Basic Computing 1 — Introduction to R, Part 1*

Peter Carbonetto and Aarti Venkat *University of Chicago*

The aim of this workshop is to introduce R and RStudio, and to use R to analyze data interactively. We will focus on one particularly widely used data structure, the *data frame.* We will learn how to import tabular data into a data frame, and we will learn how to inspect, manipulate and analyze the data frame. This workshop is intended for biologists with little to no background in programming.

About this tutorial

In this tutorial, we will learn about the basic elements of R through analysis of a small data set. You are welcome to try this tutorial on your own, before or after the in-class tutorial. To try the examples before the data have been collected in class, you may use the data from previous years, stored in files `laptops_2019.csv`, `laptops_2020.csv` and `laptops_2021.csv`. To use these data, make a copy of one of these files, and call it `laptops.csv`.

Do I have what I need?

To follow the examples in this tutorial, you 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`.

From GitHub, download the tutorial materials to your computer, and make sure you know where to find them.

Now you can launch RStudio.

Where am I?

Make sure your R working directory in is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this.

Quit applications that are not needed and other "clutter" to reduce distractions. *Remember, the computer is device of distraction.*

Also, it is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu.

*This document is included as part of the Basic Computing 1—Introduction to R tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2022. **Current version**: August 11, 2022; **Corresponding author**: pcarbo@uchicago.edu. Thanks to Stefano Allesina, John Novembre, Stephanie Palmer and Matthew Stephens for their guidance.

A data-centric view of programming

There are many programming languages. Some of you may have experience with one of them, or more than one. R belongs to the modern family of *interactive programming languages* (e.g., Python, MATLAB, Julia), and was built on the revolutionary programming language Scheme developed at MIT. R inspires a "data-centric" view of programming, in which we focus on *data structures* (the objects) rather than procedures.

A brief tour of RStudio

For this introductory tutorial, we will use RStudio. It is an Integrated Development Environment (IDE) for R. One advantage of RStudio is that the environment will look identical irrespective of your computer architecture (Windows, Mac, Linux). RStudio also has many features that make writing code easier.

The main RStudio interface is split up into "panels", including:

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 your code and save it to a file. The code and also be run from this panel, but the actual results show up in the console.
3. **Environment:** This panel lists all the variables (objects) you created in your R environment.
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 (e.g., type `help(p.adjust)` in the console).

Activity: Collaborative data collection

To perform a data analysis, we need a data set to analyze. We will collect the data collaboratively.

Specifically, we will collect data about the laptops of incoming BSD graduate students. Once we have collected the data, we will write some R code to answer basic questions such as: what is the most common operating system among the population (where "population" is the cohort of incoming BSD grad students)?

We will collect the data in the Etherpad. We will record four data points: first/given name, operating system, amount of memory (in GB), and number of processors (CPUs). We will enter these data into a table with four columns. We will use commas to separate the columns in the table. The first few lines of the table will look something like this:

```
name,os,mem,cpus
rishi,mac,8,4
anna,windows,4,1
peter,mac,16,4
```

Ask your teammates or neighbors for help if you are unsure how to find this information about

your laptop.

Once we have collected the data from the entire class, I will create a CSV file, `laptops.csv`, and share it with you on Box.

S. cerevisiae

Import the data into R

You should now have a CSV file on your computer, `laptops.csv`, containing the data we collected. (What is CSV short for?) The basic R function for importing data from a CSV file is called `read.csv`:

```
laptops <- read.csv("laptops.csv",comment = "#",stringsAsFactors = FALSE)
```

In time, we will understand what this code is doing. For now, we will jump ahead.

This command will only work if your R working directory is the same as the directory containing `laptops.csv`. You can check your working directory with `getwd()` and `list.files()`. If you are in the wrong directory, use **Session > Set Working Directory** from the RStudio menu bar to move to the right directory.

Whenever you create an object, you need to give it a name. A good name reminds you of its contents and its purpose.

Assuming you started with a clean workspace, your workspace, or "environment", should now contain a single object, `laptops`:

```
ls()
```

This object stores the output of our `read.csv` call. Invoking its name will print its contents:

```
laptops
```

Or you can explicitly type

```
print(laptops)
```

which does the same thing.

