.numeric(5)
```

```
#> [1] TRUE
```

Test if string

```
is.character('hi')
```

```
#> [1] TRUE
```

Test if boolean

```
is.logical(TRUE)
```

```
#> [1] TRUE
```

3.6 Converting and parsing

Parse or convert to numeric

```
as.numeric(c("5", TRUE, 1:3, "abc"))
```

```
#> Warning: NAs introduced by coercion
```

```
#> [1]  5 NA  1  2  3 NA
```

To string

```
as.character(5)
```

```
#> [1] "5"
```

```
format(1/3)
```

```
#> [1] "0.3333333"
```

```
format(1/3 , digits = 16)
```

```
#> [1] "0.3333333333333333"
```

```
as.character(TRUE)
```

```
#> [1] "TRUE"
```

Convert to boolean. Zero is false. Other numbers are true.

```
as.logical(0:2)
```

```
#> [1] FALSE  TRUE  TRUE
```

3.7 Special types

3.7.1 NA

Missing values are very common in datasets.

```
is.na(c(NA, 1, ""))
```

```
#> [1]  TRUE FALSE FALSE
```

Any operation performed on NA will also yield NA. So, you can operate on arrays with missing values.

```
c(5, NA, 7) + 1
```

```
#> [1]  6 NA  8
```

Be careful about aggregation functions like `min()`, `max()`, and `mean()`. To ignore NAs, use the `na.rm` parameter.

```
mean(c(5, NA, 7), na.rm=TRUE)
```

```
#> [1] 6
```

3.7.2 Factor

A factor is like an enum in other languages. It encodes strings as integers via a dictionary.

Create an array with many repeating values

```
data = sample(c("hello", "cruel", "world"), 12, replace=TRUE)
data
```

```
#>  [1] "world" "cruel" "world" "hello" "world" "hello" "cruel" "world" "world"
#> [10] "hello" "cruel" "world"
```

Make it into a `factor`

```
data = factor(data)
data
```

```
#>  [1] world cruel world hello world hello cruel world world hello cruel world
#> Levels: cruel hello world
```

The array is now an integer array with a dictionary

```
as.numeric(data)
```

```
#>  [1] 3 1 3 2 3 2 1 3 3 2 1 3
```

```
data[1]
```

```
#> [1] world
#> Levels: cruel hello world
```

See the different values in the array

```
levels(data)
```

```
#> [1] "cruel" "hello" "world"
```

For more info, see forcats.

Chapter 4

Control Flow

R is primarily a functional language, so you often don't need control flow yourself. But if you want to, go for it. If you can write some quick code with a for loop, go for it! Tell the R bullies to fuck off. Do what feels comfortable to you.

4.1 If

Simple if

```r
a = TRUE
if (a)
  print("a is TRUE")
```

```
#> [1] "a is TRUE"
```

```r
# conditionally run multiple expressions
if (a) {
  print("a is TRUE")
  print("a is TRUE")
}
```

```
#> [1] "a is TRUE"
#> [1] "a is TRUE"
```

If Else

```
x = 5
y = 8
if (x > y) {
  print("x is greater than y")
} else {
  print("x is less than or equal to y")
}
```

```
#> [1] "x is less than or equal to y"
```

The `ifelse` function is the way to handle vector operations. It is like a vectorized version of `? :` in C or javascript.

```
x = 1:10
ifelse(x %% 2 == 0, "even", "odd")
```

```
#>  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
```

4.2 While

```
x = runif(1)
while (x < 0.95) {
  x = runif(1)
}
```

4.3 For

For works like foreach in other languages.

```
a = runif(100, 1, 100)
for (x in a) {
  if (x > 95)
    print(x)
}
```

```
#> [1] 97.59641
#> [1] 97.67089
#> [1] 95.54705
```

Chapter 5

Functions

Basic function

```
foo = function () {
  print("hello world")
}
foo()
```

```
#> [1] "hello world"
```

Note: in the function, you need to use `print` to output

5.1 Parameters

Parameters and return values

```
addOne = function (x) {
  return(x + 1)
}
addOne(5)
```

```
#> [1] 6
```

The syntax for return is like a function: `return(value)`

Parameter order can be arbitrary

```r
add = function (x, y) {
  return(x + (y*10))
}
add(x=2, y=10)
```

```
#> [1] 102
```

```r
add(y=10, x=2)
```

```
#> [1] 102
```

Functions are vectorized by default

```r
addOne(1:5)
```

```
#> [1] 2 3 4 5 6
```

All parameters are pass-by-value because functions are immutable.

