Basic Computing 1 — Introduction to R*

Stefano Allesina and Peter Carbonetto *University of Chicago*

The aim of this workshop is to introduce `R` (and `RStudio`), and show how it can be used to analyze data in an automated, replicable way. We will illustrate the notion of assignment and present the main data structures available in `R`. We will learn how to read and write data, how to execute simple programs, and how to modify the stream of execution of a program through conditional branching and looping. *This workshop is intended for biologists with little to no background in programming.*

Setup

To complete the tutorial, we will need to install `R` and `RStudio`. To install `R`, follow instructions at `cran.rstudio.com`. Then install `Rstudio` following the instructions at `bit.ly/2JARd4v`.

Motivation

When it comes to analyzing data, there are two competing paradigms. First, one could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; second, one could write programs that can be run to perform the analysis of the data, the generation of tables and statistics, and the production of figures automatically.

This latter approach is preferred because it allows for the automation of analysis, it requires a good documentation of the procedures, and is completely replicable.

A few motivating examples:

- You have written code to analyze your data. You receive from your collaborators a new batch of data. With simple modifications of your code, you can update your results, tables and figures automatically.
- A new student joins the laboratory. The new student can read the code and understand the analysis without the need of a lab mate showing the procedure step-by-step.
- The reviewers of your manuscript ask you to slightly alter the analysis. Rather than having to start over, you can modify a few lines of code and satisfy the reviewers.

Here we introduce `R`, which can help you write simple programs to analyze your data, perform statistical analysis, and draw beautiful figures.

What is R?

`R` is a statistical software that is completely programmable. This means that one can write a program ("script") containing a series of commands for the analysis of data, and execute them auto-

*This document is included as part of the Basic Computing 1—Introduction to R tutorial packet for the BSD qBio Bootcamp, MBL, 2019. **Current version**: July 31, 2019; **Corresponding author**: sallesina@uchicago.edu.

matically. This approach is especially good as it makes the analysis of data well-documented, and completely replicable.

R is free software: anyone can download its source code, modify it, and improve it. The R community of users is vast and very active. In particular, scientists have enthusiastically embraced the program, creating thousands of packages to perform specific types of analysis, and adding many new capabilities. You can find a list of official packages (which have been vetted by R core developers) at goo.gl/SOSDWA; many more are available on GitHub, Bioconductor, and other websites.

The main hurdle new users face when approaching R is that it is based on a command-line interface; when you launch R, you simply open a console with the character > signaling that R is ready to accept an input. When you write a command and press Enter, the command is interpreted by R, and the result is printed immediately after the command. For example,

```
2 * 3
# [1] 6
```

A little history

R was modeled after the commercial statistical software S by Robert Gentleman and Ross Ihaka. The project was started in 1992, first released in 1994, and the first stable version appeared in 2000. Today, R is managed by the *R Core Team*.

RStudio

For this introduction, we're going to use RStudio, an Integrated Development Environment (IDE) for R. The main advantage is that the environment will look identical irrespective of your computer architecture (Linux, Windows, Mac). Also, RStudio makes writing code much easier by automatically completing commands and file names (simply type the beginning of the name and press Tab), and allowing you to easily inspect data and code.

The main RStudio interface is split up into "panels". The most important panels are:

1. **Console:** This is a panel containing an instance of R. For this tutorial, we will work mainly in this panel.
2. **Source:** In this panel, you can write a program and save it to a file pressing `control + S` (or `command + S`). The code can also be executed by pressing `control + shift + S` (or `command + shift + S`).
3. **Environment:** This panel lists all the variables you created (more on this later).
4. **History:** This gives you the history of the commands you typed.
5. **Plots:** This panel shows you all the plots you drew. Other tabs allow you to access the list of packages you have loaded, and the help page for commands (just type `help(name_of_command)` in the Console) and packages.

How to write an R program

An `R` program is simply a list of commands, which are executed one after the other. The commands are witten in a text file (with extension `.R`). When `R` executes the program, it will start from the beginning of the file and proceed toward the end of the file. Every time `R` encounters a command, it will execute it. Special commands can modify this basic flow of the program by, for example, executing a series of commands only when a condition is met, or repeating the execution of a series of commands multiple times.

Note that if you were to copy and paste (or type) the code into the **Console** you would obtain exactly the same result. Writing a program is advatageous, however, because it can be automated and shared with other researchers. Moreover, after a while you will have accumulated a lot of code, at which point you can reuse much of your code for new projects.

