Lesson 01 - Getting Started with R

Jun 13, 2016

We will be using RStudio for all of the R lessons. You should first install `R` (or update your current `R` installation), and then install `RStudio`.

Install R

Windows

1. Download R-3.3.0-win.exe, or the latest version, from this site: R for Windows.
2. Run the downloaded executable (double click and follow instructions).
3. R is now available in your Program Files folder.

Mac

1. Download R-3.3.0.pkg, or the latest version, from this site: R for Mac.
2. Run the downloaded package (double click and follow instructions).
3. R is now available in your Applications folder.

Install RStudio

1. Download RStudio Desktop Installer from this site: RStudio link.
    - Mac: RStudio 0.99.902 - Mac OS X 10.6+ (64-bit), or latest/appropriate version
    - Win: RStudio 0.99.902 - Windows Vista/7/8/10, or latest/appropriate version
2. Run the downloaded package (double click and follow instructions).
3. RStudio is now available in your Applications or Program Files folder.

Parts of this material will be very basic and obvious for some of you. Please bear with us, and make yourself available to help your neighbors who may be less familiar, as we work through this introduction together.

Exploring R

What is R?

R was developed in the 1990's by Robert Gentleman and Ross Ihaka from the Statistics Department of the University of Auckland, New Zealand, for use in statistical computing.[1]

Because it is free and open-source, it has become a convenient and powerful tool for data science and statistical modeling in many academic research fields, in government, and in industry.

Evaluating R

R was developed to allow users to engage with it **interactively** as well as for users to **develop programs** and packages that can be added to the open source repositories, for anyone to use.

The idea was that it could be flexible enough to be useful for novices, who would use existing programs/packages and functions that R provides, as well as for software developers, who would create R packages for their own custom scientific and statistical applications.

Advantages	**Disadvantages**
Open-Source	Packages are of varying quality
Academic/professional community	Very different from other standards [2]
High-quality visualization	Less intuitive plotting
Multi-dimensional/large-scale data analysis	Memory (RAM) use is not the best
Extensive documentation	Overwhelming / disparate references
There are many ways to do the same thing	There are many ways to do the same thing

> *There are only two kinds of languages: the ones people complain about and the ones nobody uses.*
>
> *— Bjarne Stroustrup (inventor of C++)*

A Word About Programming

If you think of programming essentially as part communication and part action, the following is a simplified way to think about what you are doing when you are coding.

Process:

1. Give Instructions (you, to the computer)
2. Perform Action (computer, usually hidden from you)
3. Present Output (computer, to you)

So, the fact that R is a program *and* a **programming** language means that you can customize the instructions you give to R, as well as the actions (or combinations of actions) that are available to you, enabling large-scale and repetitive/reproducible analyses that are specific to your needs.

Coding Requires Precise Instructions [3]

As with any programming language, precision in coding is critical, and beyond what we expect

in a normal conversation with other people. For example, a parent could issue an instruction to his/her child: "**Go to bed**". This seems like a complete enough set of instructions for the child to understand.

However, if the parent issued this instruction to their computer (or their robot child), the computer might ask:

- "**Who** is instructing me?"
- "**Where** should I go to bed?"
- "**When** should I go to bed?", etc.

To satisfy the robot child, perhaps "I, your parent, am asking you to **go to bed** in your room, now" is a better set of instructions.

I.e., if you are on a Linux or Mac, and you try to tell your computer to `shutdown` via the command line, it likely wants to know:

- **Who** is asking it to shutdown (the superuser, i.e. someone with authority who knows a password)
- **How** it should shutdown (halt, reboot, etc.), and
- **When** it should shutdown (usually "now").

If you are imprecise with your **instructions**, and simply type `shutdown` you get no **action** and no **output** or result.[4]

If you get the **instructions** precisely right, then the computer's **actions** will precisely follow your instructions, and the **output**, in this case, will be that the machine is powered off. [5]

We will run into this scenario with R over and over, and we hope to help you learn to troubleshoot any problems you might have communicating with R.

At this point, let's start talking about how we can use R for exploring, visualizing, and analyzing our data. Please start RStudio from your Applications/Programs folder. We will explore 4 aspects of R:

- R as a Scientific Calculator
- R as a Tool for Statistical Analysis
- R as a Tool for High Quality Plots
- R as a Tool for Reproducible Research

R is a Scientific Calculator

You can follow along by starting in the lower left-hand corner of your RStudio session. This is the **console**, where you can code *interactively*. I encourage you to type interactively, rather than copy and paste the commands.

```
> 1+1
```

The sideways carat, `>`, indicates the beginning of a line where you have entered some code, in this case an *operation*, or set of **instructions**. You should not type the `>` yourself.

