Quantitative Understanding in Biology
Introduction to R

Luce Skrabanek, Jason Banfelder

September 10th, 2019

3 Data Structures

3.1 Overview

R has several different types of data structures and knowing what each is used for and when they are appropriate is fundamental to the efficient use of R. The ones that we are going to examine in detail here are: vectors, matrices, lists and data frames.

A quick summary of the four main data structures:

Vectors are ordered collections of elements, where each of the objects must be of the same data type or mode, but can be any mode.

A **matrix** is a rectangular array, having some number of columns and some number of rows. Matrices can only comprise one data type (if you want multiple data types in a single structure, use a data frame).

Lists are like vectors, but whereas elements in vectors must all be of the same type, a single list can include elements from any data type. Elements in lists can be named. A common use of lists is to combine multiple values into a single object that can then be passed to, or returned by, a function.

Data frames are similar to matrices, in that they can have multiple rows and multiple columns, but in a data frame, each of the columns can be of a different data type; within a column, all elements must be of the same data type. You can think of a data frame as being like a list, where each element corresponds to a complete vector, and all elements are the same length.

3.4 Matrices

Multi-dimensional structures in R, where all the elements are of the same data type, are called arrays. Arrays have three dimensions: number of rows, number of columns and number of layers. Matrices are a special type of array using only two of those dimensions (rows and columns) and can be thought of as tables.

1. Let's use one of R's built-in datasets, the `USPersonalExpenditure` dataset, which describes how much Americans spent in five categories from 1940-1960.

   ```
   ?USPersonalExpenditure
   USPersonalExpenditure
   ```

2. Notice that the rows and columns of the matrix are named, using the categories and years as row names and column names, respectively. You can access (and set) the row names and column names

using the `rownames()` and `colnames()` functions. There is a third function, `dim()`, which tells you the number of rows and columns making up your matrix.

```
rownames(USPersonalExpenditure)
colnames(USPersonalExpenditure)
dim(USPersonalExpenditure)
```

These functions return vectors, which can be accessed with all the usual access methods.

3. Accessing (and assigning) elements of a matrix is analogous to accessing (and assigning) elements of a vector, except that matrices need two indexing vectors, one for rows and one for columns.

4. Let's say we wanted to see the Food and Tobacco expenditure for 1950:

```
USPersonalExpenditure[1, 3]
USPersonalExpenditure["Food and Tobacco", "1950"]
USPersonalExpenditure[1, "1950"]
```

5. To get this expenditure for several years, we pass a vector for the column index.

```
USPersonalExpenditure[1, c(5, 3, 1)]
USPersonalExpenditure["Food and Tobacco", c("1960", "1950", "1940")]
```

6. Omitting a row or column index implies that all of the elements are wanted along that dimension.

```
USPersonalExpenditure[1, ]
USPersonalExpenditure["Food and Tobacco", ]
USPersonalExpenditure["Food and Tobacco", , drop = FALSE]
USPersonalExpenditure[ , c("1940","1950")]
USPersonalExpenditure[1:3, c("1940","1950")]
```

7. Observe that if your result is one-dimensional, it is by default returned as a vector. All the vector operations you learned can be used on this output. If you don't want this behavior, you can use the `drop=FALSE` option, as was shown in the third example above.

```
sum(USPersonalExpenditure[ , "1940"])
```

8. If you access an element of a matrix using only one index, R will treat the elements of the matrix as a vector of stacked columns.

```
USPersonalExpenditure[1]
USPersonalExpenditure[2]
USPersonalExpenditure[7]
```

9. This is usually not the clearest way to access elements, but can be useful when you want to work with all of the elements.

```
length(USPersonalExpenditure)
sum(USPersonalExpenditure)
```

10. There are a few ways to make a new matrix. The `matrix()` function takes as arguments a vector of all the elements, and then some information about how many rows and columns there are. By default, the matrix is filled in column order, but you can change this behaviour with the `by.rows=TRUE` option.

```
game1 <- matrix(c("X","","O","","X","O","","",""), ncol = 3)
game2 <- matrix(c("X","","O","","X","O","","",""), ncol = 3, byrow = TRUE)
```

Notes:

a. The elements of this matrix are characters. As with vectors, all elements in a matrix must be of the same datatype.

b. In this matrix, the rows and columns are not named, and can only be accessed using numeric indices.