Activity: Collaborative data collection

In this activity, you and the rest of the class will collaborate on creating a data set. Specifically, we will collect data about the laptops of incoming BSD graduate students. We will use these data to learn about R, and how it can be used to analyze a data set; we will write some R code that can be used to answer basic questions such as, what is the most common operating system among the population (where "population" is all the current incoming BSD grad students)? This is also an opportunity to learn about your laptop's hardware.

The instructor will provide you with further instructions.

Activity: Formulating the questions

Before writing any R code to import and analyze the data set, you first need to determine the aims of working with this data set. In particular, we need to formulate some *guiding questions*. For example, some very basic questions we might want to know are:

- How many samples have been collected?
- What are the different operating systems used?

Or, some more complicated questions we could ask are:

- Which is the most common (and least common) operating system?
- On average, do Mac laptops have more (or less) memory than Windows laptops, and how much more (or less)?

Write down a few more questions guiding your data analysis:

Reading in the data

Your R environment should be empty:

```
ls()
```

Your first step is to create a new object in our R environment that contains the data from the CSV file. Let's call this object "laptops". Write down your R code for reading in the data:

If this was successful, your environment should now contain a single object, "laptops":

```
ls()
```

Inspecting the data

What kind of object is it?

```
class(laptops)
```

It should be a "data frame".

Data frames contain data organized in a table or spreadsheet. The columns typically represent different measurements, and they can be of different types (e.g., date of measurement, weight of individual, volume of cell, treatment of sample). The rows typically represent different individuals or samples.

When you read a spreadsheet or CSV file in R, it is automatically stored as a data frame.

Run some basic commands to inspect the table, and check that the data were read correctly. After running each line of code, add a note next to the line of code summarizing what it did:

```
nrow(laptops)
ncol(laptops)
length(laptops)
head(laptops)
tail(laptops)
summary(laptops)
str(laptops)
```

Accessing rows and columns

There are many ways to inspect parts of the data frame. Here are some examples—as before, add a note next to each line of code explaining what it did.

```
laptops[4, 2]
laptops[4, "os"]
laptops[4, ]
laptops[ ,2]
laptops[2]
laptops[,"os"]
laptops$os
laptops["os"]
laptops$os[4]
laptops[4, ]$os
```

Here are some slightly more complicated examples:

```
laptops[1:4, ]
laptops[ ,2:3]
laptops[ ,c("os", "mem")]
laptops[c("os", "mem")]
laptops[c(1:4, 8, 12), c("os", "cores")]
```

Creating—and overwriting—objects

The most basic operation in any programming language is assignment. In R, assignment is marked by the operator <-. Above, we used the assignment operator to create a new object, laptops, containing the data from the CSV file.

When you type a command in R, it is executed, and the output is printed in the **Console**. For example, run:

```
sqrt(9)
```

If we want to save the result of this operation, we can assign the result to an object, or variable. Let's call this new object "x":

```
x <- sqrt(9)
x
```

What happened? We wrote a command containing an assignment operator (`<-`). `R` evaluated the right-hand side of the command (`sqrt(9)`), and stored the result (`3`) in a newly created variable called `x`. Now we can use `x` in our commands: every time the command needs to be evaluated, the program will look up which value is associated with the variable `x`, and substitute it. For example, we can run:

```
2 * x
```

We can then use `x` to create a new object storing the result of multiplication:

```
y <- 2 * x
y
```

Or we can *overwrite* the existing value of `x`:

```
x <- 2 * x
x
```

Observe that both assignment (creating new objects) and overwriting both use `<-`. So be careful—*there is no undo command in R!*

Next, we will use the assignment operator to extract individual columns from the data frame, and understand these columns in more detail.

Side note: "<-" versus "="

A common point of confusion is that the equality symbol (`=`) *can also be used for creating and overwriting objects.* And many people prefer to use this.

But we strongly recommend against using `=` for assignment. We have two reasons:

1. The `=` is easily confused with `==`, which does something completely different (as we will see soon)—it is used to check that two values are equal.
2. The `=` is also used elsewhere for named arguments to functions (we saw one example of this when we called `read.csv`). This use of "=" in this case is *not* assignment (consider that no objects are created or overwritten in your environment).

