Demo 1 part I: Introduction to R

Experimental and Statistical Methods in Biological Sciences I

September 15, 2014

1 Very basics

1.1 Command-line syntax

The basic way to interact with R is by typing commands into the R prompt. In this document, we'll refer to the command prompt using the '>' symbol, to match what you see on your screen. Type the number '10' into the R prompt and press return.

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

When you press return, R will evaluate what you have entered and return the result (the output), in this case '`[1] 10`', giving us back the number we entered in. (The '1' in brackets is a counter, and can be ignored.)

1.2 Calculations

R can also be used as a calculator. Type the following in to R

```
> 10 + 1
```

Question 1 What is the result?

Question 2 Try making calculations using -, *, and / . Do these work as expected?

Arithmetic operations can be strung together

```
> 10 + 1/2
[1] 10.5
```

Note the order in which the expression was evaluated. The 1 was divided by 2 before being added to 10. You can use brackets to guide R in evaluating the arithmetic in the order you want. Try entering

```
> (10 + 1)/2
```

Question 3 How does the output differ from the previous expression we entered?

Tip You can go back through the commands you've already entered into R using the up arrow key.

1.3 Variable assignment

Variables are used to store the result of an expression for later use. Assign variables using a greater-than symbol followed by a hyphen, `<-`, which looks like an arrow pointing to the left. On the left side of the arrow, you put the name you want the variable to have. To the right of the arrow, put the value you want the variable to have, like

```
> x <- 1
```

which assigns the value of 1 to the variable `x`.

Question 4 What happens when you enter in a variable you've just made on its own into the R prompt? What output does R give?

```
> x
```

Anything that you can type into the R prompt, you can assign to a variable

```
> y <- (20/5) + 100
> y
[1] 104
```

Notice that `y` only stores the final result of '(20 / 5) + 100'.

Question 5 Add the variables `x` and `y` together. Assign the result to a new variable called `z`.

There are a few simple rules and guidelines for naming variables.

- Variable names can be a mix of letters and numbers.

```
> abc <- 2 + 2
> result1 <- 20 * 200
```

- Variable names can also contain _ (underscore) and . (period)

```
> number_of_subjects <- 6
> number.of.subjects <- 6
```

Giving variables informative names helps you remember what they mean and makes the steps in your analysis easier to read and understand later.

Question 6 Make the assignment arrow go the "other way". For example, type '`100 -> a`' into the prompt. Explain what is happening.

2 Data types

2.1 Numeric

Numbers can be grouped together into a 'vector' which can be made in the following way

```
> c(1, 2, 4)
[1] 1 2 4
```

which is a vector with the values 1, 2, and 4. The 'c' is short for "combine".

Question 7 Type in `c(10, 20, 30) + 1`. What is the result?

Question 8 Assign the result of the previous expression to a variable.

You can generate regular sequence of numbers as a vector by placing a : (colon) between two numbers.

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

Question 9 Make a vector that goes from 40 to 100.

2.2 Characters

Another type of variable that you often work with is character. Characters are specified in R by putting them between ' ' (single quotes) or " " (double quotes).

```
> "Hello, World!"
[1] "Hello, World!"
```

Like numbers, characters can be combined into vectors

```
> c("a", "b", "c", "d")
[1] "a" "b" "c" "d"
```

and assigned to variables

```
> my.characters <- c("a", "b", "c", "d")
> my.characters # check what you stored
```

Character strings can be used for representing some non-numerical data (like sex or place names). They are also used to tell R the location of data to retrieve.

3 Data structures

3.1 Vectors

You have already encountered two ways to make a vector: `c` and `:`. There are also other efficient commands that make vectors, including `seq` and `rep`.

You can use `seq` to create a sequence in two ways. Try typing

```
> vector1 <- seq(1,20,by=0.5)
> vector2 <- seq(1,20,length=5)
```

Question 10 What is the difference between the two vectors and the use of named arguments (`by, length`) in defining the sequence?

With `rep`, you can repeat values to make, for instance, a vector. Again, there are multiple ways to do this:

```
> rep(1:5, times=2)
> rep(1:5, times=c(3,1,1,5,2))
> rep(1:5, each=3)
```

Question 11 Explain the differences in these three ways to create vectors.

