# Some Basics of Using R

ANTH 3720-001 Archaeological and Forensic Science Lab Methods: Data Analysis with R

Elic Weitzel

Jan. 29-31, 2021

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

## 1 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:

### 1.1 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.

### 1.2 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.

### 1.3 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.

### 1.4 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.

### 1.5 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), the retraction of research articles (http://retractionwatch.com/2016/08/17/doing-the-right-thing-authors-pull-psych-review-after-finding-inaccuracies/), and the renaming of dozens of genes to accomodate Excel's quirks (https://www.theverge.com/2020/8/6/21355674/human-genes-rename-microsoft-excel-misreading-dates). 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). The numerous negative implications of this fact are disturbing to say the least...

### 1.6 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.

## 2 Some Basics of 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, though other options do exist.

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 (.R) file. You can also open an R Markdown file (.Rmd), which is what this document is, but let's stick with scripts for now. 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.

## 2.1 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.

## 2.2 Creating 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.

## 2.3 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. We will therefore focus on vectors and data frames for the time being.

### 2.3.1 Vectors

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 (sometimes called the *combine* 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)
```

Vectors must be comprised of only one type of data (numeric, character, or logical). Therefore, if you were to try and create a vector with a combination of different data types, you would run into problems... For example:

```
vec.fail <- c(TRUE, 1, "archaeology", FALSE, FALSE, 17)
```

Creating this vector, which combines logical, numeric, and character values together, will not give you an error message but it will result in a problem...

```
str(vec.fail)
```

```
##  chr [1:6] "TRUE" "1" "archaeology" "FALSE" "FALSE" "17"
```

We can inspect the **structure** of this vector object using the `str()` function, as above. This is an incredibly useful function for understanding objects with which you're working. When we apply this function to the vec.fail object, we see that R is understanding this vector as a character vector, indicated by `chr [1:6]` in the function output. Because vectors must be all the same data type, R is treating the logical and numeric values in this vector as characters even though only one of the six values we concatenated is a character.

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.

### 2.3.2 Data Frames

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 (i.e. column) within a data frame, use `$` as follows:

```
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
```

## 2.4 Functions

Functions are used to manipulate objects in some specified way. Instead of hand-writing a potentially complex mathematical process every time you need to use it, one can assign that process to a function. This simplifies your code and expedites things.

### 2.4.1 Built-In Functions

In addition to `c()` and `data.frame()`, which you used above, 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 somewhat 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 and R is unable to use that non-numeric argument in a mathematical function.

As we did above, let's inspect `vec.1` and `vec.2` using the `str()` function to make sense of why we're encountering this problem.

```
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, thus the logarithm can't be calculated. `str()` can be used on any type of object, not just vectors, and is in fact much more useful for more complicated objects.

### 2.4.2 Help Documentation

Perhaps the most important skill needed to use functions in R is knowing how to understand the function's help page. To access the documentation for a function, use `?` as below:

```
?log
```

When you run `?log` RStudio will pull up this function's documentation in the **Help** tab in the bottom right quadrant of the screen. Here you can read a verbal *Description* of the function and what it does.

This page will also show a section on *Usage* which illustrates how to implement the function and the arguments it requires. In the case of `log()`, you need to specify `x`. You can also specify an argument called `base` which is referring to the logarithm's base: whether *e* or *10*. The default logarithm base in R is *e*, which is specified in this `log()` function as `base = exp(1)`. To compute a base-10 logarithm, you could change this `base` argument in the `log()` function, or use the related function `log10()` which is also shown in this *Usage* section of the Help document.

Scrolling down, you will then see an *Arguments* section that explains in greater detail the arguments included in this function which were shown in context in the *Usage* section. This section is useful for reading a bit more about what each argument is doing. For example, if you weren't sure what `x` is supposed to be, you can see here that it is "a numeric or complex vector", meaning that you can provide R with a single number or a series of numbers in vector form, such as:

```
log(17)
```

```
## [1] 2.833213
```

```
log(vec.1)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595
```

You can see that running `log(17)` results in a single value in the output, while running `log(vec.1)` results in six values: one for each of the elements in this vector.

Next is the *Details* section which provides even greater depth of information about the function and its arguments.

Several other sections may or may not exist for a given function's help page, but you will see a *References* section which provides the citations relevant to this function, which are often what you can cite if needed.

Importantly, the last section of a function's help documentation will usually include *Examples*. These are real, functional examples of how to use the function which you can copy into your R Console or R Script and run. This *Examples* section is particularly useful if the rest of the documentation is a bit unclear and you need to see how exactly the function is used and what its output looks like.

