How to create your first vector in R

Are you an expert R programmer? If so, this is *not* for you. This is a short tutorial for R novices, explaining vectors, a basic R data structure. Here's an example:

10 150 30 45 20.3

And here's another one:

-5 -4 -3 -2 -1 0 1 2 3

still another one:

"Darth Vader" "Luke Skywalker" "Han Solo"

and our final example:

389.3491

These examples show that a vector is, simply speaking, just a collection of one (fourth example) or more (first and second example) numbers or character strings (third example). In R, a vector is considered a data structure (it's a very simple data structure, and we'll cover more complex structures in another tutorial).

So, how can we set up and use a vector in R?

We can construct a vector from a series of individual elements, using the c() function, as follows:

```
c(10, 150, 30, 45, 20.3)
```

```
## [1]  10.0 150.0  30.0  45.0  20.3
```

(In examples like these, lines starting with ## show the output from R on the screen).

Assigning a vector

As you'll see, once you have entered the vector, R will respond by displaying its elements. In many cases it will be convenient to refer to this vector using a name, instead of having to enter it over and over again. We can accomplish this using the assign() function, which is equivalent to the <- and = operators:

```
assign('a', c(10, 150, 30, 45, 20.3))
a <- c(10, 150, 30, 45, 20.3)
a = c(10, 150, 30, 45, 20.3)
```

The second statement (using the <- operator) is the most common way of assigning in R, and we'll therefore use this form rather than the = operator or the assign() function.

Once we have assigned a vector to a name, we can refer to the vector using this name. For example, if we type a, R will now show the elements of vector a.

```
a
```

```
## [1]  10.0 150.0  30.0  45.0  20.3
```

Instead of a, we could have chosen any other name, e.g.:

```
aVeryLongNameWhichIsCaseSensitive_AndDoesNotContainSpaces  <-
c(10, 150, 30, 45, 20.3)
```

Strictly speaking, we call this "name" an object.

To familiarize yourself with the vector data structure, now try to construct a couple of vectors in R and assign them to a named object, as in the example above.

To summarize: A vector is a data structure, which can be constructed using the c() function, and assigned to a named object using the <- operator.

Now, let's move on to the first set of real exercises on vectors!

Celebrating our 100th R exercise set

Yesterday we published our 100th set of exercises on R-exercises. Kudos and many thanks to Avi, Maria Elisa, Euthymios, Francisco, Imtiaz, John, Karolis, Mary Anne, Matteo, Miodrag, Paritosh, Sammy, Siva, Vasileios, and Walter for contributing so much great material to practice R programming! Even more thanks to Onno, who is working (largely) behind the scenes to get everything working smoothly.

I thought perhaps this would be a good time to share some thoughts on the ideas behind the site, and how to proceed from this point onward. The main idea is pretty simple: it helps to practice if you want to learn R programming.

The two problems we're trying to solve

Although the idea itself is simple, for many people, and perhaps you as well, following up on this idea is a challenge. For example, practicing R programming requires a certain task that has to be completed, a solution to an analytical problem that has to be found, or broader goal definition. Without this, we would just be typing random R syntax, or copy-paste code we found somewhere on the web, which will contribute little to improving our R skills. The main problem R-exercises is trying to solve is how to specify these tasks, problems and goals in a useful, creative and structured way. The exercise sets are our (current) solution to this problem.

But there's a second challenge for those who want to practice: Staying focused. Live throws many distractions at us and while you perhaps found some interesting problems to practice your R skills, sooner or later practicing fades away when more urgent matters pop up. So, the second problem R-exercises is trying to solve is how to practice in a focused, persistent way. Offering new exercises on a daily basis, rather than one-time communication (e.g. a book or course) is our solution to this second problem.

Filling a gap in existing solutions

Is there a need for a site filled with exercises? There is an enormous amount of educational material on R available already. Our Course Finder directory includes 140 R courses, offered on 17 different platforms. Many universities teach R as part of their methods/statistics course programs. There are plenty of books on R. A search for "tutorial" on blog aggregator R-bloggers, reveals 1783 articles. And then there's Youtube. It seems, with so much material, gaps are unlikely. But are they?