What *kind* of object is it?

```
class(laptops)
```

It is a *data frame*. This is R's way of saying "tabular data" or "spreadsheet".

If you are used to the "point-and-click" way of doing things in software such as Excel, it may seem burdensome to have arrived at this point where we have written a bunch of code, and all we have

done is printed the contents of the table to the screen. But now that we have created a data frame, there are many things we can do it.

Note the advantages of R are less striking when working with a small data set. We will work with a larger data set in Part 2.

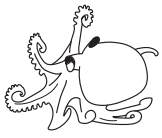
Inspect the data

R has many commands that are easy to use and quickly give you insight into the data. Let's try some of the more commonly used commands for inspecting and summarizing the contents of a data frame.

Write your code here.

Some of these commands, like `head` and `summary`, are "generic" functions, meaning that they work for many types of objects.

Now we will proceed to learn about R by trying to answer some basic questions about laptops. The first questions can be answered with very simple code, and then the code will get progressively more complicated.



O. bimaculoides

Simple questions

What is the largest and smallest amount of memory?

Write your code here.

What is the most and least number of CPUs?

Write your code here.

What is the most common CPU amount, the most common memory amount, and the most common combi-

nation of CPUs and memory?

> *Write your code here.*

T. thermophila

More difficult questions

Does anyone in Basic Computing 1 share the same name?

> *Write your code here.*

Are all the laptops multicore (more than one processor)?

> *Write your code here.*

Who has the most CPUs?

> *Write your code here.*

How many different OSs are there, and which OS is used most?

Write your code here.

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

This ability to overwrite and have multiple data objects floating around in your environment underscores the importance of *naming your objects well* and *keeping track of what is in your environment*; names such as "x", "y", "temp" or "data", while commonly used, should only be used for temporary or "throwaway" calculations. It isn't unusual to have half a dozen different copies of a data set over the course of an analysis. Another important practice is to document your code and periodically save your results in case you do accidentally overwrite something important!

Among Mac laptops (or Windows laptops), what is the average amount of memory?

Write your code here.

Hypothesis: Windows laptops tend to have more memory than Mac laptops. *True or false?*

Write your code here.

Save your work

This is a good time to save our work. To save your results, go to **Session > Save Workspace As** in RStudio, or run

```
save.image("basic_computing_1.RData")
```

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

```
load("basic_computing_1.RData")
```

What is the difference between **File > Save As** *and* **Session > Save Workspace As**?

C. jacchus

Deconstructing the data frame

Let's now take a second look at the laptops data frame, with a greater intention toward understanding R.

The data frame is an example of a *composite data structure*—that is, it is made of simpler data objects. These "atomic" data objects are R's "building blocks" for all other data structures. R has a very large variety of composite data structures, but the data frame may be the most popular.

In a data frame, the *columns are the atomic data objects*; in the next sections, we will closely examine the individual columns of the laptops data frame.

Text data

Let's begin with the "os" column:

```
x <- laptops$os
print(x)
```

In R, the "character" type is used to store text data:

```
class(x)
```

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

```
x[1]
x[2]
```

R has many built-in functions for operating on text data. Here are some examples:

```
nchar(x)
toupper(x)
sort(x)
unique(x)
paste(x[1],x[2],x[3])
```

Which of these functions are specific to text data?

Just as data types can be combined to form more complex data structures, operations can also be combined (provided the combination makes sense, of course!). For example,

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

is effectively equivalent to

```
y <- unique(x)
toupper(y)
```

Numeric data

The "mem" column is an example of numeric data:

```
x <- laptops$mem
class(x)
```

Specifically, it is an integer type since all the numbers are 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 / 100
class(y)
```

Like the character data type, a numeric object is a vector, and individual elements can be accessed their index,

```
x[1]
x[2]
```

Also, like the character data type, R has many built-in functions for working with numbers, such as