### 2.4.3 Writing Functions

Writing your own functions is possible in R. This can be quite complicated depending on what your goals are, but it can also be very simple for simple operations.

Simply as an example, let's create a function called `do_math` that takes a number or a vector and runs a specific mathematical operation on it. To create this function, we first need to give it a name. I picked `do_math` because it often helps to have function names which are verbs since functions do things to objects, which are like nouns in the place of objects of sentences. You can name your function anything you'd like, but it should be short and relevant to the operation being performed (calling this function `cat_in_the_hat` might be cute, but I have no idea what such a function is doing based on that name). Function names should also be unique: I wouldn't want to call this function `log` or `c` because functions by those names already exist and therefore things could get very confusing.

Next, we assign our specified function to our function name using `<-`. There are two key parts to what comes next. First, we need `function(x)` which tells R that we are creating a function which will be run on `x`. The user must supply `x` to the function `do_math`. Second, in brackets, you must specify what this function is doing with the object `x`. In this case, we're multiplying `x` by 7, then dividing by 23.6. Run the following code to create this `do_math` function.

```
do_math <- function(x) {
  (7 * x) / 23.6
  }
```

It may seem like nothing has happened, but if you look in your Environment in the top right of RStudio, you will see that this `do_math` function has been added to your list of Functions. Now you can use this function, as below:

```
do_math(9.6)
```

```
## [1] 2.847458
```

```
do_math(vec.1)
```

```
## [1] 0.2966102 0.5932203 0.8898305 1.1864407 1.4830508 1.7796610
```

Here, I first ran this function on the number 9.6, and R spits out the answer to this mathematical operation. I then ran this function on `vec.1`, which results in six output values: one for each of the elements in `vec.1` as this function applies the specified mathematical formula across all six of the values in this vector.

There is so much more involved in writing functions, but this very simple introduction will suffice for now.

## 2.5 Packages

R has many built-in functions like we've seen above. These built-in functions are part of what is called *base R*, which refers to the set of capabilities available from simply loading R. However, there are many other functions and datasets that are stored in the CRAN repository in the form of *packages*, which can be used to supplement the capabilities of base R. 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")
```

Once you have installed the `rcarbon` package, it exists on your computer and can be called up by R. Therefore, you don't need to install this package again if you close R and open it again later: the package still exists. If you delete R from your computer or update to a new version, you will need to reinstall the package, however.

It is best practice to only type the above code into the R Console, not to save it as part of your R Script. This is because R Scripts are often shared between people or included as part of a publication. If someone else downloads your R code and runs it, and you have included the `install.packages()` function in your Script, you will cause this person to install the specified software on their computer. While installing an R package is rather benign, it is generally rude and invasive to install software 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)
```

```
## Warning: package 'rcarbon' was built under R version 4.0.3
```

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 stated above.

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, let's inspect one of the built-in datasets in the `rcarbon` package. We can do this using `?` followed by the dataset name:

```
?emedyd
```

I knew this dataset existed in this package because I ran `?rcarbon` and clicked on the link provided in the *Note* section to the package authors GitHub page, where they curate up-to-date versions of the package and provide all of the data and functions (https://github.com/ahb108/rcarbon).

We can inspect the `calibrate()` function in the `rcarbon` package the same way:

```
?calibrate
```

Note that this function contains more arguments than the one we explored above. You must provide this function an object `x` and a list of `errrors` at minimum, which the *Arguments* section explains are the uncalibrated radiocarbon dates and their errors (standard deviations). You can also specify a variety of other arguments. You'll see, under the *Usage* section, that these other arguments have default values specified. For example, the default calibration curve used by the `calCurves` argument is "intcal20" (the IntCal20 curve), but if you don't want to use this default you can change it to one of the other options described in the *Arguments* section of the help page.

## 2.6 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 or .csv files) from outside of R instead of defining everything here, you'll need to set what is called 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:

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")*

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. This is because the file pathway to where you keep your files won't be the same as theirs since you're working on different computers.

### 2.6.1 Working with .csv files

Given the (rather unfortunately...) widespread use of Excel to create and maintain spreadsheets and databases among archaeologists and anthropologists, it is useful to know how to incorporate Excel files into R. You can both import and export a variety of file types into R, but we'll try and make things a bit more generally applicable by using .csv (comma separated values) files. These are commonly created by saving your Excel spreadsheet as a .csv file instead of an .xls or .xlsx file.

