A Very Basic R Tutorial

Third Annual Workshop for Archaeology Graduate Students in the Northeast (WAGS: NE)

Elic M. Weitzel

4/27/2019

If you would like the original R Markdown file, find it on my GitHub page at https://github.com/weitzele/Basic-R-Tutorial

What is R?

R is a programming language designed for statistical analysis as well as a free, open-source software environment. Anyone can use R for free, and anyone can contribute to R by writing their own code and depositing it on the CRAN: the Comprehensive R Archive Network (https://cran.r-project.org/). This is the official online repository for all things R.

R was created in 1993 as an updated version of an older programming language called S. In the years since, R has become an incredibly valuable statistical tool and is growing in popularity in all scientific disciplines, archaeology and anthropology included. It has several advantages over other programs such as Excel, SPSS, PAST, and ArcGIS:

1. Working With Data

Organizing, sorting, classifying, and otherwise wrangling large amounts of data can be time-consuming and laborious work. Many archaeologists will have spent long hours highlighting, moving, and aggregating cells in Excel spreadsheets. This kind of manual data manipulation often can get the job done, but is exhausting and susceptible to the introduction of errors. Like other programming languages, R greatly facilitates working with data by providing users with a diverse toolkit for data manipulation. It also eliminates much user error in this process by utilizing code instead of manual cursor movements and button clicks. Substantially easier data manipulation may be the most important reason for archaeologists to learn R.

2. Reproducibility

Statistical software that utilizes button-clicking to conduct analyses has the benefit of often being user-friendly, but it sacrifices reproducibility. Reproducibility is a key tenet of science: if a result cannot be replicated, perhaps it cannot be trusted. To replicate a statistical analysis conducted in a software program such as Excel, SPSS, or ArcGIS, one would have to manually repeat every step. It's easy to miss things or do things slightly differently in such cases. Using R, a researcher can (and should) upload their code with any publications so that others can simply run it to replicate every step of the analysis exactly.

3. Automation

Similar to the above statements, while replicating analyses in SPSS, Excel, or ArcGIS requires manually clicking through and recreating formulas, an entire analysis can be rerun in R with a single keystroke. This has implications not only for replicating analyses, but for testing sample data or exploring new data: the

same code can be run on a completely different dataset quite easily. Many archaeologists will have analyzed data only to realize that they left out additional data or must add more later. Using R, one can simply import the new data and redo the entire analysis with a single keystroke. This represents a marked improvement over the manual button-clicking of many other software programs.

4. Complex analyses

Software such as Excel, which is commonly used by archaeologists and anthropologists, is limited to (a good number of) simple, basic statistical functions. With more complicated analyses, Excel and SPSS are less useful... Simple archaeological analyses such as correlations and t tests work fine in Excel and SPSS, but with more complex analyses (hierarchical models, generalized additive modeling, cluster analysis, Bayesian approaches, etc.), R has greater utility. Additionally, for more complex analyses, R users can write their own functions and statistical packages of functions that can be published on the CRAN and used by others.

5. Opening the black box

Many software programs with built-in functions will accept the data you give it and deliver results, but what the program is doing to generate these results can be a black box. The software may be making inappropriate assumptions about the structure of the data or automatically attribute certain values to certain variables in a function. With R, the user has the ability (and responsibility) to control all of the details of each analysis. Even with prepackaged functions, everything can be cracked open in order to be modified or just better understood. The assumptions made by other software programs have been known to cause severe problems leading to the publication of inaccurate results (https://www.bloomberg.com/news/articles/2013-04-18/faq-reinhart-rogoff-and-the-excel-error-that-changed-history) and the retraction of research articles (http://retractionwatch.com/2016/08/17/doing-the-right-thing-authors-pull-psych-review-after-finding-inaccuracies/). One particular study found that automatic formatting in Excel tables has led to 1 out of 5 biomedical research papers using Excel to have errors (https://genomebiology.biomedcentral.com/articles/10.1186/s13059-016-1044-7).

6. Cost

As already stated, R is free. Excel, SPSS, and ArcGIS all cost lots of money to purchase and license. This matters less for some people affiliated with institutions, companies, and organizations with the money to spend on software licenses, but is an important draw for independent researchers or those from smaller, less well-funded institutions.

Using R

Using R itself poses quite a few challenges... This is why a company called RStudio has developed a free and open source integrated development environment (IDE) to aid users in working with R. Using RStudio is recommended by the majority of R users.