Going back to the two challenges we just described, we think what we're offering is complementary to courses, books, classes and tutorials. Because the focus of most courses, books, classes and tutorials is on explaining/demonstrating things instead of practicing (the first challenge). And their focus is temporary, not necessarily persistent (the second challenge). It's gone after you completed the course, read the book or watched the video tutorial.

In their excellent book "Make it stick", Roediger and McDaniel explain that many of our intuitive approaches to learning (e.g. rereading a text) are unproductive. Instead they advise: "One of the best habits a learner can instill in herself is regular self-quizzing to recalibrate her understanding of what she does and does not know." From this perspective, R-exercises can help you to recalibrate your understanding of what you know and don't know about R.

The next 100 sets

We're committed to keep expanding R-exercises, and adding more exercise sets. A while ago we started to differentiate sets in terms of difficulty (beginner, intermediate and advanced), an idea that many readers seemed to like when we proposed it. Recently we started to include information about online courses directly related to the exercises in a set, so for those who want to learn more, it's easy to find a relevant course quickly.

Another idea we have is to offer premium (paid) memberships, with access to more extensive learning materials related to each exercise set. We'd actually love to hear your suggestions on how we can improve and expand R-exercises. What would you like to see on the site in 2017?

Data table exercises: keys and subsetting – solutions

Below are the solutions to this first set of exercises on the data.table package.

```
library(data.table)

###############
#             #
# Exercise 1  #
#             #
###############

df                                                                  <-
fread('http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv')
df <- df[rep(1:nrow(df), 1000), ]
format(object.size(df), 'auto')

## [1] "429.7 Mb"

dim(df)

## [1] 4898000      12

df

##            fixed acidity volatile acidity citric acid
residual sugar
##       1:             7.0              0.27        0.36
20.7
##       2:             6.3              0.30        0.34
1.6
```

```
##       3:            8.1              0.28         0.40
6.9
##       4:            7.2              0.23         0.32
8.5
##       5:            7.2              0.23         0.32
8.5
##      ---
## 4897996:            6.2              0.21         0.29
1.6
## 4897997:            6.6              0.32         0.36
8.0
## 4897998:            6.5              0.24         0.19
1.2
## 4897999:            5.5              0.29         0.30
1.1
## 4898000:            6.0              0.21         0.38
0.8
##         chlorides free sulfur dioxide total sulfur dioxide
density   pH
##       1:     0.045                45                  170
1.00100 3.00
##       2:     0.049                14                  132
0.99400 3.30
##       3:     0.050                30                   97
0.99510 3.26
##       4:     0.058                47                  186
0.99560 3.19
##       5:     0.058                47                  186
0.99560 3.19
##      ---
## 4897996:     0.039                24                   92
0.99114 3.27
## 4897997:     0.047                57                  168
0.99490 3.15
## 4897998:     0.041                30                  111
0.99254 2.99
## 4897999:     0.022                20                  110
0.98869 3.34
## 4898000:     0.020                22                   98
0.98941 3.26
##         sulphates alcohol quality
```

```
##       1:      0.45     8.8       6
##       2:      0.49     9.5       6
##       3:      0.44    10.1       6
##       4:      0.40     9.9       6
##       5:      0.40     9.9       6
##      ---
## 4897996:      0.50    11.2       6
## 4897997:      0.46     9.6       5
## 4897998:      0.46     9.4       6
## 4897999:      0.38    12.8       7
## 4898000:      0.32    11.8       6

###############
#             #
# Exercise 2  #
#             #
###############

haskey(df)

## [1] FALSE

setkey(df, quality)
key(df)

## [1] "quality"

###############
#             #
# Exercise 3  #
#             #
###############

system.time(df2 <- df[.(9)])

##    user  system elapsed
##   0.002   0.000   0.002

###############
#             #
```

```
# Exercise 4  #
#             #
###############

setkey(df, NULL)
key(df) ; key2(df)     # no primary and secondary keys are set

## NULL

## NULL

system.time(df2 <- df[quality == 9, ])

##    user  system elapsed
##   0.018   0.000   0.019

key(df) ; key2(df)      # secondary key is generated
automatically

## NULL

## [1] "quality"

system.time(df2 <- df[quality == 9, ])

##    user  system elapsed
##   0.001   0.000   0.002

# Note that without manually setting keys, executing the same
# subset a second time is faster than the first time,
# due to automatically generated secondary key


###############
#             #
# Exercise 5  #
#             #
###############
```

```
setkey(df, NULL)
system.time(df2 <- df[quality %in% 7:9, ])

##    user  system elapsed
##   0.068   0.005   0.096

setkey(df, quality)
system.time(df2 <- df[.(7:9)])

##    user  system elapsed
##   0.051   0.000   0.051

###############
#             #
# Exercise 6  #
#             #
###############

system.time(df3 <- df[`fixed acidity` < 8 & `residual sugar` <
5 & pH < 3])

##    user  system elapsed
##   0.397   0.006   0.431

setkeyv(df, c('fixed acidity', 'residual sugar', 'pH'))
system.time(df3 <- df[`fixed acidity` < 8 & `residual sugar` <
5 & pH < 3])

##    user  system elapsed
##   0.389   0.000   0.388

###############
#             #
# Exercise 7  #
#             #
###############

setkey(df, NULL)
system.time(df3 <- df[sample(.N, .N, replace=TRUE)])
```

```
##    user  system elapsed
##   1.017   0.016   1.110
```

```
df <- as.data.frame(df)
system.time(df3   <-   df[sample(nrow(df),   nrow(df),
replace=TRUE), ])    # much slower
```

```
##    user  system elapsed
##  14.264   0.201  14.462
```

```
df$id <- 1:nrow(df)
df <- data.table(df, key='id')
system.time(df3 <- df[.(sample(.N, .N, replace=TRUE))])  #
slower than sampling directly from rows
```

```
##    user  system elapsed
##   1.635   0.000   1.636
```