Basic data types and operations: the basic building blocks of R

A data frame is a *composite* data structure—it is made of simpler data objects which can be thought of as `R` "building blocks". Here we will take a close look at these building blocks.

Text data

What columns of `laptops` contains text data? Write a line of code to select a text column, and assign it to a new variable, `x`:

Then check the data type of `x`:

```
class(x)
```

In `R`, the `character` data type is used to store text.

You can access individual elements by their index: the first element is indexed at 1, the second at 2, *etc.*

```
x[1]
x[5]
```

`R` has many built-in functions for operating on text data. Here are some examples (add your notes explaining what each line of code does):

```
nchar(x)
toupper(x)
sort(x)
unique(x)
table(x)
paste(x[1], x[2])
```

Just as data types can be combined to form more complex data structures, operations can also be combined:

```
unique(toupper(x))
```

Numeric data

What columns of `laptops` contain numeric data? Write code to assign a numeric column to `x`, and check its type:

The `integer` data type stores whole numbers.

There is another data type, `numeric`, used to store real numbers (or rather their approximations, as computers have limited memory and thus cannot store numbers like π, or even 0.2), e.g.,

```
y <- x / 10
class(y)
```

The numeric data type can store very small or very large numbers (what does the ^ operator do?):

```
x^10
```

Like characters, `R` has many built-in functions for working with numbers, such as:

```
abs(x)
sqrt(x)
cos(x)
log(x)
```

Some functions allow different data types:

```
sort(laptops$mem)
sort(laptops$os)
table(laptops$mem)
table(laptops$os)
```

Given that `R` was born for statistics, there are many statistical functions you can perform on numeric data (add notes next to these lines explaining what they do):

```
min(x)
max(x)
range(x)
sum(x)
median(x)
mean(x)
summary(x)
```

Finally, standard mathematical operations such as + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation) can be applied to numbers. A possibly unfamiliar operator is the modulo (%%), calculating the remainder of an integer division:

```
x %% 3
```

Question: What is the modulo operator, and could it be useful?

Exercise: Combine some of the functions and/or operators mentioned above to compute the *standard deviation* of `x`, then compare your answer against R's built-in solution, `sd(x)`. *Hint:* Use the definition of variance. Write your answer in the box below.

Logical data

The `laptops` data set does not contain any logical data, but we can easily generate logical data from one of the columns. Write code to create an object `x` containing logical data, and check the type of `x`:

The "logical" data type takes only two possible values `TRUE` and `FALSE`.

Question: Why are two equal signs used to check that two numbers are equal?

Exercises:

1. Besides equality, other comparison operators include > (greater than), < (less than), `!=` (differs) and >= and <=. Write code using one or more of these operators to create some other (hopefully useful!) logical data:

2. Additionally, you can formulate more complicated logical statements (perhaps using multiple variables or columns of a data frame) using `&` (and), `|` (or), and `!` (not). Write code using these operators, as well as the ones above, to generate some new (and hopefully useful!) logical data:

Using numeric data to select rows of a data frame

Above, we saw that rows and columns can be selected by numberx. The row and column numbers are just numeric data—therefore, *you can think of the selection of rows as data*. Let's take an example: suppose you would like to inspect only data on Mac laptops. You can do this by using the `which` function to create a new numeric data object containing the row numbers corresponding to Mac laptops:

```
rows <- which(laptops$os == "mac")
laptops[rows, ]
```

Exercise: Create a new `rows` object based on a logical expression involving two columns of the `laptops` data frame. Write two different ways of implementing this row selection:

Categorical data: factors

There is one last basic data type that isn't present in the `laptops` data. But we can easily create it using a column of the data frame. This last data type is a *factor*. Write down your R code for creating a factor:

Inspect the factor:

```
x
summary(x)
```

Question: What is a factor? When would a factor be useful?

Manipulating data

So far, after reading in the data, we have treated the data frame as a static object. But the data frame is an object like any other in R—it can be modified and overwritten. This is a very powerful—but also very dangerous—idea! Let's illustrate a few of the many kinds of data manipulations we can make, but before doing so, let's create a copy of the data frame in case we make a mistake along the way:

```
laptops2 <- laptops
```

You can overwrite individual entries of the data frame, or several entries at once:

```
laptops2[c(1, 2), "mem"] <- 2
```

Write your code to replace all "mac" values in the "os" column with "macOS":

Above, we found that factors are sometimes useful. Replace the text data in the "os" column with categorical data:

```
laptops2$os <- factor(laptops2$os)
```

If you don't like the column names, you can change them:

```
colnames(laptops2) <- c("name", "OS", "GB", "CPUs")
```

You can reorder the rows or columns:

```
rows     <- order(laptops2$name)
laptops2 <- laptops2[rows, ]
laptops2 <- laptops2[c("name", "CPUs", "GB", "OS")]
```

Or you can even create new columns:

```
n <- nrow(laptops2)
laptops2$sample <- seq(1, n)
```

Save the state of your environment

It is important to periodically save the state of your R environment. Let's do this now.