```r
a = 5
foo = function (a) {
  a = 6
  print(paste("Inside the function as a parameter: ", a))
}
print(paste("Before the function: ", a))
```

```
#> [1] "Before the function:  5"
```

```r
foo(1)
```

```
#> [1] "Inside the function as a parameter:  6"
```

```r
print(paste("After the function: ", a))
```

```
#> [1] "After the function:  5"
```

5.2 Scope

When you assign a value inside a function, it creates a local variable in the scope of the function. You can't access the global variable inside the function. (OK, you can, but the syntax is so obnoxious that I pretend it doesn't exist)

```
a = 5
foo = function () {
  a = 6
  b = 100
  print(paste("Inside the function a =", a))
  print(paste("Inside the function b =", b))
}
print(paste("Before the function a =", a))
```

```
#> [1] "Before the function a = 5"
```

```
foo()
```

```
#> [1] "Inside the function a = 6"
#> [1] "Inside the function b = 100"
```

```
print(paste("After the function a =", a))
```

```
#> [1] "After the function a = 5"
```

```
#trying to use `b` will cause an error because it is out of scope
```

5.3 A function in a function

Might be useful for encapsulation

```
foo = function (a, b) {
  square = function(x) {
    return(x ^ x)
  }
  print(c(a, b))
  print(c(square(a), square(b)))
}
foo(1, 10)
```

```
#> [1]  1 10
#> [1] 1e+00 1e+10
```

5.4 Dot dot dot

```
foo = function (a, b) {
  return (a / b)
}
bar = function(a, ...) {
  return(foo(a, ...))
}
bar(50, 10)
```

```
#> [1] 5
```

```
bar(b = 10, 50) # named works too
```

```
#> [1] 5
```

5.5 Operators

Operators like + or - or even [are all functions. To use them like a function, surround them with ‘.

```
`+`(3, 4)
```

```
#> [1] 7
```

```
`*`(3, 4)
```

```
#> [1] 12
```

```
`[`(5:10, 2) # you don't need the close bracket (])
```

```
#> [1] 6
```

Chapter 6

Lists

A list is like an array but it can contain multiple types of elements.

6.1 Make a list

```
x = list(
  a = 5,
  b = 2,
  Long_Name = 4.8,
  "named with spaces" = 0,
  12, # not every element needs a name
  a = 20 # names don't have to be unique (but you really should avoid this)
)
```

6.2 Accessing elements in a list

Get a tuple of the key and value

```
x['a'] # by key name
```

```
#> $a
#> [1] 5
```

```
x[1]    # by index
```

```
#> $a
#> [1] 5
```

Multiple keys

```
x[c('b', 'a')] # by key name
```

```
#> $b
#> [1] 2
#>
#> $a
#> [1] 5
```

Type the list name, then $, and press tab. R will pop up a list of keys to auto-complete. R uses $ in the way that other languages use .

```
x$Long_Name
```

```
#> [1] 4.8
```

Note: Only the value is returned.

6.3 Brackets for real

Sometimes, R will return the whole list or object even though you asked for just one element. So you need to use double brackets. Why? Because R is snarky and doesn't believe you actually want what you said. So you need to use double brackets to explain to R that you're sure this is what you want.

Double brackets only works for single items, not subsetting.

```
x[['a']] # by key name
```

```
#> [1] 5
```

```
x[[1]]    # by index
```

```
#> [1] 5
```

6.4 Names and values

Use the `names()` function to get and set names. It behaves like an array.

```
names(x)
```

```
#> [1] "a"                  "b"                    "Long_Name"
#> [4] "named with spaces" ""                     "a"
```

```
names(x)[3]
```

```
#> [1] "Long_Name"
```

You can modify names by assigning strings to the names function. This is weird. Take a minute to let it sink in.

```
names(x) = c("first", "second", "third", "fourth")
names(x)[3] = "new name"
```

If all elements are the same type, this will get a vector of values

```
myList = list(a=1, b=2, c=3, d=4)
as.vector(unlist(myList))
```

```
#> [1] 1 2 3 4
```

Chapter 7

Libraries and packages

A library or package is a collection of variables, datasets, functions, and/or operators.

It's called a "package" when being installed `install.packages("tidyverse")` and a "library" when being loaded for use `library(tidyverse)`.

A library and a package are the same thing, but R people will insist there is a difference. Whenever talking to R people, you've got a 50-50 chance of getting it right. If you get it wrong, you're going to get a short lecture. Just nod, and say "yes, that makes sense, and the distinction is clearly important". If you say anything else, you'll get a much longer more boring lecture.

I define these functions, so I don't have to worry about confusing the two.

```
install.library = install.packages
package = library
```

If you only want to access one function or variable in a library without loading the whole thing, you can use `::`

```
dplyr::band_instruments
```

```
#> # A tibble: 3 x 2
#>   name  plays
#>   <chr> <chr>
#> 1 John  guitar
#> 2 Paul  bass
#> 3 Keith guitar
```