Data table exercises: keys and subsetting

The data.table package is a popular R package that facilitates fast selections, aggregations and joins on large data sets. It is well-documented through several vignettes, and even has its own interactive course, offered by Datacamp. For those who want to build some mileage practising the use of data.table, there's good news! In the coming weeks, we'll dive into the package with several exercise sets. We'll start with the first set today,

focusing on creating data.tables, defining keys and subsetting. Before proceeding, make sure you have installed the data.table package from CRAN and studied the vignettes.

Answers to the exercises are available here. For the other (upcoming) exercise sets on data.table, check back next week here. If there are any particular topics/problems related to data.table, you'd like to see included in subsequent exercise sets, please post as a comment below.

Exercise 1
Setup: Read the wine quality dataset from the uci repository as a data.table (available for download from: http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv) into an object named df. To demonstrate the speed of data.table, we're going to make this dataset much bigger, with:
df <- df[rep(1:nrow(df), 1000),]
Check that the resulting data.table has 1.2 mln. rows and 12 variables.

Exercise 2
Check if df contains any keys. If no keys are present, create a key for the quality variable. Confirm that the key has been set.

Exercise 3
Create a new data.table df2, containing the subset of df with quality equal to 9. Use system.time() to measure run-time.

Exercise 4
Remove the key from df, and repeat exercise 3. How much slower is this? Now, repeat exercise 3 once more and check timing. Explain the difference in speed [hint: use the key2() function.]

Exercise 5
Create a new data.table df2, containing the subset of df with quality equal to 7, 8 or 9. First without setting keys, then

with setting keys and compare run-time.

Exercise 6
Create a new data.table df3 containing the subset of observations from df with:
fixed acidity < 8 and residual sugar < 5 and pH < 3. First without setting keys, then with setting keys and compare run-time. Explain why differences are small.

Exercise 7
Take a bootstrap sample (i.e., with replacement) of the full df data.table without keys, and record run-time. Then, convert to a regular data frame, and repeat. What is the difference in speed? Is there any (speed) benefit in creating a new variable id equal to the row number, creating a key for this variable, and use this key to select the bootstrap?

Get-your-stuff-in-order exercises

In the exercises below we cover the basics of ordering vectors, matrices and data frames. We consider both column-wise and row-wise ordering, single and multiple variables, ascending and descending sorting, and sorting based on numeric, character and factor variables. Before proceeding, it might be helpful to look over the help pages for the sort, order, and xtfrm functions.

Answers to the exercises are available here.

