R Programming Reference

Andrew Andrade (andrew at andrewandrade.ca) @ uWaterloo DataTeam

Last Updated: June 24, 2015

Contents

0.1 Introduction

This documents serves as a reference sheet with example for the basics of R programming.

The purpose is to provide a compilation of use information and code snippets for basics of R Progamming for data science. Please help contribute by adding snipets you find useful.

Source: https://github.com/uWaterlooDataTeam/r-programming-tutorial

Looking forward to seeing pull requests!

1 Data Types and Basic Operations

1.1 Basic Operators

`+` Addition
`-` Subtraction
`*` Multiplication
`/` Division
ˆ Power

1.1.1 Logical Operators

`<` Less than
`>` Greater than
`<=` Less than or equal to
`>=` Greater than or equal to
`==` Equal to
`!=` Not Equal to
`|` Element-wise Or
`&` Element-wise And
`||` Or
`&&` And

1.1.2 Assignment Operators

Comes in 3 forms:

`<-` var_name <- evaluation
`=` var_name = evaluation
`->` var_name -> evaluation

```
x <- 1
x = 1
1 -> x
```

Note: var_name <- evaluation is not the same as var_name < -evaluation

```
x <- 1
x
```

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

```
x < -1
```

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

1.2 Atomic classes of objects

- Character

- Numeric
- Integer
- Complex
- Logical

1.2.1 character

Use " "

```
c("My", "name", "is")
```

```
## [1] "My"   "name" "is"
```

Combining elements

```
my_char <- c("My", "name", "is")
paste(my_char, collapse = " ")
```

```
## [1] "My name is"
```

```
# Vectorized
paste(1:3, c("X", "Y", "Z"), sep = "")
```

```
## [1] "1X" "2Y" "3Z"
```

```
# recycling
paste(LETTERS, 1:4, sep = "-")
```

```
##  [1] "A-1" "B-2" "C-3" "D-4" "E-1" "F-2" "G-3" "H-4" "I-1" "J-2" "K-3"
## [12] "L-4" "M-1" "N-2" "O-3" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2"
## [23] "W-3" "X-4" "Y-1" "Z-2"
```

1.2.2 numeric

- Generally treated as a double real number or “numeric”

```
class(1)
```

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

1.2.3 integer

```
class (1L)
```

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

- Inf represents infinity

```
1/0
```

```
## [1] Inf
```

- Inf can be used in calculations

```
1/Inf
```

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

- NaN represent undefined value or missing value

```
0/0
```

```
## [1] NaN
```

1.2.4 complex

```
z <- complex(real = 1, imaginary = 2)
z
```

```
## [1] 1+2i
```

1.2.5 logical

```
my_vector <-1:10
my_logical <- my_vector == 5
my_logical
```

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

1.3 Attributes

Sample attributes:

- names, dimnames
- dimentions
- class
- length

```r
temp <- 1:10
length (temp)
```

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

- other meta data

Attributes can be used using attributes()

1.4 Sequences

Simplest way is using :

```r
temp <- 1:10
temp
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
temp <- pi:10
temp
```

```
## [1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593
```

```r
temp <- 15:1
temp
```

```
##  [1] 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

Alternatively you can use seq()

```r
my_seq <- seq(0, 20)
my_seq <- seq(0, 10, by=0.5)
my_seq <- seq(5, 10, length=30)
```

1.4.1 seq_along

```r
seq_along (my_seq)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30
```

1.4.2 along.with

```r
my_seq <- seq(along.with = my_seq)
```

1.4.3 rep()

```r
rep(0, times = 40)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [36] 0 0 0 0 0
```

```r
rep(c(0, 1, 2), times = 10)
```

```
##  [1] 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

```r
rep(c(0, 1, 2), each = 10)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

1.5 Vectors

- Vectors can only contain objects of same class
- List is a vector that can contain objects of different classes

1.5.1 Creating vectors

- Vector can be created using vector() function

```r
x <- vector("numeric", length = 10)

x
```

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

- c() function can be used to create vectors

```r
x <- c(0.5, 0.6)
x <- c(TRUE, FALSE) ## logical
x <- c(T, F) ## logical
x <- c("a", "b", "c") ## character
x <- 9:29 ## integer
x <- c(1+0i, 2+4i) ## complex
```

1.5.2 Coercion

Coercion occurs when mixing objects