After you are done typing, press enter, and below your **instructions** should be the **output**. You can ignore the `[1]` for now (it will make sense later), and just look at what follows.

```
[1] 2
```

Internally, R is doing some sort of **action**, using 0's and 1's, to produce output from your instructions.

Let's try something else:

```
> 5*3
[1] 15
```

Although there are stylistic conventions with spacing, you'll notice that R does not care if you type `5*3` or `5 * 3` – it will output the same result either way.

Many other basic operations are available:[6]

```
> 4/3
[1] 1.333333

> 10^2
[1] 100

> 10e2
[1] 1000

> log(1)
[1] 0
```

The second to last line is a sort of special case, where a letter acts as an operator. We can

ignore this for now, but keep in mind that sometimes a single letter means something to R that you might not expect.

That last line of instructions uses an operator with parentheses, which indicates that you are using a **function**. [7] Functions can often take more than one **argument** in between the parentheses. These change what input the function gets, what the function does, and/or what it communicates with you. We will talk more about these later.

Sometimes, you will get output where R has chosen to represent a numeric value using letters. This is an example of a **reserved word** in R:

```
> log(0)
[1] -Inf
```

In this case, you can do some simple operations using `Inf` yourself.

```
> Inf+Inf
[1] Inf
> Inf/0
[Inf]
> 1/Inf
0
```

Sometimes the instruction you give does not make sense to `R`.

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

See that we have received output, `[1] NaN`, as well a warning message below it describing the situation. When something goes wrong in R, it will give you a **warning**, or an **error** which is more "severe." [8] We will go over **troubleshooting** these warnings and errors later.

Regarding `NaN`, R help says "These apply to numeric values and real and imaginary parts of complex values but not to values of integer vectors". Or, more simply, `NaN = Not a Number.`

```
> Inf - Inf
[1] NaN
```

In this case, `NaN` means "undefined", but you don't get a warning message. Keep in mind that you can get errors, or unexpected/unintended output, *without* warning or error messages, so we will try to introduce you to some good coding practices such as **error checking** or **sanity checking** later on in the course.

Some basic operations are listed at this website, and you can view this video, in order to learn to use **R** to do basic calculations in your day-to-day work.

Try to use some of these on your own, providing your own input: `log2(), log10(), exp(), sqrt(), exp(), abs()`.

Did you get the results you would expect? Did you get any errors or warnings?

Now, suppose you want to define a letter or a word to represent a number or other type of **object** (such as a string of letters, a function, etc.), so that you can easily refer to the object without typing it again and again. Let's call this a **variable**.

You can **assign** the object to the letter or word using the `<-` operation, which looks like a left-facing arrow.

It works like this: **variable <- object**

We will be doing a whole lot of this, but we will introduce it very quickly here. Start by creating a variable named `myvar`, and assign a value to it:

```
> myvar <- 25
```

Now, when you type `myvar` into the console, it gives you output corresponding to the value you assigned to it:

```
> myvar
[1] 25
```

It is conventional to make assignments in this direction, assigning an object on the right to a variable on the left. It is uncommon to go the other direction in R, however, you can assign from left to right: **object -> variable**.

In R, it is also preferred to use the assignment operator instead of the more intuitive equals sign: `myvar = 25`. We will talk about when you should use the equals sign later, especially in functions, and in logic, as `==`.

Variables are convenient and critical in programming with R. Suppose you wanted to ask what the value of $25x25$ is, you could type:

```
> myvar * myvar
[1] 625
```

or

```
> myvar^2
[1] 625
```

You can then assign that set of instructions, itself, to a new variable, `myvar1`:

```
> myvar1 <- myvar^2
> myvar1
[1] 625
```

Let's try that with a **string**:

```
> mystring <- "twenty five times twenty five is six hundred twenty five"
> mystring
[1] "twenty five times twenty five is six hundred twenty five"
> mystring <- 'this is also a string'
> mystring
[1] "this is also a string"
```

By enclosing words or letters in quotes (single or double), we can assign them to the variable just like we did with the numbers and operations.