When you open RStudio, you will see a screen with several panes, each with some buttons. At the top right will be your *Environment*. This is where RStudio stores the objects, data, and functions you create. We will discuss creating these things soon. This top right pane also contains your *History* where you can review the commands you have run.

At the bottom right of your screen will be a pane with several tabs: *Files*, *Plots*, *Packages*, and *Help* are the key ones here. *Files* is where you can find file pathways, *Plots* is where figures will be generated, *Packages* is where you can view and install packages for additional functionality (more on this later), and *Help* provides information on functions and packages.

At the bottom left of your screen, or possibly the entire left-hand side of your screen, will be your R Console. This is R itself. You can enter commands here and run them directly into R, but your code will not be saved.

To save your code, you must open an R Script (or R Markdown, but let's stick with Script) file. If there is not already one open in the top left pane, this is where it will go. Click the **File** tab at the top left of RStudio, select *New File* and then *R Script*. Now you have an R Script open. Running commands here will send them directly to the R Console but will also allow you to save your code. This is the simplest way to both run and save your code, and thus I recommend using R Scripts when writing code. The R Console is still useful for things like quick calculations that do not need to be saved, checking help files, inspecting objects/data/functions, and testing out commands.

Running code

At its most basic level, R can be used as a calculator. Copy (or rewrite) the following code into your R Script in RStudio:

```
2+2
```

If you are entering code directly into the R Console, you may simply press **Enter** to "run" this command. If you are typing into an R Script, you have several other options for running this command. You may click the button at the top right of your R Script pane that says **Run** with a green arrow to the left of it. Note that this button is also a dropdown menu with additional options for running commands. You may also use your keyboard and type **Ctrl+Enter**. Note that your cursor must be somewhere on the line of code you wish to run, or the code must be highlighted. Be aware that the hotkey for running your entire R Script, not just the selected line, is **Ctrl+Shift+Enter**. When you run the above code, your R Console pane will display the code that was run as well as the output.

Now run the following:

```
2-2

2*2

2/2

2^2
```

Note that, like most other software programs, R does not recognize **x** or adjacent parentheses as multiplication. For example, run the following lines of code:

```
(2+2)(2/2)
```

```
## Error in eval(expr, envir, enclos): attempt to apply non-function
```

```
(2+2)*(2/2)
```

The former will return an error message, while the latter is the correct syntax for multiplication.

Assigning Objects

An object holds a value, or a set of values. To create one, simply assign a value or equation as follows:

```
obj <- 22
```

The `<-` is the *assign* operator. An equals sign (`=`) also works as an assignment operator, but can create problems with more advanced use of R when it can be used for other things. The majority of R style guides recommend using `<-` as best practice, though this may be unfamiliar to those with coding experience in certain other languages. As you may have noted above, adding spaces before or after portions of your code does not impact the output, however you cannot add a space between the `<` and `-`. Doing so changes the meaning of this code from *assign* to *less than negative*.

After running the above code, the object named "obj" now holds the value 22. This object will appear in your **Environment** in RStudio. You can run the object name to see its contents:

```
obj
```

You can also assign an equation or function to an object, and the object will hold the output:

```
obj2 <- 3 * (14 / 4)
```

To change the value(s) an object holds, simply assign something else to it. Note that R will not warn you before overwriting a previously defined object.

```
obj2 <- 4 * (14 / 9)
```

When deciding what to name objects, do not use spaces or hyphens. Periods, underscores, and all lowercase and capital letters are fair game. Numbers can be used as long as the object name does not begin with a number. There are many stylistic suggestions for how to name objects that we won't get into here, but the key thing to avoid is object names that are the same as function names. We'll talk about functions in a minute.

Different Types of Objects

R has specific terms that refer to various types of objects. These are:

- vectors
- lists
- arrays
- matrices
- tables
- data frames

For basic uses of R, vectors and data frames are perhaps most commonly used, with lists being a close third.

A vector is one of the most important object types. It contains a list of either numbers (a *numeric* vector), letters (a *character* vector), or `TRUE` and/or `FALSE` values (a *logical* vector). The following is a numeric vector:

```
vec.1 <- c(1, 2, 3, 4, 5, 6)
vec.1
```

`c()` is the *concatenate* function, which tells R to combine everything in the parentheses. When specifying a series of values, you will often need to use the concatenate function: forgetting to do so is a common reason for error messages.