If you obtained a different (correct) answer than those listed on the solutions page, please feel free to post your answer as a comment on that page.

Exercise 1
Sort the vector x <- c(1, 3, 2, 5, 4) in:
a. ascending order
b. descending order

Exercise 2
Sort the matrix x <- matrix(1:100, ncol=10):
a. in descending order by its second column (call the sorted matrix x1)
b. in descending order by its second row (call the sorted matrix x2)

Exercise 3
Sort only the first column of x in descending order.

Exercise 4
Consider the women data.
a. Confirm that the data are sorted in increasing order for both the height and weight variable, without looking at the data.
b. Create a new variable bmi, based on the following equation: BMI = (Weight in Pounds / (Height in inches) x (Height in inches)) x 703. Check, again without looking at the data, whether BMI increases monotonically with weight and height.
c. Sort the dataframe on bmi, and its variable names alphabetically

Exercise 5
Consider the CO2 data.
a. Sort the data based on the Plant variable, alphabetically. (Note that Plant is a factor!). Check that the data are sorted correctly by printing the data on the screen.
b. Sort the data based on the uptake (increasing) and Plant (alphabetically) variables (in that order).
c. Sort again, based on uptake (increasing) and Plant (reversed alphabetically), in that order.

Exercise 6

Create a dataframe df with 40 columns, as follows:
df <- as.data.frame(matrix(sample(1:5, 2000, T), ncol=40))
a. Sort the dataframe on all 40 columns, from left to right, in increasing order.
a. Sort the dataframe on all 40 columns, from left to right, in decreasing order.
c. Sort the dataframe on all 40 columns, from right to left, in increasing order.

Image: en:User:RolandH [GFDL, CC-BY-SA-3.0 or CC BY-SA 2.5-2.0-1.0], via Wikimedia Commons

---

Get-your-stuff-in-order: solutions

Below are the solutions to these exercises on sorting and ordering.

```
###############
#             #
# Exercise 1  #
#             #
###############
```

```
x <- c(1, 3, 2, 5, 4)
sort(x)
```

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

```
sort(x, decreasing=T)
```

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

```
###############
#             #
# Exercise 2  #
#             #
###############

x <- matrix(1:100, ncol=10)
x1 <- x[order(-x[,2]), ]
x2 <- x[, order(-x[2, ])]

###############
#             #
# Exercise 3  #
#             #
###############

x[, 1] <- sort(x[, 1], decreasing=T)

###############
#             #
# Exercise 4  #
#             #
###############

is.unsorted(women$height)

## [1] FALSE

is.unsorted(women$weight)

## [1] FALSE

women$bmi <- women$weight  / women$height^2 * 703
is.unsorted(women$bmi)

## [1] TRUE

women <- women[order(women$bmi), sort(names(women))]

###############
```

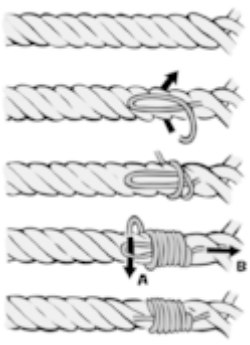
```
#             #
# Exercise 5  #
#             #
###############

CO2 <- CO2[order(as.character(CO2$Plant)), ]
CO2 <- CO2[order(CO2$uptake, as.character(CO2$Plant)), ]
CO2 <- CO2[order(CO2$uptake, -xtfrm(as.character(CO2$Plant))),
]

###############
#             #
# Exercise 6  #
#             #
###############

df <- as.data.frame(matrix(sample(1:5, 2000, T), ncol=40))
df <- df[do.call(order, df), ]
df <- df[do.call(order, -df), ]
df <- df[do.call(order, rev(df)), ]
```

Bind exercises



Binding vectors, matrices and data frames using rbind and cbind is a common R task. However, when dimensions or classes differ between the objects passed to these functions, errors or unexpected results are common as well. Sounds familiar? Time to practice!

Answers to the exercises are available here.