While this might not appear to be applicable to scientific calculation, you will likely do a decent amount of string manipulation in R. There are strings in your data tables ("Male", "Female", "Treated", "Control") and in your genomic sequences ("ATG CGC AAT CCT"), and we will go over how you can learn to chop them up and parse them when needed.

Finally, there are times you will want to put words in your code that you would like R to ignore or *not* evaluate, and we will talk more about those below (i.e. commenting your code).

R is a Tool for Statistical Analysis

This is a good time to introduce an R cheat sheet, here. The second page summarizes some statistical distributions.

More on this in Lecture 5.

R is a Tool for High-Quality Plots

Let's try to plot some of these distributions.

More on this in Lecture 3.

R is a Tool for Reproducible Research

You can think of **commenting** your code as similar to keeping a good laboratory notebook. You should comment your code so that you understand what the code means when you come back to it months later, and/or so that someone else can follow along with what did, if they ever try to reproduce your analysis. Comments can be useful when you are using R interactively, especially if you save your R history in a file, but you will probably comment your code much more when you are writing R **scripts**.

Getting Around in R

We will go over some of these in class, interactively:

```
ls()
rm()

search()
environment()
help()

list.files()
getwd()
setwd()

subset()
c()

q() # ctrl + D on the command line
```

Getting Help for Coding in R

```
1. Google
2. Stackoverflow
3. Reference manuals
```

Reference Texts for Learning R

- R for Data Science, by Roger Peng
- simpleR, by John Verzani
- A Beginner's Guide to R, by Alain F. Zuur, Elena N. Ieno, and Erik Meesters
- The Art of R Programming, by Norman Matloff

Data Types

1. Vectors
2. Matrices
3. Arrays
4. Lists
5. Data Frames

A good, existing, resource for learning data types in R is available at: Codeschool: Try-R, Lessons 1-6. You have to set up an account, but the initial lessons are free. There is also this introductory video by Roger Peng at Hopkins: data types video.

Other topics: * tab completion * vector addition

Packages

To find packages for R, there are several places you can look:

1. CRAN
2. GitHub
3. Bioconductor

Sometimes, websites, books, or journal articles will have details about an R package you may be interested in using. One good, peer-reviewed, open access resource is *The R Journal*.

Review

Look I will try to update this section after class to include a summary of the topics discussed and to cover the questions raised during class.

> *One of the things I really like about programming languages is that it's the perfect excuse to stick your nose into any field. So if you're interested in high energy physics and the structure of the universe, being a programmer is one of the best ways to get in there. It's probably easier than becoming a theoretical physicist.*
>
> *— Bjarne Stroustrup* [9]

Homework

We hope that you feel comfortable enough to try interactive coding with R on your own. Prior to the next class, please try to learn about data types in R at the following site, and we will start importing and exporting data in class.

Before next week, your homework is to:

1. Complete these sessions Codeschool: Try-R, Lessons 1-6. You have to set up an account, but the initial lessons are free.

Other optional resources for your learning:

1. Read and reproduce as much as you can get through section 5 of R Programming or Data Science (pages 21-31 of the pdf), which also covers data types.
2. Read this blog (for fun!): R The Master Troll

Sources

Some of this lesson is based on the online notes for this course, and from this book.

---

1. https://www.r-project.org/contributors.html ↩
2. These differ based on your research field: SASS, SPSS, Matlab, Python, GraphPad Prism, etc. ↩

3. Another take on this principle is Garbage In, Garbage out. See this site. Even so, R is unlike many other languages in that there are often many sets of instructions that can be issued to produce the same or similar output. ↩

4. (aside from any error or warning messages.) ↩

5. Sometimes, programmers will provide **defaults** in the functions that they write. This means that if the user does not give all the details (who, what, where, when, how, etc.) when sending instructions to the function, there are stored instructions that the function will use. This can be helpful to the user, simplifying the use of the function, but it is also dangerous if the user is unfamiliar with what the defaults are. ↩

6. Note that the `log()` function defaults to *natural* log, a.k.a. "ln", or `log(..., base=exp(1))`. You might mistake it for `log10`, or `log(..., base=10)`. ↩

7. We will discuss functions and what you put inside the parentheses later. Parentheses can also be used to group operations together (i.e. PEMDAS). ↩

8. And hopefully not too many **segfaults** – segmentation faults, which would cause the program to fail or **crash**. ↩

9. https://en.wikiquote.org/wiki/Bjarne_Stroustrup ↩