To save your environment, go to **Session > Save Workspace As** in RStudio, or run this code:

```
save.image("mbl_tutorial.RData")
```

Later, to restore your environment, select **Session > Load Workspace** in RStudio, or run this code:

```
load("mbl_tutorial.RData")
```

Question: In RStudio, what is the difference between **File > Save As** and **Session > Save Workspace As**? Record your answer here:

Group activity: Data analysis, Part 1

You now have the basic tools you need to analyze a data set in R. Return to the guiding questions you (and others) have formulated. In this activity, you will apply your newly acquired R skills to answer some of these questions. (Some of these questions may have already been answered as we worked through the examples above.) Record your answers, as well as the code you wrote to generate these answers.

Group activty: Data analysis, Part 2

Use your code from Part 1 to create a *script*—a text file containing all the code necessary to run the complete data analysis, starting with the `read.csv` call to import the data, and ending with the code used to generate the answers to your guiding questions. Give it a useful name such as `analyze_laptops_data.R`. The purpose of a script is to record your code so that you can easily repeat, or *automate*, the analysis later.

Once you have written this script, make sure that it runs, and save the file somewhere on your computer. Please also add comments (lines starting with "#") explaining in plain language what the code does. The instructor will give you instructions for sharing your script with the rest of the class.

Other topics

Creating data

Above, we imported data into R. In addition to viewing and manipulating existing data, R also has many facilities for creating data structures, either from scratch, or from existing data. (We already saw a few examples of this.)

The most basic tool is the `c` function, which is short for "combine", and can combine multiple objects or values:

```
x      <- c(2, 3, 5, 27, 31, 13, 17, 19)
sex    <- c("M", "M", "F", "M", "F")           # Sex of Drosophila
weight <- c(0.230, 0.281, 0.228, 0.260, 0.231) # Weight (in mg)
```

You can generate sequences of numbers using `seq`. Write R code that uses the `seq` function to generate all odd numbers from 1 to 100:

For simpler sequences of numbers, use the "colon" operator:

```
x <- 1:10
```

To repeat a value (or values) several times, use `rep`:

```
x <- rep("treated", 5)  # Treatment status
x <- rep(c(1, 2, 3), 4)
```

Finally, there are many functions for generating random numbers. For example,

```
x <- runif(100)
```

What des `runif(100)` return?

Matrices

A matrix is like a data frame—it also has rows and columns—but a key difference is that all the columns must be of the same type. Here's an example of a 2 x 2 matrix:

```
A <- matrix(c(1, 2, 3, 4), 2, 2)  # Inputs are values, nrows, ncols.
```

In the case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, *etc*).

```
A %*% A         # Matrix product
solve(A)        # Matrix inverse
t(A)            # Transpose
A %*% solve(A) # This should return the identity matrix.
```

Determine the dimensions of a matrix:

```
nrow(A)
ncol(A)
dim(A)
```

Use indices to access a particular row or column of a matrix:

```
Z <- matrix(1:9, 3, 3)
Z[1, ]              # First row.
Z[ ,2]              # Second column.
Z[1:2, 2:3]         # Submatrix with coefficients in rows 1 & 2, and columns 2 & 3.
Z[c(1, 3), c(1, 3)] # Indexing non-adjacent rows and columns.
```

Some operations apply to all the elements of the matrix:

```
sum(Z)
mean(Z)
```

Question: When is it better to store data in a matrix instead of a data frame?

Arrays

If you need tables with more than two dimensions, use arrays:

```
A <- array(1:24, c(4, 3, 2))
```

A matrix is a special case of an array with only 2 dimensions.