```
sum(x)
max(x)
range(x)
sqrt(x)
cos(x)
log(x)
```

Notice that, for example, `log(x)` computes the logarithm for *each data point in* `x`, and the result is *a vector of the same length as* `x`. This allows large calculations to be accomplished very simply; for example, if `x` contains 10 million numbers, `y <- sqrt(x)` will compute the square root of these 10 million numbers, and store the result in vector `y`.

Given that R was born for statistics, there are many statistical operations—from basic to sophisticated—that you can perform on numeric data:

```
median(x)
var(x)
quantile(x)
```

Finally, standard mathematical operations such as `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `^` (exponentiation) can be applied to numbers.

For more complex calculations, the mathematical and statistical operations can be combined, for example,

```
y <- log(x + 0.1)
```

Logical data

The laptops data frame does not contain logical data. But we can generate logical data from one of the columns:

```
x <- laptops$os == "mac"
print(x)
class(x)
```

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

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

Write your code here.

Exercise 2: You can also formulate more complicated logical statements (perhaps using multiple variables or columns of a data frame) using `&` (and), `|` (or), and `!` (not). Write an expression using these operators, as well as the ones above, to create some new logical data.

Write your code here.

E. coli

Activity: what is a "factor"?

So far, we have gotten familiar with three basic data types: character, numeric and logical. There is a fourth important atomic data type in R: *the factor.* Unknowningly, we have already used a factor in the examples above.

What is unusual about factors is that there is no equivalent of factors in other popular programming languages, at least not as a native data type. And yet you will find that they are extremely useful.

None of the columns in the laptops data frame are a factor. But, like the logical data type, we can create a factor from other data types. Let's try this with both the "mem" and "os" columns:

```
x <- factor(laptops$mem)
y <- factor(laptops$os)
class(x)
class(y)
```

Now run the following lines of code:

```
print(x)
print(y)
summary(x)
summary(y)
as.numeric(x)
as.numeric(y)
```

Question: Based on the results from running thiis code, what do you think a factor is?

More questions: Which representation to do you prefer for "mem", numeric or factor? What about the "os" column? Are other columns good candidates for being factors?

These questions—should I convert my data to a factor—touch on the broader question of *data representation* or *data encoding*. Choosing the right representation of your data can be important to an effective data analysis. (You might find that factors are useful in the programming challenge.)

If you prefer the "os" column to be a factor, for example, you can change it inside the data frame by doing

```
summary(laptops)
laptops$os <- factor(laptops$os)
summary(laptops)
```

C. elegans

Activity: Analyze laptop trends over time

Having gotten a bit of a feel for how to analyze data in R, we will now use the laptops data collected by BSD grad students over several years to *explore trends in laptops over time.* This is a freeform activity that will require some creativity and teamwork.

Before writing any R code, formulate some questions which will guide your analysis, such as: Did memory or number of CPUs tend to increase over time? Do more students prefer Macs now than they did in 2019? Write down a few guiding questions in the space below.

Formulate your guiding questions here.

These next few lines of code import the laptops data from the years 2019–2021, and combine them into a single data frame. *You will need to modify this code if you want to also include the data from 2022.*

```
laptops_2019 <- read.csv("laptops_2019.csv",comment = "#",stringsAsFactors = FALSE)
laptops_2020 <- read.csv("laptops_2020.csv",comment = "#",stringsAsFactors = FALSE)
laptops_2021 <- read.csv("laptops_2021.csv",comment = "#",stringsAsFactors = FALSE)
laptops_2019$year <- 2019
laptops_2020$year <- 2020
laptops_2021$year <- 2021
laptops <- rbind(laptops_2019,laptops_2020,laptops_2021)
```

At the end of this activity we will share and discuss these analyses using the Etherpad.

S. aegyptiacus

Programming Challenge

Instructions

Work with your team to solve the following exercise. When you have found the solution, go to the jnovembre.github.io/BSD-QBio8 and follow the link "Submit solution to challenge 1" to submit your answer.

Collaboration strategy

Before diving into the problems, first agree on a collaboration strategy with your teammates. Important aspects include communication and co-ordination practices, and setting goals and deadlines. How will your team collaborate on code, and share solutions? (Consider online resources such as Etherpad or Google Drive.) The aim is not just to complete the challenges, but also to do collaboratively; all team members should be included, and should have the opportunity to contribute and learn from each other.

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 your calculations more clear, it may be 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`.
- Record your solutions *as well as the code you used to obtain such solutions*.

Other topics

Why R?

When it comes to analyzing data, there are two competing paradigms. One could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; or 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 labmate 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.

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 automatically. 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. See www.r-pkg.org for a listing of official packages (which have been vetted by R core developers); many more are available on Bioconductor, GitHub, and other websites.

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*.

X. laevis

Data subviews

When you are working with a much larger data set you need a strategy to inspect manageable subsets ("slices") of the data.