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

As we saw previously, the `calibrate()` function allows you to calibrate radiocarbon dates according to a specified calibration curve:

```
?calibrate
```

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 pre-loaded with `rcarbon`. Let's say that we want to calibrate the dates from site of Jericho. The dates from this site are included in the `emedyd` dataset, but so are many other sites that we're not interested in right now. To inspect this dataset, we can use the `View()` function:

```
View(emedyd)
```

This function will open up a new tab in the top left of your RStudio screen showing the data you specified. We can see that there are 10 columns and 1,915 rows. Manually sorting through the sites in this dataset and pulling out only the dates for Jericho could take quite some time. But fortunately, R makes things easy!

We can use indexing (the use of brackets after an object) as we did earlier to extract the relevant rows from this huge dataset. However, before we used indexing to extract values from known rows and columns by providing the number of each, like `obj[1,2]` which would return the value in the first row from the second column of the `obj` object. But we don't know the numbers of the rows which contain the Jericho dates in this dataset... So what we can do is get creative with our indexing and use logical operations.

First, let's figure out what columns exist in this `emedyd` dataset. We can see these columns visually by using `View()` but we can more directly obtain this information by using the `colnames()` function:

```
colnames(emedyd)
```

```
##  [1] "LabID"     "CRA"       "Error"     "Material"  "Species"   "SiteName"
##  [7] "Country"   "Longitude" "Latitude"  "Region"
```

Now we know the names of the ten columns of `emedyd`. The one we want is `SiteName` as this contains the name of each site in the dataset. But we only want the rows of this dataset which relate to the "Jericho" site name... So let's inspect the `SiteName` column to see what our options are. To do this, let's use the `unique()` function, which returns all the unique values of the specified vector.

```
unique(emedyd$SiteName)
```

This function will output a long list of all 196 site names in the `SiteName` column. They're listed alphabetically, and Jericho is the 74th one. This function has now informed us on what the Jericho site is called in this data frame, which can be important information if the data frame is using abbreviations or has typos.

To extract the Jericho dates, let's create an object called `jericho` to which we will assign only the rows of the `emedyd` data frame which have "Jericho" in the `SiteName` column. Let's assign `emedyd` to `jericho`, but we're going to index `emedyd` using brackets so that we're not just assigning the entire data frame to a new object name. The code below will index the `emedyd` data frame and assign all rows for which `SiteName` is equal to `"Jericho"`, along with all columns, to the new object `jericho`.

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

Note that within the brackets, `emedyd$SiteName == "Jericho"` is to the left of the comma, and nothing is to the right of the comma. Remember that indexing is structured as `data.frame[rows, columns]`. This is because `emedyd$SiteName == "Jericho"` is referring only to the rows of `emedyd`. We only want the rows of `emedyd` for which "Jericho" is the site name, but we want *all* of the columns of `emedyd` to be kept. Leaving nothing to the right of the comma tells R that we want all columns, as we do not specify any.

Now that we've created a new object called `jericho` which contains only the rows of `emedyd` relevant to the site of Jericho, we can calibrate these dates using the `calibrate()` function specifying two arguments - `x`

and `errors` - as noted above. Because we know from looking at the help page that `x` is the first argument of the function and `errors` is the second, we could type `x =` and `errors =`, or we could simply provide the relevant vectors in their proper order and R will understand that they correspond to `x` and `errors`.

```
jericho.dates <- calibrate(jericho$CRA, jericho$Error)
```

In our `jericho` data frame, just as in the `emedyd` data frame, `CRA` is the column name of the uncalibrated dates and `Error` is the column name for the standard deviations. Thus, running the function above produces calibrated radiocarbon dates and assigns them to the object we specified, `jericho.dates`. This function has a nice feature which tracks the progress of the function as it works and lets you know when it's done.

In the help file, note that the IntCal20 calibration curve is set as the default in the `calibrate()` function, and so it does not need to be included as an argument.

The result of this `calibrate()` function (the `jericho.dates` object) is a complex object that you don't need to worry about (if you're curious, you can see how complex it is using the `str()` function mentioned previously). Let's pretend that 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:

```
jericho.dates.med <- medCal(jericho.dates)
```

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

```
jericho.dates <- data.frame("LabNo"=jericho$LabID, "Median"=jericho.dates.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.

Instead of creating a .csv file in R, let's say 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 to do so. Here, we can load this .csv back into R (the one that we just created in our working directory) and assign it 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.

# 3 A 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 first!).