11. Assignment of values is just like for vectors, except using two dimensions.

```
game1[3, 3] <- "X" # I win!!!!!
```

12. When making a fresh game, we can set the size in advance, and fill it in piecemeal. This is much more efficient than extending the matrix each time a new row or column is added.

```
new.game <- matrix(data = "", ncol = 3, nrow = 3)
```

13. We can assemble a matrix by combining vectors or matrices. They can be stacked by row or column, using the `rbind()` or `cbind()` functions, respectively. Here, we build a chess board from the component pieces.

```
pieces <- c("rook", "knight", "bishop", "queen", "king", "bishop", "knight", "rook")
pawns <- rep("pawn", 8)
board <- rbind(pieces, pawns, matrix("", nrow = 4, ncol = 8), pawns, pieces)
rownames(board) <- 8:1
colnames(board) <- letters[1:8]
```

`letters` is a very handy built-in vector of all 26 lowercase letters. See the help for `constants` for a few more handy built-ins.

14. Remember all of the elements in a matrix must be of the same data type. If you assign one element of a different data type, R will convert (or coerce) elements as necessary.

```
USPersonalExpenditure["Personal Care", "1955"] <- "Unknown"
USPersonalExpenditure
```

All the numbers have been converted to characters. How should we have done this to avoid this coercion?

```
rm(USPersonalExpenditure) # revert to built-in dataset
```

15. R coerces elements up the data type chain, only far enough to satisfy the rule that all elements remain the same type. If you add an integer to a matrix of logicals, you'll get a matrix of integers, not characters, as in the example above.

```
NULL < raw < logical < integer < double < complex < character < list < expression
```

The same coercion strategy applies to vectors.

The next section introduces a data structure that allows mixed types.

3.5 Lists

Lists are similar to vectors, but with some differences. Lists can hold data structures of different types, and of different sizes. Each component in a list can be (optionally, but commonly) separately named (a list in R is analogous to a hash or associative array in most other programming languages). In fact, one list can be a member of another list, allowing for deeply nested and arbitrarily complex data structures to be modeled.

1. The results of many high-level analyses in R are packaged as lists. Let's use R to fit a linear model to some data.

```
(edu.spend <- unname(USPersonalExpenditure["Private Education", ]))
(edu.yr <- seq(from = 0, to = 20, by = 5))
plot(edu.yr, edu.spend)
my.model <- lm(edu.spend ~ edu.yr)
my.model
summary(my.model)
abline(my.model)
plot(my.model)
```

   The `lm()` function fits data to a linear model (i.e., performs a linear regression), and packages up all of the results into an object we have named `my.model`. The `my.model` object is of class `lm` (linear model), which is a special kind of `list`. You may have to `rm()` USPersonalExpediture from your environment before running the above commands, if your object is still coerced into character strings from the previous section.

2. Remember that we can look into any object using the `str()` function. Let's do that now with our linear model. Brace yourself!

```
str(my.model)
```

   Take a moment to get a sense of what is packaged in the linear model object (but don't stress over the details).

   Now look carefully at the very first line of the output and note that this object is a "List of 12". This object has 12 components, and many of those have sub-components (an element of a list can be another list).

   We'll need to learn about lists so we can access the subcomponents of the model object; for example, you'd probably want to extract the slope and intercept of the best-fit line from this object. Let's start with the basics...

3. Lists can be created using the `list()` function. When using the `list()` function, you can optionally give names to the components (component names are called tags).

```
ad.mouse.colony <- list("9.1",  FALSE)
ad.mouse.colony <- list(room = "9.1",  bsl3 = FALSE)
```

   Note that we have different data types in the same list (character, and logical). Note also the difference between the first and second attempt to model a mouse colony. In the first case, you need to know what the position of each component is; in the second, each component is named (tagged) with the tag we provided at creation.

4. We can add a component to an existing list

```
ad.mouse.colony$conditions <- list(bedding = "straw", light_hrs = 12)
ad.mouse.colony$count <- c(male = 10, female = 0)
ad.mouse.colony[["variants"]] <- c("APP695swe", "PS1-dE9")
```

   Note that we have used two different notations to refer to components of a list.

5. There are several ways to refer to the components of a list; the differences can be subtle, but important.

   The most straightforward way to refer to the contents of a single component is using the `[[` notation. You can always provide an integer to access the component by position, or you can provide a character string if the component is tagged.