Question 12 Test whether the same commands can also be used when creating vectors from characters:

```
> "a":"b"
```

3.2 Matrices

There are also several functions that make matrices. Try these:

```
> m <- 1:1 # create example vector m
> n <- seq(2,20,by=2)  # and example vector n
> matrix(m, nrow=2, ncol=5)
> matrix(m, nrow=2, ncol=5, byrow=TRUE)
> cbind(m,n)
> rbind(m,n)
```

Question 13 Explain how the functions `matrix, cbind, rbind` work.

3.3 Data frames

3.3.1 Creating and modifying data frames

Vectors can be combined into data frame using the `data.frame` function:

```
> city <- c("Panama", "Dodoma", "Tokyo")
> population <- c(408168, 180541, 8483000)
> Cities <- data.frame(city, population)
> Cities   # check what you stored in the data frame
```

Use the `$` to add a new variable to the data frame

```
> Cities$temperature <- c(29, 29, 21)
> Cities

    city population temperature
1 Panama  408168   29
2 Dodoma  180541   29
3  Tokyo 8483000    21
```

3.4 Reading in data

R can read data from a file on your computer or from the web. `read.csv` is a function that will read in CSV (comma-separated values) into R. We can grab a data set of information on sex, age, weight, height, and smoking history from a group of 100 subjects:

```
> read.csv("http://becs.aalto.fi/~heikkih3/age_weight.csv")
```

This will read in the data and print it out. It is, of course, convenient to save the result.

```
> AgeWeight <- read.csv("http://becs.aalto.fi/~heikkih3/age_weight.csv")
```

The result of `read.csv` is a data frame. A data frame is a matrix that organized variables into columns and cases into rows. You can get the names of the variables in the data frame using the `names` function.

```
> names(AgeWeight)
```

showing that the variables are `MALE`, `AGE`, `WEIGHT`, `HEIGHT`, `SMOKE1`, and `SMOKE2`.

3.4.1 Examining the data

The first step of analysis with new data is to get a sense of what the data look like and how the variables are coded. The function `head` will show you the first 6 rows of data

```
> head(AgeWeight)
```

Question 14 How is the variable `MALE` coded?

Question 15 What is the age of the 3rd participant?

Question 16 What does the `tail` function do?

Question 17 Use the help system to see how you can ask `head` to return more than 6 rows of a data frame.

3.4.2 The size of the data

The functions `dim`, `ncol`, and `nrow` will show the dimensions, number of columns, and number of rows of a data frame.

```
> dim(AgeWeight)
[1] 100 6
> ncol(AgeWeight)
[1] 6
> nrow(AgeWeight)
[1] 100
```

indicating that there are 6 columns (variables) and 100 rows (observations, i.e. participants).

Question 18 Is the first value returned by `dim` the number of columns or the number of rows?

3.4.3 Accessing variables

Individual variables in a data frame can be accessed using '$':

```
> AgeWeight$AGE
 [1] 58 62 59 64 55 53 30 38 30 57 61 60 47 49 26 26 56 29 59 46 47 34 64 41 35
[26] 41 47 21 26 58 41 49 51 59 65 58 40 53 23 27 43 20 51 26 42 37 63 54 57 42
[51] 59 33 65 22 27 52 38 33 26 23 48 60 21 60 28 45 57 54 62 46 26 47 23 21 35
[76] 62 36 42 43 24 38 35 36 27 32 25 46 53 38 49 21 49 47 59 27 28 27 30 55 24
```

which gives the ages of all the participants.

Question 19 What type of variable is `AgeWeight$AGE`?

3.4.4 More exercises

Question 20 Convert the column names to lower-case and save the result in a new data frame (e.g., `ageweight`). Try functions `tolower, names`. Tip: you have to call one function `tolower` within another `names`.

Question 21 Get a rough summary of the original data. Try functions `summary, head`. Describe what you see.

4 Subscripting

It is often necessary to use only part of your data (for example, running an analysis separately for males and females). There are several ways to get only a part of a vector or data frame.

4.1 Indexing with subscripts

Specific elements from a vector or rows and columns from a data frame can be retrieved using the '`[ ]`' (square bracket) notation. To get the first element of a vector[1]

```
> m <- c(48, 32, 78, 22, 16, 60)
> m[1]
[1] 48
```

You can get more than one element by putting a vector of indices between the brackets

```
> m[c(1, 3, 4)]
[1] 48 78 22
```

returns the 1st, 3rd, and 4th elements.

