Basic Course on R: Entering and Importing Data

Karl Brand[*] and Elizabeth Ribble[†]

17-21 December 2018

Contents

[*]brandk@gmail.com
[†]emcclel3@msudenver.edu

1 Entering and Importing Data

1.1 Entering Data

In the previous section we created a couple of short vectors, a small matrix, a data frame, and a list. There are many ways to create such objects, especially by utilizing a few convenient functions, and many combinations of these data types are also possible.

1.1.1 A few functions to get started

The most basic function presented first is one we've seen before: the combine function: `c`. This combines into a vector the entries passed to it, so long as they are of the same class:

```
c(1, 2, 3, 4)

## [1] 1 2 3 4

a <- c(1:10, 100)
a

##  [1]   1   2   3   4   5   6   7   8   9  10 100
```

If we want to replicate numbers in the vector, we can type them out (e.g. `1, 1, 1, ...`) or use the function `rep`, with the number to be replicated followed by the number of times to replicate it:

```
b <- rep(x = 1, times = 11)
b

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

The function `seq` generates a sequence of numbers based on the specified start, end and increment of the sequence:

```
c <- seq(from = 1, to = 110, by = 10)
c

##  [1]   1  11  21  31  41  51  61  71  81  91 101
```

To bind two or more vectors (or data frames) together, we can use `cbind` (to combine columns) or `rbind` (to combine rows):

```
abcc <- cbind(a, b, c)
abcc

##         a b   c
##  [1,]   1 1   1
##  [2,]   2 1  11
##  [3,]   3 1  21
##  [4,]   4 1  31
##  [5,]   5 1  41
##  [6,]   6 1  51
##  [7,]   7 1  61
##  [8,]   8 1  71
##  [9,]   9 1  81
## [10,]  10 1  91
## [11,] 100 1 101

abcr <- rbind(a, b, c)
abcr

##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
## a    1    2    3    4    5    6    7    8    9    10   100
## b    1    1    1    1    1    1    1    1    1     1     1
## c    1   11   21   31   41   51   61   71   81    91   101
```

We can use `View` to look at the data - it displays the data values along with the names of the columns:

```
View(abcr)
```

or, to just get the dimension of the resulting object, use `dim`, `nrow` or `ncol`:

```
dim(abcr)

## [1]  3 11

nrow(abcr)

## [1] 3

ncol(abcr)

## [1] 11
```

And remember `str()` will display the data structure and `class()` the class:

```
str(abcr)

##  num [1:3, 1:11] 1 1 1 2 1 11 3 1 21 4 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:3] "a" "b" "c"
##   ..$ : NULL

class(abcr)

## [1] "matrix"
```

1.1.2 Merging data frames

Suppose we want to add some new samples to `mygenes`:

```
mygenes <- data.frame(samp1 = c(33, 22, 12),
                      samp2 = c(44, 111, 13),
                      samp3 = c(33, 53, 65))
row.names(mygenes) <- c("CRP", "BRCA1", "HOXA")
```

but we happen to have the rows (here, genes) in a different order than before, and in fact we have an extra gene as well:

```
newsamp <- data.frame(samp4 = c(56, 13, 106, 10),
                      samp5 = c(45, 15, 99, 13))
row.names(newsamp) <- c("CRP", "HOXA", "BRCA1", "GAPDH")
newsamp

##       samp4 samp5
## CRP      56    45
## HOXA     13    15
## BRCA1   106    99
## GAPDH    10    13
```

We can't use e.g. `cbind` because not only are the dimensions different, but the row order differs. The `merge` function solves this problem: it concatenates based on matching attributes. Use the argument `by` to specify what columns to merge by, i.e., the name(s) of a column(s), or the row names:

```
mygenes2 <- merge(x = mygenes, y = newsamp, by = "row.names")
mygenes2

##   Row.names samp1 samp2 samp3 samp4 samp5
## 1     BRCA1    22   111    53   106    99
## 2       CRP    33    44    33    56    45
## 3      HOXA    12    13    65    13    15
```

The default setting however is to not include anything that doesn't match, so above we lost our new gene, GAPDH. We can force the function to keep all rows of the first and/or second argument with the arguments `all`, `all.x`, or `all.y`:

```
mygenes2 <- merge(x = mygenes, y = newsamp, by = "row.names", all.y = TRUE)
mygenes2

##   Row.names samp1 samp2 samp3 samp4 samp5
## 1     BRCA1    22   111    53   106    99
## 2       CRP    33    44    33    56    45
## 3     GAPDH    NA    NA    NA    10    13
## 4      HOXA    12    13    65    13    15
```

To be conservative and keep everything, use `all = TRUE`. Note that for every column of the first argument that didn't have a value for GAPDH we now have an `NA`. Also note that the row names are no longer the row names, but instead have been put in a new column. We can leave it as it is, and perhaps change the name (as shown in the first line of code below), or use our indexing and naming skills to create another data frame with the row names back where they were:

```
names(mygenes2)[1] <- "gene"
## or
mygenes3 <- mygenes2[, -1]
row.names(mygenes3) <- mygenes2[, 1]
mygenes3