For tagged components where the tag is a literal character string, you can use the `$` notation. This is less flexible, but looks clearer. Use this when you can.

6. Just as for vectors, to get the names of tagged components, use `names()`.

   ```
   names(ad.mouse.colony)
   ```

7. If you want to access multiple components, you can use the `[` notation, which takes the usual indexing vector. Note that this will always return a list (this has to be so, because the components returned could be of different datatypes).

   Exercise:

   a. What does this return?

      ```
      ad.mouse.colony[[3]]
      ```

   b. How about this?

      ```
      ad.mouse.colony[[3]]$bedding
      ```

   c. Why does this NOT work?

      ```
      ad.mouse.colony[3]$bedding
      ```

   d. How would you add a new element (say, temp = 36.5) to the conditions list?

8. To remove a component, assign it the value `NULL`.

   ```
   ad.mouse.colony$bsl3 <- NULL
   ```

   Note that this changes the index positions of all subsequent components (yet another reason to use tags!)

9. All objects can have arbitrary additional attributes, which can be used to store metadata about the object, which are stored as a named list (with unique names). Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

   ```
   x <- c(0, 1, 1, 2, 3, 5, 8, 13, 21)
   attr(x, "description") <- "Fibonacci sequence"
   attr(x, "description")
   attributes(x)
   str(attributes(x))
   ```

10. Remember our linear model?

    Exercise:

    a. Can you extract the slope and intercept of the best-fit line?

    b. Can you extract the value of R^2 reported by the `summary()` function we used? *Hint: save the result of this function, and look into its structure.*

3.6 Data frames

Data frames are two-dimensional data structures like matrices, but, unlike matrices, they can contain multiple different data types. You can think of a data frame as a list of vectors, where all the vector lengths are the same. Data frames are commonly used to represent tabular data.

1. When we were learning about vectors, we used several parallel vectors, each with length 50 to represent information about US states. The collection of vectors really belongs together, and a data frame is the tool for doing this.

   ```
   state.db <- data.frame(state.name, state.abb, state.area, state.center,
                          stringsAsFactors = FALSE)
   state.db
   ```

   The `data.frame()` function combines the four data sets into a single data frame. Note that the first three data sets are vectors (two character, one numeric), but the last data set is a list with two components.

   Note the `stringsAsFactors = FALSE` argument. Some of the vectors that we are using are character vectors, but will be automatically converted to factors if this option is not set. Since we will want to work with our character data as vectors, not as factors, we want to set this argument to `FALSE`.

2. In addition to the `str()` function, you can glean useful information about a data frame (and other data structures) using the `summary()` and `head()` functions.

   ```
   summary(state.db)
   head(state.db)
   ```

3. Data frames have a split personality. They behave both like a tagged list of vectors, and like a matrix! This gives you many options for accessing elements. Fortunately, you know them all already!

   When accessing a single column, the list notation is preferred.

   ```
   state.db$state.abb
   state.db[[ "state.abb" ]]
   state.db[[ 2 ]]
   ```

   When accessing multiple columns or a subset of rows, the matrix notation is used (rows and columns are given indexing vectors).

   ```
   state.db[ , 1:2]
   state.db[41:50, 1:2]
   state.db[c(50, 1), c("state.abb", "x", "y")]
   state.db[order(state.db$state.area)[1:5], ]
   state.db[order(state.db$state.area), ][1:5, ]
   ```

   The last two examples produce the same output; which is more efficient?

4. As with matrices, the rows can be given names as well. This makes picking out specific rows less error-prone. Column names are accessed with the `names()` or `colnames()` functions.

   ```
   rownames(state.db) <- state.abb
   state.db[c("NY", "NJ", "CT", "RI"), c("x", "y")]
   names(state.db) <- c("name", "abb", "area", "long", "lat")
   ```

   Note that if you only fetch data from one column, you'll get a vector back. If you want a one-column data frame, use the `drop = FALSE` option.