```
y <- c(1.7, "a") ## character
y
```

```
## [1] "1.7" "a"
```

```
y <- c(TRUE, 2) ## numeric
y
```

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

```
y <- c("a", TRUE) ## character
y
```

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

1.5.3 Explicit Coercion

use the as.* functions

```
x <- 0:6
class(x)
```

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

```
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```
as.logical(x)
```

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

```
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

Note: nonsensical coercion results in NA

```
x <- c("a", "b", "c")
as.numeric(x)
```

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

```
## [1] NA NA NA
```

1.5.4 Subsetting Vectors

```
x <- rnorm(100)
x[1:10]
```

```
##  [1]  0.36097178 -1.58265643  0.38058937 -0.28532469  1.45699530
##  [6]  1.30710142 -0.79910796 -0.30846266 -0.05414524 -0.57797822
```

```
#subset everything except 2 and 10
x[c(-2, -10)]
```

```
##  [1]  0.36097178  0.38058937 -0.28532469  1.45699530  1.30710142
##  [6] -0.79910796 -0.30846266 -0.05414524 -0.39700097 -1.24950974
## [11] -0.17718082 -0.04801837  0.07278154  0.35021605  0.15289766
## [16] -0.07212695  0.34466519  0.39108614 -0.74736374  1.91965284
## [21]  0.89760350  0.40846651 -0.05976597 -1.76405539 -0.74246209
## [26] -0.84282289  1.07489572  0.84398081 -0.99810233  0.35285093
## [31] -0.66438551  1.30382054 -0.94112185 -0.30071001  0.71634284
## [36] -0.26130007 -0.96302177 -0.73850171  0.21306398 -0.92100579
## [41]  0.81518623  0.72257446 -0.47175223  1.00659737  0.82670335
## [46]  0.50210330 -1.34566008  1.41895775 -0.07002015 -0.84219883
## [51]  0.62453970  0.87985524  0.12066914 -1.90601448 -0.24907506
## [56]  0.71699224  2.40171921 -1.39954695  1.16961015  0.88220421
## [61] -1.09630841  1.24461940 -0.14825317  0.54876814 -0.72631735
## [66] -1.71349772  0.56238704 -1.01526209 -0.02380358 -1.33393594
## [71] -1.34620790  0.88513128  0.99622031  0.90434193  0.50394204
## [76]  0.11299284 -1.22293965 -0.57110726  2.32932414  0.22448684
## [81] -0.07521080 -0.43163910 -0.18189390  0.25221185  1.62871935
## [86]  0.80641423  1.25102185 -1.33208234 -0.86741727  1.28513461
## [91]  0.68290280  1.09503912  0.42164084  1.04725310  0.78524976
## [96] -0.38840797 -0.57512020  1.44500122
```

```
x[-c(2, 10)]
```

```
##  [1]  0.36097178  0.38058937 -0.28532469  1.45699530  1.30710142
##  [6] -0.79910796 -0.30846266 -0.05414524 -0.39700097 -1.24950974
## [11] -0.17718082 -0.04801837  0.07278154  0.35021605  0.15289766
## [16] -0.07212695  0.34466519  0.39108614 -0.74736374  1.91965284
## [21]  0.89760350  0.40846651 -0.05976597 -1.76405539 -0.74246209
## [26] -0.84282289  1.07489572  0.84398081 -0.99810233  0.35285093
## [31] -0.66438551  1.30382054 -0.94112185 -0.30071001  0.71634284
## [36] -0.26130007 -0.96302177 -0.73850171  0.21306398 -0.92100579
## [41]  0.81518623  0.72257446 -0.47175223  1.00659737  0.82670335
## [46]  0.50210330 -1.34566008  1.41895775 -0.07002015 -0.84219883
## [51]  0.62453970  0.87985524  0.12066914 -1.90601448 -0.24907506
## [56]  0.71699224  2.40171921 -1.39954695  1.16961015  0.88220421
## [61] -1.09630841  1.24461940 -0.14825317  0.54876814 -0.72631735
## [66] -1.71349772  0.56238704 -1.01526209 -0.02380358 -1.33393594
## [71] -1.34620790  0.88513128  0.99622031  0.90434193  0.50394204
## [76]  0.11299284 -1.22293965 -0.57110726  2.32932414  0.22448684
## [81] -0.07521080 -0.43163910 -0.18189390  0.25221185  1.62871935
## [86]  0.80641423  1.25102185 -1.33208234 -0.86741727  1.28513461
## [91]  0.68290280  1.09503912  0.42164084  1.04725310  0.78524976
## [96] -0.38840797 -0.57512020  1.44500122
```