Exercise 1

Try to create matrices from the vectors below, by binding them column-wise. First, without using R, write down whether

binding the vectors to a matrix is actually possible; then the resulting matrix and its mode (e.g., character, numeric etc.). Finally check your answer using R.
a. a <- 1:5 ; b <- 1:5
b. a <- 1:5 ; b <- c('1', '2', '3', '4', '5')
c. a <- 1:5 ; b <- 1:4; c <- 1:3

Exercise 2
Repeat exercise 1, binding vectors row-wise instead of column-wise while avoiding any row names.

Exercise 3
Bind the following matrices column-wise. First, without using R, write down whether binding the matrices is actually possible; then the resulting matrix and its mode (e.g., character, numeric etc.). Finally check your answer using R.
a. a <- matrix(1:12, ncol=4); b <- matrix(21:35, ncol=5)
b. a <- matrix(1:12, ncol=4); b <- matrix(21:35, ncol=3)
c. a <- matrix(1:39, ncol=3); b <- matrix(LETTERS, ncol=2)

Exercise 4
Bind the matrix a <- matrix(1:1089, ncol=33) to itself, column-wise, 20 times (i.e., resulting in a new matrix with 21*33 columns). Hint: Avoid using cbind() to obtain an efficient solution. Various solutions are possible. If yours is different from those shown on the solutions page, please post yours on that page as comment, so we can all benefit.

Exercise 5
Try to create new data frames from the data frames below, by binding them column-wise. First, without using R, write down whether binding the data frames is actually possible; then the resulting data frame and the class of each column (e.g., integer, character, factor etc.). Finally check your answer using R.
a. a <- data.frame(v1=1:5, v2=LETTERS[1:5]) ; b <- data.frame(var1=6:10, var2=LETTERS[6:10])
b. a <- data.frame(v1=1:6, v2=LETTERS[1:6]) ; b <-

```
data.frame(var1=6:10, var2=LETTERS[6:10])
```

Exercise 6
Try to create new data frames from the data frames below, by binding them row-wise. First, without using R, write down whether binding the data frames is actually possible; then the resulting data frame and the class of each column (e.g., integer, character, factor etc.). Finally check your answer using R, and explain any unexpected output.
a. `a <- data.frame(v1=1:5, v2=LETTERS[1:5]) ; b <- data.frame(v1=6:10, v2=LETTERS[6:10])`
b. `a <- data.frame(v1=1:6, v2=LETTERS[1:6]) ; b <- data.frame(v2=6:10, v1=LETTERS[6:10])`

Exercise 7
a. Use cbind() to add vector v3 <- 1:5 as a new variable to the data frame created in exercise 6b.
b. Reorder the columns of this data frame, as follows: v1, v3, v2.

Exercise 8
Consider again the matrices of exercise 3b. Use both cbind() and rbind() to bind both matrices column-wise, adding NA for empty cells.

Exercise 9
Consider again the data frames of exercise 5b. Use both cbind() and rbind() to bind both matrices column-wise, adding NA for empty cells.

Image: By Hella, Handdrawing 1995.

Bind exercises: solutions

Below are the solutions to these exercises on cbind and rbind.

```
####################
#                  #
#    Exercise 1    #
#                  #
####################
```

```
a <- 1:5; b <- 1:5
m <- cbind(a, b)
m
```

```
##      a b
## [1,] 1 1
## [2,] 2 2
## [3,] 3 3
## [4,] 4 4
## [5,] 5 5
```

```
is.matrix(m)
```