5. You can add a new column the same way you would add one to a list.

   ```
   state.db$division <- state.division # Remember, this is a factor

   state.db$z.size <- (state.db$area - mean(state.db$area))/sd(state.db$area)
   state.db[ , "z.size", drop = FALSE]
   ```


6. Remember that you can pass logical vectors as indices.

```
state.db[state.db$area < median(state.db$area), "name"]
state.db[state.db$area < median(state.db$area), "name", drop = FALSE]
coastal <- state.db[ state.db$division %in%
  c("New England", "Middle Atlantic", "South Atlantic", "Pacific"), ]
```

7. Another way to select data from a data frame is using the `subset()` function. You can choose rows based on a logical expression and can choose columns with the `select` option. With this function, we only have to type the dataset name once.

```
subset(state.db, area < median(area), select = name)
coastal <- subset(state.db, division %in%
  c("New England", "Middle Atlantic", "South Atlantic", "Pacific") )
```

   The logical condition is optional, and you can specify columns to omit instead of columns to include.

```
subset(state.db, select = c(name, abb))
subset(state.db, select = -c(long, lat))
```

8. Many tools in R work naturally with data frames. For example, visualizing the size distribution of state within each division could not be easier once the data is in a well-designed data frame.

```
plot(area ~ division, data = state.db)
plot(log(area) ~ division, data = state.db)
plot(lat ~ long, data = state.db)
text(lat ~ long, data = state.db, rownames(state.db))
```

   Here we are using R's function notation. Read the first argument in the first `plot()` example as "area as a function of division".

9. **Exercise:**

   a. Can you add Puerto Rico to our data frame? [abb = "PR", area = 3515 sq mi, long = -66.1, lat = 18.45, division = "South Atlantic"]

   b. How about Greenland? [abb = "GL", area = 836330 sq mi, long = -51.73, lat = 64.17, division = "Arctic Circle"]

10. In most cases in the lab, you won't be typing the data in by hand but rather importing it. R provides tools for importing a variety of text file formats. If you receive data in Excel format, you'll want to save it as tab-delimited or CSV (comma separated values) text. The `read.delim()` or `read.csv()` functions can then be used to import the data into a data frame. Of course, you should also tell your colleagues that R is the preferred tool for data wrangling!

    We have prepared an Excel file for you to import.

    a. Go to course website and download the ablation.xlsx file into your project folder.

    b. Open the file using Microsoft Excel and save it in CSV format. You can quit Excel now!

    c. Return to RStudio.

    d. Use the Files tab (bottom right) to take a look at the .csv file.

    e. Import the data into a data frame:

```
ablation <- read.csv("ablation.csv", header = TRUE, stringsAsFactors = TRUE)
```


Note that we explicitly asked that strings be converted to factors. In cases where you are importing string data, you would want to set this to `FALSE`, and after import, convert appropriate columns to factors.

f. Rename the `SCORE` column to be consistent with the other column names.

```
names(ablation)[names(ablation) == "SCORE"] <- "Score"
```

Note that we did not use `names(ablation)[6] <- "Score"`. Why do you think that is?

There is another more general import function, `read.table()`, that gives you exquisite control over how to import your data. One of the defaults of this function is `header = FALSE`. For this reason, we suggest that you always explicitly use the `header` option (you don't want to accidentally miss your first data point).

11. If your uninitiated colleagues insist on an Excel-compatible format, you can also export a data frame using `write.table()`.

```
write.table(ablation, file = "ablation.txt",
            quote = FALSE, col.names = NA,
            sep = "\t")

write.table(ablation, file = "ablation.txt",
            quote = FALSE, row.names = FALSE,
            sep = "\t")
```

4 User-defined functions

Although R has a wide array of in-built functions, as well as many other functions that can be accessed via third party packages (more on that later), we can also create our own functions, using `function()`. We define not only the arguments, or inputs, to the function, but also what it does, and what it returns.

1. Within the body of a function, all of the input arguments become variables that you can use, including passing them to other functions.

```
mySummary <- function(x) {
  my.mean <- mean(x)
  my.sd <- sd(x)
  list(mean = my.mean, sd = my.sd)
}
mySummary(rnorm(100))
```

Here, our function, called `mySummary`, takes a single argument called `x`. This function assumes that `x` is a numeric vector, and computes the mean and standard deviation of that vector. The function returns a list with two tagged components; the return value of a function is the last expression evaluated in the function body. The code executed by our function is enclosed in curly braces `{}`.

2. We can also create functions which can have multiple arguments, and default values for arguments.

```
raiseNumber <- function(x, power = 1) {
  x ^ power
}
raiseNumber(10)
raiseNumber(10, 3)
```