Question 22 Come up with two ways to retrieve the 2nd through the 5th elements of `m`.

Question 23 `length(m)` returns the number of elements in `m`. Use this to get the second to last element of `m`.

Specific columns and rows of data frames can also be retrieved by index.

```
> AgeWeight[2, ]
  MALE AGE WEIGHT HEIGHT   SMOKE1 SMOKE2
2 0  62 101.22 1.679851  no  no
```

selects the second row of the `AgeWeight` data frame. Note the use of the comma followed by nothing, which implicitly asks R to return all of the columns.

Question 24 How would you select the 2nd, 10th, and 21st rows?

[1] If you are familiar with other programming languages, note that R starts counting subscripts from 1, not from 0.

Question 25 How would you select rows 1, 2, and 30-45?

Columns are indexed using the space after the comma

```
> AgeWeight[, 1]
 [1] 1 0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 1 1 1 0 1 0 0 0 1 0 0 1 0 1 0 1 0 0 1 1
[38] 1 1 0 1 0 1 1 0 1 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0
[75] 1 0 0 1 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 1 0 0 0 0
```

Question 26 Retrieve the 3rd and 4th columns.

Question 27 What is the weight of the 54th participant?

4.2 Getting columns by name

Besides accessing a particular variable in a data frame using the `$` notation, a vector of column names can also be passed to a data frame.

```
> AgeWeight[,"MALE"]
> AgeWeight[, c("AGE", "WEIGHT")]
```

Question 28 Get a subset of the `AgeWeight` data frame where weights are in the first column and ages are in the second column.

Question 29 Get sex and age for the last 10 participants.

4.3 Logical subscripting

Parts of vectors and data frames can also be selected by finding values that meet particular conditions. The first thing to know is that R has two special constants, `TRUE` and `FALSE`. These values are returned by the logical comparison operators, `= =`, `<`, `>`, `<=` and `>=`. For example,

```
> 1 == 1
[1] TRUE
> 2 >= 3
[1] FALSE
x <- 4
x < 10
[1] TRUE
```

Question 30 Is $100 + \frac{35}{23 \times 3}$ greater than $110 - \frac{16}{5.5}$?

Using a logical comparison with a vector will return a vector of true and false values telling you whether each element meets the condition.

```
> m
[1] 48 32 78 22 16 60
> m >= 50
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Conveniently, putting a vector of `TRUE`s and `FALSE`s between the [ ]-brackets of a vector or data frame will return elements where there is a `TRUE`.

```
> c(4,1,10)[c(TRUE, TRUE, FALSE)]
[1] 4 1
```

Combining these two features together, you can grab parts of a vector that meet the specified condition

```
> m[m >= 50]
[1] 78 60
```

Question 31 Find all the participants in the `AgeWeight` data frame who are over 60 years of age.

4.4 More exercises

Use the AgeWeight data again.

Question 32 Access..

1. the 10th row in the 7th column
2. rows 4:5 and columns 2:3
3. rows 4:5 and all columns
4. Rows 4 and 6 and columns 1 and 3
5. all rows except for 4 and 5 in column 1

Question 33 Like in the previous exercise, but now address the columns in the data frame by their names rather than their numerical values. Simply look up the column names of the data frame by using the functions `names` or `head`. To index the columns, you will need to use character strings or vectors of character strings as subscripts. Character strings need to be enclosed by quotes, e.g. "MALE". Access..

1. the 10th row in the 7th column
2. rows 4:5 and columns 2:3
3. rows 4 and 6 and columns 1 and 3

Question 34 Filter out the data that are available for male (MALE=1). Store the result in a new data frame "age_weight.male". Tip: `AgeWeight[rows for males only, all columns]`. When subscripting the rows, you need to use a logical expression. The easiest way of addressing a column name is by using the dollar sign $ symbol.

5 Operations on your data

5.1 Vector arithmetic

Question 35 Create a vector x containing the following numbers: 0, 2, 4, 6, 8, 10. Create a vector y containing the following numbers: 1, 3, 5, 7, 9, 11. Obtain the sum of both vectors. `x+y` Describe what happened. Try also with other arithmetic operations (-, *, /, ^).

The same logic can be applied to data frames:

Question 36 Calculate the body mass index (BMI) for each participant. The BMI is the weight divided by height squared:

$\text{BMI} = \frac{\text{weight (kg)}}{(\text{height (m)})^2}$

Store the results in a new column called "BMI".

Tip: the easiest way of addressing a column name is by using the dollar sign symbol: `dataFrame$newColumnNam`

5.2 Functions

R contains a vast array of methods for manipulating and plotting data and running statistical tests. These methods are packaged together as functions, which are procedures or routines that take input and produce output. For example, you can sum a list of numbers using