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

```
mode(m)
```

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

```
a <- 1:5; b <- c('1', '2', '3', '4', '5')
m <- cbind(a, b)
m
```

```
##      a   b
## [1,] "1" "1"
## [2,] "2" "2"
## [3,] "3" "3"
## [4,] "4" "4"
```

```
## [5,] "5" "5"

is.matrix(m)

## [1] TRUE

mode(m)

## [1] "character"

a <- 1:5; b <- 1:4; c <- 1:3
m <- cbind(a, b)

## Warning in cbind(a, b): number of rows of result is not a
multiple of
## vector length (arg 2)

m

##      a b
## [1,] 1 1
## [2,] 2 2
## [3,] 3 3
## [4,] 4 4
## [5,] 5 1

is.matrix(m)

## [1] TRUE

mode(m)

## [1] "numeric"

####################
#                  #
#    Exercise 2    #
#                  #
```

```
####################

a <- 1:5; b <- 1:5
m <- rbind(a, b, deparse.level=)
m

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

is.matrix(m)

## [1] TRUE

mode(m)

## [1] "numeric"

a <- 1:5; b <- c('1', '2', '3', '4', '5')
m <- rbind(a, b, deparse.level=)
m

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

is.matrix(m)

## [1] TRUE

mode(m)

## [1] "character"

a <- 1:5; b <- 1:4; c <- 1:3
m <- rbind(a, b, deparse.level=)

## Warning in rbind(a, b, deparse.level = 0): number of
```

```
columns of result is
## not a multiple of vector length (arg 2)


m


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

is.matrix(m)


## [1] TRUE

mode(m)


## [1] "numeric"

####################
#                  #
#    Exercise 3    #
#                  #
####################

a <- matrix(1:12, ncol=4) ; b <- matrix(21:35, ncol=5)
nrow(a) == nrow(b)  # to check if cbind is possible


## [1] TRUE

m <- cbind(a, b)
m


##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    4    7   10   21   24   27   30   33
## [2,]    2    5    8   11   22   25   28   31   34
## [3,]    3    6    9   12   23   26   29   32   35

mode(m)
```

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

```
a <- matrix(1:12, ncol=4) ; b <- matrix(21:35, ncol=3)
nrow(a) == nrow(b)  # cbind not possible due to different
number of rows
```

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

```
a <- matrix(1:39, ncol=3) ; b <- matrix(LETTERS, ncol=2)
nrow(a) == nrow(b)  # to check if cbind is possible
```

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

```
m <- cbind(a, b)
m
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,] "1"  "14" "27" "A"  "N"
##  [2,] "2"  "15" "28" "B"  "O"
##  [3,] "3"  "16" "29" "C"  "P"
##  [4,] "4"  "17" "30" "D"  "Q"
##  [5,] "5"  "18" "31" "E"  "R"
##  [6,] "6"  "19" "32" "F"  "S"
##  [7,] "7"  "20" "33" "G"  "T"
##  [8,] "8"  "21" "34" "H"  "U"
##  [9,] "9"  "22" "35" "I"  "V"
## [10,] "10" "23" "36" "J"  "W"
## [11,] "11" "24" "37" "K"  "X"
## [12,] "12" "25" "38" "L"  "Y"
## [13,] "13" "26" "39" "M"  "Z"
```

```
mode(m)
```

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

```
####################
#                  #
#    Exercise 4    #
#                  #
```

```
####################

a <- matrix(1:1089, ncol=33)
a1 <- a[,rep(1:33, 21)]    # possible solution
a2 <- matrix(a, ncol=21*33, nrow=33)   # another solution
all.equal(a1, a2)
```

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

```
####################
#                  #
#    Exercise 5    #
#                  #
####################
```

```
a  <-  data.frame(v1=1:5,  v2=LETTERS[1:5]);  b  <-
data.frame(var1=6:10, var2=LETTERS[6:10])
nrow(a) == nrow(b)  # to check if cbind is possible
```

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

```
m <- cbind(a, b)
m
```

```
##   v1 v2 var1 var2
## 1  1  A    6    F
## 2  2  B    7    G
## 3  3  C    8    H
## 4  4  D    9    I
## 5  5  E   10    J
```

```
str(m)
```

```
## 'data.frame':        5 obs. of  4 variables:
##  $ v1  : int  1 2 3 4 5
##  $ v2  : Factor w/ 5 levels "A","B","C","D",..: 1 2 3 4 5
##  $ var1: int  6 7 8 9 10
##  $ var2: Factor w/ 5 levels "F","G","H","I",..: 1 2 3 4 5
```

```
a  <-  data.frame(v1=1:6,  v2=LETTERS[1:6]);  b  <-
data.frame(var1=6:10, var2=LETTERS[6:10])
nrow(a) == nrow(b)  # cbind not possible due to different
number of rows
```