##       samp1 samp2 samp3 samp4 samp5
## BRCA1    22   111    53   106    99
## CRP      33    44    33    56    45
## GAPDH    NA    NA    NA    10    13
## HOXA     12    13    65    13    15
```

1.1.3 Concatenating strings

The function `paste` is great for (re)naming, writing, and creating data. For example, in the last section we created the data frame `mygenes` by entering the name of each data frame element as follows:

```
mygenes <- data.frame(samp1 = c(33, 22, 12),
                      samp2 = c(44, 111, 13),
                      samp3 = c(33, 53, 65))
```

but with a naming convention where the same text is repeated several times ("samp") it is handy to be able to type it in only once:

```
mygenes <- data.frame(c(33, 22, 12),
                      c(44, 111, 13),
                      c(33, 53, 65))
names(mygenes) <- paste("samp", 1:3, sep = "")
row.names(mygenes) <- c("CRP", "BRCA1", "HOXA")
mygenes

##       samp1 samp2 samp3
## CRP      33    44    33
## BRCA1    22   111    53
## HOXA     12    13    65
```

Note the `sep` argument specifies how to separate the strings we are pasting together. We can use any character we want for this parameter:

```
paste("The", "Club", sep = " Mickey Mouse ")

## [1] "The Mickey Mouse Club"
```

By default `sep = " "`. That is, omitting `sep` in the `paste` function will insert spaces:

```
paste("The", "Mickey", "Mouse", "Club")

## [1] "The Mickey Mouse Club"
```

In the next subsection we'll see how paste can be helpful for reading and writing files.

1.2 Importing from a Dataset or File

1.2.1 Importing from a built-in dataset

Many `R` packages come with built-in datasets. Extracting these data are straightforward using `data()`. Invoking this will output a list of all available datasets in a new window:

```r
data()
```

If we introduce the name of the dataset, then the function loads that dataset into the workspace:

```r
HairEyeColor

## , , Sex = Male
##
##        Eye
## Hair    Brown Blue Hazel Green
##   Black    32   11    10     3
##   Brown    53   50    25    15
##   Red      10   10     7     7
##   Blond     3   30     5     8
##
## , , Sex = Female
##
##        Eye
## Hair    Brown Blue Hazel Green
##   Black    36    9     5     2
##   Brown    66   34    29    14
##   Red      16    7     7     7
##   Blond     4   64     5     8
```

1.2.2 Importing from a file

Suppose we have collected some data that we want to analyze in `R` and the data are in matrix format in e.g. a .txt or .csv file. Then we can read this in with a function like `read.table` and assign the output to an object:

```r
data <- read.table(file = "Rcourse_data.txt",
                   header = TRUE, row.names = 1)
data

##       cohort age
## 1001 disease  34
## 1002 control  38
## 1003 disease  22
## 1004 disease  50
## 1005 control  46
## 1006 control  27
## 1007 disease  44
## 1008 control  49
## 1009 control  33
## 1010 disease  30
```

In the arguments we can specify things like:

- the separator e.g., `sep = "\t"` for tab separated (.txt) or `sep = ","` for comma separated (.csv) files;
- the header (as above);
- which column contains the row names (as above);
- whether to skip any columns or fill in blank lines with `NA`;

and so on. The function `read.csv` is the same as `read.table` but uses the default `sep = ","` for reading in .csv files (note that `read.csv2` has default `sep = ";"` and `dec = ","`).

Also be aware of R's default behaviour of converting columns of text strings in a dataframe into the class `factor`. You may not want your column of text strings to be factors, if so, set the argument `stringsAsFactors = FALSE` in the `read.table()` function.

Now suppose the file we want isn't in our working directory (displayed by calling `getwd()`). Then we can type in the path in the name of the file:

```
data <- read.table(file = "C:/temp/Rcourse_data.txt",
                   header = TRUE,
                   row.names = 1)
data
```

But let's say we have a lot of files to read in from places other than our working directory and don't want to keep typing in the same parts of a path every time. Then we can `paste`:

```
mydir <- "C:/temp"
newfile <- paste(mydir, "Rcourse_data.txt", sep = "/")
data <- read.table(file = newfile,
                   header = TRUE,
                   row.names = 1)
```

See also `?file.path` which is built specifically for this purpose, whereas `paste` has more general utility.

1.3 Writing to a File

Now suppose we've done some analysis and want to store our results outside of R. The functions `write.table` and `write.csv` do this:

```r
write.table(x = data, file = "C:/temp/results.txt", sep = "\t")
## or
write.csv(x = data, file = "C:/temp/results.csv")
```

We specified the separator in `write.table` (default is a space), but the default in `write.csv` is a comma. If you do not want to write the row names to the file, set the argument `row.names = FALSE`.

Note we can change the location of our working directory using `setwd()` so that we don't have to worry about specifying a path over and over when reading/writing files, e.g.:

```r
# not run
setwd("C:/Users/Documents/RSTUFF")
```

2 Document License

GNU General Public License v2.0 or higher (GPL>=v2)