```
laptops <- read.csv("laptops.csv",comment = "#",stringsAsFactors = FALSE)
```

Each of the examples below print a subset of the data.

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

Here are a few slightly more complex examples:

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

Once you are comfortable with the basic elements of selecting subsets, you can combine these elements in an endless variety of ways, e.g.,

```
laptops[c(1:3,5:7),c("name","os")]
```

Creating new data sets from subviews

The result of almost any calculation in R can be saved to an object. This includes selecting subsets. For example,

```
x <- laptops[1:10,]
print(x)
class(x)
```

creates a new data frame from the first 10 rows.

Conditional subviews

One powerful way to inspect subsets is by condition. For example, to view all the laptops with 4 processors, do

```
rows <- which(laptops$cpus == 4)
laptops[rows,]
```

Exercise: How would you select the subset of laptops that are running the Windows operating system?

Manipulating data

Up until now, we have treated the laptops data frame as if it were a static object. But like almost any other object in R, a data frame can be modified and overwritten. This is very powerful but also obviously dangerous! Here we will illustrate a few of the many kinds of data manipulations we can make, building on the elements we learned above. But before doing so, it is good practice to create a copy of the data frame in case we make a mistake along the way:

```
laptops2 <- laptops
```

Note that `laptops` and `laptops2` are two *entirely independent* copies of the data. Although they are identical at first, as soon you a make change to `laptop2`, you will have two different data sets. Again, when you have multiple versions of a data set present in your environment, it is your responsibility to keep your environment organized.

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

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

Using a conditional subview, we can replace all instances of "mac" values in the "os" column with "macOS":

```
rows <- which(laptops2$os == "mac")
laptops2[rows,"os"] <- "macOS"
```

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

```
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:

```
laptops2$MB <- 1000 * laptops2$GB
```

P. polytes

Building a data frame

Above, we deconstructed the laptops data frame. We can also go in the reverse direction and build a data frame by joining together data vectors. Here's an illustration.

```
x <- laptops$os
y <- laptops$cpu
```

Given text data `x` and numeric data `y`, construct a new data frame:

```
dat <- data.frame(os = x,cpu = y)
```

Creating data

R also has many facilities for creating data structures from scratch.

The most basic tool is the `c` function, short for "combine". It 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 quickly generate sequences of numbers using `seq`. For example, this generates all odd numbers from 1 to 100:

```
x <- seq(from = 1,to = 100,by = 2)
```

For simpler number sequences, use the colon operator:

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

To repeat a value (or values), 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)
```

Matrices

A matrix is like a data frame—it also has rows and columns. 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 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 two dimensions.

You can still determine the dimensions with `dim`:

```
dim(A)
```

And you can access the elements as for matrices. One thing to be careful about: 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 (perhaps without realizing it). The laptops data frame is also a list:

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

A data frame is a special kind 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 a very general and very widely used R data structure for storing complex data.

"<-" versus "="

In R, a common point of confusion in is that the equality symbol (=) can also be used for creating or overwriting objects. Many people prefer to use =, but we recommend against using it because:

1. = is easily confused with ==. (How are = and == different?)
2. The = is also used for named arguments to functions, *which is different from assignment.*

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 the value `NA`; for example, to set the first row and column to `NA` you could do `laptops[1,1] <- NA`. (You might want to make these changes to a copy of the data frame instead.) Then run `summary(laptops)`. How are the missing data reported in the summary?

M. musculus

Optional activity: Analyzing genetic data

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 over 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?

M. mulatta