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

```
####################
#                  #
#    Exercise 6    #
#                  #
####################
```

```
a  <-  data.frame(v1=1:5,  v2=LETTERS[1:5]);  b  <-
data.frame(v1=6:10, v2=LETTERS[6:10])
m <- rbind(a, b)
m
```

```
##    v1 v2
## 1   1  A
## 2   2  B
## 3   3  C
## 4   4  D
## 5   5  E
## 6   6  F
## 7   7  G
## 8   8  H
## 9   9  I
## 10 10  J
```

```
str(m)
```

```
## 'data.frame':        10 obs. of  2 variables:
##  $ v1: int  1 2 3 4 5 6 7 8 9 10
##  $ v2: Factor w/ 10 levels "A","B","C","D",..: 1 2 3 4 5 6
7 8 9 10
```

```
a  <-  data.frame(v1=1:6,  v2=LETTERS[1:6]);  b  <-
data.frame(v2=6:10, v1=LETTERS[6:10])
m <- rbind(a, b)
```

```
## Warning in `[<-.factor`(`*tmp*`, ri, value = 6:10): invalid
factor level,
## NA generated
```

```
m
```

```
##    v1   v2
## 1   1    A
## 2   2    B
## 3   3    C
## 4   4    D
## 5   5    E
## 6   6    F
## 7   F <NA>
## 8   G <NA>
## 9   H <NA>
## 10  I <NA>
## 11  J <NA>
```

```
str(m)
```

```
## 'data.frame':        11 obs. of  2 variables:
##  $ v1: chr  "1" "2" "3" "4" ...
##  $ v2: Factor w/ 6 levels "A","B","C","D",..: 1 2 3 4 5 6
NA NA NA NA ...
```

```
# If the NAs came as a surprise: Note that a$v2 if of class
factor, while b$v2 is of class integer.
```

```
####################
#                  #
#    Exercise 7    #
#                  #
####################
```

```
v3 <- 11:21
m <- cbind(m, v3)
m <- m[c('v1', 'v3', 'v2')]
```

```
####################
#                  #
#    Exercise 8    #
#                  #
####################

a <- matrix(1:12, ncol=4) ; b <- matrix(21:35, ncol=3)
a <- rbind(a, matrix(NA, ncol=4, nrow=2))
m <- cbind(a, b)


####################
#                  #
#    Exercise 9    #
#                  #
####################

a  <-  data.frame(v1=1:6,  v2=LETTERS[1:6]);  b  <-
data.frame(var1=6:10, var2=LETTERS[6:10])
b <- rbind(b, c(NA, NA))
m <- cbind(a, b)
```

Practical uses of R object modes: some examples

One of our readers commented on our mode exercises post: *"What real world tasks are you using mode to solve?"* I think it's an interesting question, from a somewhat larger perspective. Obviously, it'd be a waste of time to learn all kinds of obscure commands that don't have a clear application in the real world. Fortunately, that's not the case with the mode function, and its cousins, as we'll see shortly.

If you're new to R, you might be particularly interested in getting up to speed quickly, and focus on those commands that produce a graph, table, or prediction, or jump straight to some of the popular packages that offer the latest data science toys.

If this approach works for you, great! Many R beginners will suffer unnecessarily however, by taking this route. Why? Read on…

R and Italian, an analogy

Consider this analogy between learning R and learning a real language (as in, Italian or Chinese). Real languages have words that convey strong, clear messages ("Water", "Money", "Sleep"), as well al words that play a more supportive role ("you need", "A little", "Enough"), and words that may convey even less information ("Actually", "Apparently"). (For more analogies like these, read this post on the ideas behind our exercise sets).

Likewise, R has commands that carry out big, important tasks with a lot of immediate practical value (plot, glm, table), as well as commands that play a more supportive role (data.frame, cut, as.Date).

Master the full R vocabulary

My point is this: You need both the strong words/big commands, and the more supportive words/commands to get something done in practice, whether it's in Italian (like, communicating: "Apparently, you need a little sleep") or in R (like: reading some data, cleaning it, running a model and plotting the results).

Many basic R functions, like mode, play this supportive role. In and of itself, they don't have much practical value, but

they are important little nuts and bolts that are indispensable to get your ultimate task done. And are therefore important to master, as part of your R vocabulary!

Practical uses of mode, some examples

Ok, so let’s get back to the original question and look at some practical uses of mode. Looking through some of my recent code, I noticed, I mostly use mode to convert characters to numbers and vice versa, and to check input values in functions.