A character vector is a list of letters or words, as follows:

```
vec.2 <- c("One", "Two", "Three", "Four", "Five", "Six")
vec.2
```

A logical vector is a list of `TRUE` or `FALSE` values. We won't get into this type of vector here, but it looks like this:

```
vec.3 <- c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)
```

To refer to a specific element within a specific vector, you can employ *indexing*:

```
vec.1[2] #returns the second element in the vector
vec.2[4] #returns the fourth element in the vector
```

Note that adding a `#` symbol to your code signals to R that whatever follows on that line should not be run. This is called *commenting* and is a very important part of coding. You should begin to use comments to annotate your code so that you can understand what is happening on a given line and so that others who may look at your code can as well.

In addition to `c()`, R has many, many other built-in functions. Try some of these out:

```
mean(vec.1)

median(vec.1)

range(vec.1)

sqrt(vec.1)

sd(vec.1)

log(vec.1)

log(vec.2)
```

```
## Error in log(vec.2): non-numeric argument to mathematical function
```

If you run that last command (`log(vec.2)`), you should receive an error message that says `non-numeric argument to mathematical function`. Oftentimes, error messages in R are... less than helpful... But in this case, we have some sense of what's going on here: `vec.2` contains at least one non-numeric value. Indeed, if you inspect `vec.2`, you'll see that it is a character vector.

We can inspect objects using the `str()` function (i.e. *structure*). It is often important to know the types of data included in an object (e.g. numeric, character, etc.). This can be done using the `str()` function as follows:

```
str(vec.1)
```

```
##  num [1:6] 1 2 3 4 5 6
```

```
str(vec.2)
```

```
##  chr [1:6] "One" "Two" "Three" "Four" "Five" "Six"
```

The output of `str()` informs us that the object `vec.1` is a numeric vector with six observations and that the object `vec.2` is a character vector with six observations. This is why the command `log(vec.2)` didn't work: the object doesn't actually contain numbers that are recognized as such by R. `str()` can be used on any type of object, not just vectors, and is in fact much more useful for more complicated objects.

A *data frame* is a combined group of vectors which are all of the same length. You can create a *data frame* using the `data.frame()` function:

```
df <- data.frame(vec.1, vec.2)
df
```

To refer to a specific vector within a data frame, use the `$`:

```
df$vec.1
df$vec.2
```

To refer to a specific element within the data frame, use indexing via brackets. Within brackets, the structure is *[row, column]*:

```
df[3, 1] #the element in the third row of the first column

df$vec.2[5] #the fifth element of the specified vector in the data frame
```

You can also name the columns within a data frame like this:

```
df <- data.frame("Vector_1"=vec.1, "Vector_2"=vec.2)
df

df$Vector_1
```

Packages

R has many built-in functions like we've seen above. These built-in functions are part of what is called *base R*. However, there are many other functions and datasets that are stored in the CRAN repository in the form of *packages*. There are two ways to install a package.

1. Option 1 is go click the **Packages** tab in the lower right pane in RStudio. Next, click the **Install** button and type the name of the package you wish to download in the dialog box that appears. This box shows that RStudio is installing the package from CRAN. Click *Install*.
2. Option 2 is to use code. Run the code below to install the package `rcarbon`:

```
install.packages("rcarbon")
```

It is best practice to only type the above code into the R Console, not to save it as part of your R Script. It is considered rude to install packages on another person's computer, so you should not include the `install.packages()` function in your shared code.

The `rcarbon` package is now installed and you can use the datasets and functions it contains. However, before doing so, you need to tell R to call up that package with the following code:

```
library(rcarbon)
```

This code is needed because, while R has installed the package, it is not loaded into your current session until you do so yourself. You will therefore need to run this `library()` function every time you close and reopen R intending to use the specified package. You only need to install the package once, though.

As with any package, you can go online and pull up the complete details of the package on the CRAN site: https://cran.r-project.org/web/packages/rcarbon/rcarbon.pdf.

Now inspect one of the built-in datasets in the `rcarbon` package. We can do this using `?` followed by the dataset:

```
?emedyd
```

Inspect the `calibrate()` function in the `rcarbon` package the same way:

```
?calibrate
```

Note that the help page includes a description of what the function does, the arguments in the function (the things you need to plug in or the things that it assumes), descriptions of these arguments, and assorted additional information about the function including sources to cite when using it. Perhaps most importantly, it includes examples at the bottom of how to operationalize the function.