You can still determine the dimensions with `dim`:

```
dim(A)
```

And you can access the elements as for matrices. One thing you should be paying attention to: `R` drops dimensions that are not needed. So, if you access a "slice" of a 3-dimensional array, you should obtain a matrix:

```
A[ ,2, ]
dim(A[ ,2, ])
class(A[ ,2, ])
```

Lists

You have already worked with a list object (probably without realizing it). The laptops data frame is also a list:

```
is.list(laptops)
```

A data frame is a specific type of list—it is a list in which every list item is a vector of the same length. Lists also allow vectors of different lengths, for example, here we add an additional item ("date") that has a length of 1:

```
mylist <- as.list(laptops)
mylist$last_modified <- "September 9, 2019"
```

We could not do this in a data frame.

A list is also a very general data structure for creating complex objects, and the list elements can be used to store almost anything.

Missing data

Finally, R allows most types of data—text, numeric, factors, *etc*—to take on a special value, `NA`. This is short for "not available" or "not assigned", and is commonly called "missing data". One special feature of R is that it often gracefully handles data sets containing missing data.

Exercise: Set a few entries in the `laptops` data frame to `NA`, then run `summary(laptops)`. How are the missing data reported in the summary?

Group activity: Analyzing genetic data (time permitting)

In this activity, you will apply the tools we have developed so far to look at an example data set from human genetics: genotype data on chromosome 6 collected from European individuals. (Data adapted from the Human Genome Diversity Project by John Novembre.)

```
chr6 <- read.table("H938_Euro_chr6.geno", header = TRUE)
```

Setting `header = TRUE` means that we want to take the first line to be a header containing the column names. How big is this table?

```
nrow(chr6)
ncol(chr6)
```

It contains 7 columns, and more than 40,000 rows!

The table reports the number of homozygotes (`nA1A1`, `nA2A2`) and heterozygotes (`nA1A2`), for 43,141 single nucleotide polymorphisms (SNPs) obtained by sequencing European individuals. The other columns are:

- `CHR`: The chromosome (in this case, 6).
- `SNP`: The identifier of the Single Nucleotide Polymorphism.
- `A1`: One of the observed alleles.
- `A2`: The other allele.

Write R code to answer the following questions:

1. How many individuals were sampled? Find the maximum of the sum `nA1A1 + nA1A2 + nA2A2`. *Hint:* Recall that you can access the columns by index (e.g., `chr6[ ,5]`), or by name (e.g., `chr6$nA1A1`, or `chr6[ ,"nA1A1"]`).
2. Use the `rowSums` function to obtain the same answer.
3. How many SNPs have no heterozygotes (*i.e.*, no "A1A2")?
4. How many SNPs have less than 1% heterozygotes?

Programming Challenge

Instructions

You will work with your group to solve the following exercise. When you have found the solution, go to `stefanoallesina.github.io/BSD-QBio5` and follow the link "Submit solution to challenge 1" to submit your answer (alternatively, you can go directly to `goo.gl/forms/dDJKvF0dOi7KUDqp1`. At the end of the boot camp, the group with the highest number of correct solutions will be declared the winner. If you need extra time, you can work with your group during free time or in the breaks in the schedule.

Nobel nominations

The file `nobel_nominations.csv` contains data on Nobel Prize nominations from 1901 to 1964. There are three columns (the file has no "header"): (1) the field (e.g., "Phy" for physics), (2) the year of the nomination, and (3) the id and name of the nominee.

1. Take Chemistry (`Che`). Who received the most nominations?
2. Find all researchers who received nominations in more than one field.
3. Take Physics (`Phy`). Which year had the largest number of nominees?
4. For each field, what is the average number of nominees per year? Calculate the number of nominees in each field and year, and take the average across all years.

Hints:

- You will need to subset the data. To make operations clearer, it is helpful to give names to the columns. For example, suppose you imported the data into a data frame called `nobel`. Then `colnames(nobel) <- c("field", "year", "nominee")` should do the trick.
- The simplest way to obtain a count from a vector is to use the `table` function. For example, the command `sort(table(x))` produces a table of the occurrences in `x`, in which the counts are sorted from smallest to largest.
- You can also use the same function, `table`, to build a table using more than one vector. For example, suppose `x` and `y` are vectors of the same length. Then `table(x,y)` will build a table with counts for each unique pair of occurrences in `x` and `y`.
- Some other functions you may find useful for the challenge: `colMeans`, `factor`, `head`, `length`, `max`, `read.csv`, `subset`, `tail`, `tapply`, `unique`, `which` and `which.max`.
- Save your solution code for each exercise in a file.