1.5.5 Named elements

```
vect <- c(foo = 11, bar = 2, norf = NA)
#view names
names (vect)
```

```
## [1] "foo"  "bar"  "norf"
```

Named vector after creation

```
vect2 <- c(11,2,NA)
names(vect2)<- c("foo", "bar", "norf")
```

View names

```
vect[c("foo", "bar")]
```

```
## foo bar
##  11   2
```

1.6 Matrices

- are vectors with dimension attribute (a integer vector of lenth 2)

```
m <- matrix(nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

```
dim(m)
```

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

```
attributes(m)
```

```
## $dim
## [1] 2 3
```

- matricies are column wise

```
m <- matrix (1:6, nrow = 2, ncol =3)
m
```

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

- can be created from vectors by adding dimension attribute

```r
m<-1:10
m
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
dim(m)<- c(2,5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

1.6.1 column bind and row bind

- Create matrices by column-binding (cbind) or row-binding (rbind)

```r
x <- 1:3
y <- 10:12
cbind (x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```r
rbind (x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

1.6.2 Subsetting a Matrix

Matrices can be subsetted in the usual way with (*i*,*j*) type indices.

```r
x <- matrix(1:6, 2, 3)
x[1, 2]
x[2, 1]
```

Subsetting a single element resutls in returning a vector of length 1 rather than a 1x1 matrix. This can be changed using 'drop = FALSE'

```r
x <- matrix(1:6, 2, 3)
x[1, 2]
x[2, 1]
```

Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

```
x <- matrix(1:6, 2, 3)
x[1, ]
x[1, , drop = FALSE]
```

1.7 Lists

As mentioned before, lists are a vector which contain elements of different classes

```
x <- list(1, "a", TRUE, 1 + 4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

1.7.1 Subsetting Lists

```
x <- list(foo = 1:4, bar = 0.6)
x$foo
```

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

```
x$bar
```

```
## [1] 0.6
```

```
x[["bar"]]
```

```
## [1] 0.6
```

```
x["bar"]
```

```
## $bar
## [1] 0.6
```

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

The `[[` operator can be used with *computed* indices; `$` can only be used with literal names.

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]]  ## computed index for ‘foo’
x$name     ## element ‘name’ doesn’t exist!
x$foo
```

1.7.2 Subsetting Nested Elements of a List

The `[[` can take an integer sequence.

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1, 3)]]
x[[1]][[3]]
x[[c(2, 1)]]
```

1.8 Factors

Used to represent categorical data (ie male and female)

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```
table (x)
```

```
## x
##  no yes
##   2   3
```

```
unclass(x)
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

1.8.1 Setting order of levels

- Very important in linear modeling since first level is baseline level

```
x<- factor(c("yes", "yes", "no", "yes", "no"),
levels = c("yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

1.9 Missing Values

Missing values are denoted by NA or NaN for undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true
- Be very careful when combining missing values with logical expressions

```
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

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

```
is.nan(x)
```

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

```
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

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

```
is.nan(x)
```

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

1.9.1 Selecting everything with NA

```
x[is.na(x)]
```

```
## [1] NaN  NA
```

```
x[!is.na(x)]
```

```
## [1] 1 2 4
```

```r
x[is.na(x)] <-0
```

1.9.2 Subset with no missing values

```r
x <- c(1, 2, NA, 4, NA, 5)
y <- c("a", "b", NA, "d", NA, "f")
good <- complete.cases(x, y)
good
x[good]
y[good]
```

1.9.3 Set all NA to zeros

1.10 Data Frames

-Used to save tabular data

-List of same length

-attribute called ‘row.names’

- can be created using ‘read.table()’ and ‘read.csv()’
- be careful when converting to matrix ‘data.matrix()’

```r
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```r
nrow(x)
```

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

```r
ncol(x)
```

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

1.11 Names

1.11.1 Objects

R objects can have names as shown before.