The Working Directory

Every object we've created thus far has been assigned data that we generated within R (or pulled from an R package). Once you start pulling in data (e.g. Excel spreadsheets) from outside of R instead of defining everything here, you'll need to set the working directory.

A working directory is a folder on your computer where you store the files you will be using in R. Setting a working directory allows you to keep your files organized and easily accessible and lets R know where to find the relevant files. You can set any folder on your computer to be the working directory via one of the following options:

1. Option 1: at the top of the screen in RStudio, click "Session" then "Set Working Directory." If you select "To Source File Location", the directory will be set to the folder in which this R Script file is currently saved. If you select "Choose Directory..." you can select any folder. Once you select a folder, you will see a version of the following output in your R Console (the specific file pathway is unique to your computer):

setwd("~/UConn/R_Tutorial")

2. Option 2: Alternatively, you can set the working directory by passing the command `setwd()`. In the parenthesis after the function, type the file pathway in quotation marks. More simply, you could set the working directory via Option 1, and then copy the code from the R Console into your R Script. If you have this code in your R Script, you can simply run it every time you reopen the file, instead of clicking a series of buttons. However, it is best practice to remove this working directory from your R Script if you intend to share this Script with others.

Working with .csv files

Given the widespread use of Excel to create and maintain spreadsheets, it is useful to know how to incorporate Excel files into R. You can both import and export .csv (comma separated values) files, as well as .txt files and many other file types from other software programs as well. Here, we'll focus on .csv files.

To create a .csv file, let's make a table containing two columns of data. To do this, let's use the built-in data from the `rcarbon` package we installed and loaded above.

The `calibrate()` function allows you to calibrate radiocarbon dates according to a specified calibration curve:

```
?calibrate
```

Note the main arguments used in the `calibrate()` function. The functions we've used before didn't require arguments in the same way that many other functions, like `calibrate()`, do. Now, we're going to use `calibrate()` to calibrate the radiocarbon dates from one of the sites that is included in the `emedyd` dataset that comes loaded with `rcarbon`. Just run the following code and for now, don't worry about how I'm obtaining the radiocarbon dates from just the one site we're interested in.

```
jericho <- emedyd[emedyd$SiteName == "Jericho", ]

jdates <- calibrate(jericho$CRA, jericho$Error)
```

In the help file, note that the IntCal13 calibration curve is set as the default in the `calibrate()` function, and so it does not need to be included as an argument. If you wanted to replicate dates calibrated using an older curve, or if you wanteed to use a marine curve, you could do so by specifying that in the `calCurves` argument in this function.

The result of this `calibrate()` function (the `jdates` object) is a complex object that you don't need to worry about (if you're curious, you can see how complex using the `str()` function mentioned previously). Perhaps what we are interested in is just the median calibrated dates for this site, so we'll use another function in the `rcarbon` package called `medCal()` to get these values:

```
jdates_med <- medCal(jdates)
```

Now we can take these median calibrated dates and pair them with their original lab numbers to create a data frame:

```
jericho_dates <- data.frame("LabNo"=jericho$LabID, "Median"=jdates_med)
```

We can then export this data frame as a .csv file that can be read by Excel or many other programs. The `write.csv()` function will export a .csv file of the specified data frame or vector and give it the specified name:

```
write.csv(jericho_dates, "Jericho_Dates.csv")
```

This .csv file will be saved in the working directory that we specified earlier.

If we already had a .csv file that we created in Excel or obtained elsewhere and we wanted to import this file into R to work with, we could use the `read.csv()` function. Here, we can assign the .csv file to the object `jericho_median_dates`:

```
jericho_median_dates <- read.csv("Jericho_Dates.csv")
```

Now that `jericho_median_dates` is an object in our environment, we can easily work with the data it contains.

A final note on troubleshooting

When working in R, as with all other programming languages and software packages, you will encounter many problems. Nobody can know everything about R, not even those who have helped to develop the language. Searching the internet for answers is therefore a very legitimate way to find the solution you're looking for. When searching for information about R programming, Stack Exchange (https://stackexchange.com/) is your best friend. The programming website under the larger Stack Exchange umbrella is Stack Overflow (https://stackoverflow.com/), which is where you will find many answers to both common and rare problems with R. Often times, internet searches will lead you to pages associated with these websites. Odds are, if you have a question about how to do something in R, someone has already had that same question, has asked it online, and someone else has answered it. In the rare cases where this is not true, you can post your question on Stack Overflow and someone will help you out (but be sure to read their guidelines for posting questions!).