```
> sum(1, 2, 3, 4)
[1] 10
```

which gives back the output (or answer) of 10. Functions consist of two parts,

1. The function name, in this case `sum`,
2. a list of arguments to the function, separated by commas and surrounded by brackets.

Question 37 What is the result of passing a variable or a vector as an argument to `sum()`?

Question 38 Look back at previous sections. Which functions have you already used?

There are also functions that don't take any arguments. For example, the function `ls` will return a list of variable names that you've used.

```
> ls()
[1] "abc" "number_of_subjects" "number.of.subjects"
[4] "result1" "x"  "y"
[7] "z"
```

Tip As you progress, you will be entering more complex expressions into R. Sometimes you will forget to have a closing brace or quotation mark so when you press return, R is still expecting more input. This will be indicated by the `>` on the prompt changing to a `+`. I fthis happens, you can keep typing. To cancel out what you've typed, press the escape key on Windows or Ctrl-C on OS X or Linux.

5.2.1 Getting help on a particular function

R has a built-in documentation system that will give you information about what a function does. Access it by putting a question mark before a function name

```
> ?sum
```

5.2.2 "There's a package for that"

When you don't know how to do something in R, the internet is the best resource to find information. Two good websites are `http://www.statmethods.net` and `http://rseek.org`. Using these resources or others at your disposal (such as the built-in help system, asnwer the following questions:

Question 39 What does the `sort` function do? What types of variables can you pass as arguments?

Question 40 Not all functions are accessible by default. What are "packages" in R? How do you use them?

5.3 Creating your own functions

R has an impressive range of functions built in that allow you to perform many operations, e.g., `mean`, `mode`, `median`, etc. Sometimes, however, you may want to do something for which R doesn't provide a function. In such cases R allows you to write your own function (as any programming language will let you do).

The full syntax will look like this:

```
name <- function(argument 1, argument 2, ...) {expression}
```

Here `name` is the function's name, `argument` is an object passed to the function (that is, an object on which the function will act), and `expression` is a set of R commands defining the function. For example,

```
> Mean = function(X) {
+ M = sum(X)/length(X)
+ return(M)
+ }
```

This function called `Mean` is simply taking the sum of `X` and dividing it by the length of `X` amd calling it `M`. The function then returns the value of `M`. Once defined, the function can now be used to find the mean of a vector by typing `Mean(X)` just like any other function.

```
> Z = c(10, 22, 17, 18, 24, 11)
> Mean(Z)
[1] 17
```

returning the correct answer, 17.

Question 41 Using these principles, can you write another simple function that finds the range of a vector? Give it the name `range.size`.

Tip Instead of using `sum` and `length`, here you will need to use the function `max` and `min`, although not necessarily in the same way as with the `Mean` function. (`max` and `min` give you maximum and minimum score of a vector.)

Tip You can put multiple expressions on the same line in R by separating the with at ; (semi-colon).

```
> gt.50 <- m > 50; m[gt.50]
```

This can be handy for writing more compact functions[2]

```
> Mean <- function(X) {M = sum(X)/length(X); return(M)}
```

Question 42 Test your home-made function `range.size` on the AgeWeight data: Calculate the difference between the minimum and maximum weight for non-smoking (based on 1st measurement) males.

Question 43 Test your `range.size` function with the following statements:

```
> range.size(c(1,5,10)) == 9
> range.size(-234:119) == 353
> range.size(c(14.3, 5.6, 22.1) == 16.5
```

Do they all evaluate as `TRUE`?

Question 44 Using your function, create a new data frame that lists the differences between the minimum and maximum height, minimum and maximum weight, and minimum and maximum BMI. The data frame should have three columns.

Question 45 Store the result from Question 44 in a new variable called "AgeWeightDifferences". Now print the values associated with this data frame into a file called "age_weight_differences.csv", using the function "write.csv".

[2]However, you also want to write code that is easy to read later, so try to break up your analysis into multiple steps instead of putting everything on one line.