```
x <- 1:3
names(x)
```

```
## NULL
```

```
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

```
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

1.11.2 Lists

```
x <- list (a =1, b = 2, c = 3)
```

1.11.3 Matrices

```
m <- matrix(1:4, nrow = 2, ncol = 2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
m
```

```
##   c d
## a 1 3
## b 2 4
```

2 Reading and Writing Data

2.1 Reading Data

There are a few principal functions reading data into R. - `read.table`, `read.csv`, for reading tabular data

- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (`inverse` of `dump`)
- `dget`, for reading in R code files (`inverse` of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

2.1.1 Reading Data Files with read.table

The `read.table` function is one of the most commonly used functions for reading data.

For small to moderately sized datasets, you can usually call read.table without specifying any other arguments

```
data <- read.table("foo.txt")
```

2.1.2 Reading in Larger Datasets with read.table

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.
- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick an dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                     colClasses = classes)
```

2.1.3 Reading Lines of a Text File

```
con <- gzfile("words.gz")
x <- readLines(con, 10)
x
```

`writeLines` takes a character vector and writes each element one line at a time to a text file.

`readLines` can be useful for reading in lines of webpages

```
## This might take time
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
> head(x)
```

2.2 Writing Data

There are analogous functions for writing data to files - write.table - writeLines - dump - dput - save - serialize

2.2.1 Textual Formats

- `dumping` and dputing are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- `Unlike` writing out a table or csv file, `dump` and `dput` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- `Textual` formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the "Unix philosophy"
- Downside: The format is not very space-efficient

2.2.2 dput-ting R Objects

Another way to pass data around is by deparsing the R object with dput and reading it back in using `dget`.

```
y <- data.frame(a = 1, b = "a")
dput(y)
structure(list(a = 1,
               b = structure(1L, .Label = "a",
                             class = "factor")),
          .Names = c("a", "b"), row.names = c(NA, -1L),
          class = "data.frame")
dput(y, file = "y.R")
new.y <- dget("y.R")
new.y
```

2.2.3 Dumping R Objects

Multiple objects can be deparsed using the dump function and read back in using `source`.

```
x <- "foo"
y <- data.frame(a = 1, b = "a")
dump(c("x", "y"), file = "data.R")
rm(x, y)
source("data.R")
y
x
```

2.2.4 Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

2.2.5 File Connections

```
> str(file)
function (description = "", open = "", blocking = TRUE,
          encoding = getOption("encoding"))
```

- `description` is the name of the file
- `open` is a code indicating
- "r" read only
- "w" writing (and initializing a new file)
- "a" appending
- "rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

2.2.6 Connections

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
con <- file("foo.txt", "r")
data <- read.csv(con)
close(con)
```

is the same as

```
data <- read.csv("foo.txt")
```

3 Control Structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if`, `else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an interation of a loop
- `return`: exit a function

3.1 if

Example of how to use if:

if() { ## do something } else if() { ## do something different } else { ## do something different }

3.1.1 Setting variables using if

```
y <- if(x > 3) {
        10
} else {
        0
}
```

3.2 for

`for` loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
for(i in 1:10) {
        print(i)
}
```

3.2.1 Nested for loops

`for` loops can be nested.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

3.3 while

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0
while(count < 10) {
        print(count)
        count <- count + 1
}
```

3.3.1 multi conditional while

Sometimes there will be more than one condition in the test.

```
z <- 5

while(z >= 3 && z <= 10) {
        print(z)
        coin <- rbinom(1, 1, 0.5)
        if(coin == 1) {  ## random walk
                z <- z + 1
        } else {
                z <- z - 1
        }
}
```

3.4 repeat

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a `repeat` loop is to call `break`.

```
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- computeEstimate()

        if(abs(x1 - x0) < tol) {
                break
        } else {
                x0 <- x1
        }
}
```

The previous loop is a bit dangerous because there's no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

3.5 Special control functions

3.5.1 Next, return

`next` is used to skip an iteration of a loop

```
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

`return` signals that a function should exit and return a given value

3.6 lapply and sapply

3.7 Functions to add:

rnorm
sample
print () prints values identical(vect, vect2)