Ex. 1: Data cleaning

For example, your raw data might have typos or wrong characters, as in the following example, where the last element of the input vector contains the letter ‘o’, instead of the zero digit. Such data can only be read as character, so after reading it into R, you’d have something like the vector x below. Using this vector in a calculation would throw an error.

```
x <- c('20', '30', '4o')   # raw data,
x <- gsub('o', '0', x)     # cleaning
mode(x) <- 'numeric'       # convert from character to numeric
x + 1                      # we can now use x in calculations
```

```
## [1] 21 31 41
```

Ex. 2: Checking input values

Another example of practical use of mode is to check input values in functions. Consider the sum function, which requires arguments of type numeric or logical. It will stop your script

if it receives a character, e.g. sum(10, 'a'). Suppose this is not what you want. Instead you want a sum function that would simply return NA when it receives a character, without stopping the script. In that case you can use is.numeric() to check input values:

```
mysum <- function(a, b) {
  if(is.numeric(a) & is.numeric(b)) a + b
  else NA
}
mysum(10, 20)
```

```
## [1] 30
```

```
mysum(10, 'a')
```

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

Ex. 3: Subsetting

Finally, an example that involves subsetting of a table. Here we convert a numeric range (2005:2010) to character, to select a few columns (representing years) of a table:

```
df <- data.frame(years=1991:2010, v=sample(1:10, 20000, T))
mytable <- with(df, table(v, years))    # a cross-table with counts
mytable[, as.character(2005:2010)]
```

```
##     years
## v    2005 2006 2007 2008 2009 2010
##   1    92  108  115  110   90  119
##   2   101   82   91   98  103   90
##   3    96   99   93   89   84  100
##   4    99  107   92  103  108   97
##   5    99   99  118  118  104   81
##   6    94   89   93   89   95  108
##   7   104  125  108  111  115   95
```

```
##   8   116   85  100   97  108  101
##   9   109   95   94   91   89  102
##   10   90  111   96   94  104  107
```

Do you have other examples of practical uses of mode? Feel free to share below as a comment!

Image By Martorell – Self-published work by Martorell.

Mode exercises

In the exercises below we cover the basics of R object modes. Understanding mode is important, because mode is a very basic property of any R object. Practically, you'll use the mode property often to convert e.g. a character vector to a numeric vector or vice versa. Before proceeding, first read section 3.1 of An Introduction to R, and the help page for the mode function.

Answers to the exercises are available here.

Exercise 1
What is the mode of the following objects? First write down the mode, without using R. Then confirm using an approriate R command.
a. c('a', 'b', 'c')
b. 3.32e16
c. 1/3
d. sqrt(-2i)

Exercise 2
What is the mode of the following objects? First, enter the name of the object at the prompt (R will show its contents), and try to infer the mode from what you see. Then enter an R

command, such that R will print the mode on the screen.
a. pressure
b. lm
c. rivers

Exercise 3
Consider the following list:
x <- list(LETTERS, TRUE, print(1:10), print, 1:10)
What is the mode of x, and each of its elements? First write down the mode, without using R. Then confirm using the appropriate R commands.

Exercise 4
Show whether the vector x <- 1:100 is of mode numeric (TRUE) or not (FALSE).

Exercise 5
Change the mode of the vector x <- 1:100 to character, with and without using the mode function. Write down the first 5 elements of the vector, after the mode conversion. Check your answer by printing the first 5 characters on the screen.

Exercise 6
Change the mode of the character vector you created in the previous exercise, back to numeric. Again, with and without using the mode function.

Exercise 7
Change the mode of the vector x <- c('1', '2', 'three') to numeric. First write down the new vector x, without using R, then check your answer using R.

Exercise 8
Change the mode of the vector x <- c(TRUE, TRUE, FALSE, TRUE) to numeric. First write down the new vector x, without using R, then check your answer using R.

Exercise 9
Consider the vector x <- c('1', '2', 'three'). What is the

mode of y <- x + 1. First write down your answer without using R, then check using R.

Exercise 10
Create a vector y <- c('2', '4', '6') from the vector x <- c('1', '2', '3').

Exercise 11
Try to create some exercises yourself, on the mode topic. This is the best way to really master the subject... Feel free to share as a comment below, so we can all learn from it!