Lab 1- Introduction to R

Open up R or R Studio, because we are going to learn how to use it today! Don’t be nervous if you’ve never done any computer coding. There’s no magic involved, just getting used to doing something new.

**1.) Calculation and assignment**

The > symbol should be visible in your console. Type in 1+1 and hit enter. You should see

> 1+1

[1] 2

You can also do calculations on variables. You can make variables using the “assignment” function “=” or “<-”. The “=” is faster to type, whereas the “<-” represents an arrow showing that the number on the right is being stored as the variable on the left. You can use either one, as they have the same effect. Try this:

> x=5

> y<-6

Nothing comes up as a result when you assign a variable. If you want the value stored in the variable back, you need to call the variable.

> x

[1] 5

> y

[1] 6

If you assign a variable you have already used, the value is stored is replaced by the new value. Try this:

> x=x+5

> x

[1] 10

Variables don’t have to be individual letters. They can be words, and can include numbers, however don’t start a variable with a number. R is case sensitive, so remember if you have used lower- or upper-case letters. Do not include special characters like $, #, &, etc in a variable name. Experiment with these commands:

> coolnumber=3

> coolnumber\*x

[1] 30

> 4a=1

Error: unexpected symbol in “1r”

> Coolnumber

Error: object ‘Coolnumber’ not found

An error message pops up when you enter something that R doesn’t understand. It takes a while to get used to error messages, and get a sense of what they are telling you. Eventually, recognizing error messages will give you a good hint of what is wrong in your code. If you see this error, either you are calling the wrong name, or there was an earlier problem with creating the object that you are now calling. For now, don’t worry too much about the meaning, but do read error messages when they appear.

R will follow the order of operations when doing calculations. However, there is no “implicit multiplication” in R.

> 2x+coolnumber

Error: unexpected symbol in 2x

> 2\*x+coolnumber

[1] 23

R also uses parentheses to indicate the order that operations should occur. Be careful with parentheses, and try to eliminate unnecessary parentheses as they can become hard to read.

> (2\*x)+coolnumber

[1] 23

> 2\*(x+coolnumber)

[1] 26

**2.) Built-in R functions**

R has a lot of functions that are built into the program, and installed when you install R. The functions can do a lot of different things, from basic calculations, to statistical tests, to producing graphics. Many of the more basic functions will look familiar as they are used as abbreviations in math in general. In R, functions are called by entering their name, and then certain inputs, or “arguments” are placed in parentheses to be passed into the functions. The functions then return the output. Try these simple expressions:

> sin(pi/2)

[1] 1

> exp(0)

[1] 1

> sqrt(9)

[1] 3

> log(exp(18))

[1] 18

Some functions take several arguments at once. Try this simple one:

> sum(1,2)

[1] 3

> sum(1,2,3)

[1] 6

Some functions can have a graphical output:

> plot(1,2)



> plot(2,1)

****

The outputs of plot(1,2) and plot(2,1) are different from each other. Many functions can take in multiple different arguments, and the order of those arguments matters, like the plot function. Functions can also have internal variables that are set to a default value unless changed by the user. We will see examples of more complex functions and how to use them effectively in the following sections, as well as how to create new functions that are not included in the base version of R, and how download more functions that have been created by others.

**3.) Classes and objects**

Besides numeric variables, R can deal with many other types of objects. For instance, words can be stored in R as a “character” type object. This works the same as variables with numeric values. You can always see what type of object a variable represents by using the class() function. Try this:

> class(5)

[1] “numeric”

> x=10

> class(x)

[1] “numeric”

> y= “hello”

> y

[1] “hello”

> class(y)

[1] “character”

Character objects always appear with quotes around them. Otherwise, R will interpret words you type as variable names. For instance:

> y=hello

Error: object ‘hello’ not found

In some cases, you can change the class of an object. This is often necessary when inputting data.

> x=as.character(x)

> x

[1] “10”

> class(x)

[1] “character”

> x=as.factor(x)

> x

[1] 10

Levels: 10

Whenever you input data into R, check what class each variable is. Sometimes, we want data to be numeric, but it gets interpreted as factor data. This can cause all sorts of problems with analysis. The class of data can often be changed using as.[class], however be careful when changing the class of data, as it can cause unexpected results.

> as.numeric(x)

[1] 1

How did that happen? Factors have different “levels” that are a finite set of different values a factor can be. They can be words or numbers. The different levels are then numbered automatically with counting numbers, 1, 2, 3, etc. In this case, R thinks that x is a factor, with one possible level: 10. Since there is only 1 level, the level is labeled 1, which is then returned by as.numeric(). To avoid this problem, always change a factor that should be numeric to a character first. Example:

> as.numeric(as.character(x))

[1] 10

Remember that parentheses are evaluated from the inside out!

**4.) Vectors, Matrices, and Indexing**

Every variable in R is actually a vector, by default. A vector is a sequence of numbers, characters, or other objects that is stored in an ordered fashion. Vectors are indexed starting at 1. So far, whenever you’ve typed anything into R that gives some return value, you have received a [1] followed by the return value. The [1] in this case indicates the index of the value being returned, which is 1 by default. Let’s see how we can create a longer vector.

A simple way to start is by using the c() function. The “c” stands for “concatenate” which means to add on to. Try this:

> x=c(1,2,10)

> x

[1] 1 2 10

If we want a vector with a sequence of steadily increasing numbers, we can use the sequence function. The sequence function takes in the starting number in the sequence, the ending number in the sequence, and the spacing between numbers in the sequence.

> seq(1,20,1)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

[20] 20

> seq(1,20,0.5)

[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0

[12] 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0 10.5 11.0 11.5

[23] 12.0 12.5 13.0 13.5 14.0 14.5 15.0 15.5 16.0 16.5 17.0

[34] 17.5 18.0 18.5 19.0 19.5 20.0

When the vector overflows the width of the output window, R goes on to the next row, and indicates the first index number on the new line in square brackets. So we know that 6.5 is the 12th number in the sequence from 1 to 20 by increments of 0.5. Based on your window size, the output may take more or fewer lines.

Let’s say that we want to get a value from our vector back. Say we want the 26th number in our sequence from 1 to 20 by increments of 0.5. We can do this easily in R using indexing.

> x= seq(1,20,0.5)

> x[26]

[1] 13.5

Alternatively, say we want the index number of the value in our sequence that equals 15. This uses the function which().

> which(x==15)

[1] 29

The == sign is a logical symbol which we will review more in the section on logic.

Matrices are similar to vectors, but they have 2 index values. To make a matrix, you need to specify the data to enter in a vector, the dimensions of the matrix, and how to fill up the matrix with the data.

> matrix(c(1,2,3,4),ncol=2,nrow=2)

[,1] [,2]

[1,] 1 3

[2,] 2 4

> A=matrix(c(1,2,3,4),ncol=2,nrow=2,byrow=T)

> A

[,1] [,2]

[1,] 1 2

[2,] 3 4

Matrices fill by column by default, and you need to specify if they should be filled by row instead. To get out values, specify both indices. In R, the first index number denotes rows, and the second index denotes columns.

> A[2,2]

[1] 4

> A[1,2]

[1] 2

If you want to recover a certain row or column from a matrix, rather than a single value, simply leave blank the index that you are not specifying, while including commas. So to get the first row, leave the column index blank:

> A[1,]

[1] 1 3

Unfortunately, R usually doesn’t automatically return both indices if you use which(), but instead treats the matrix like a vector. If you want to recover both indices, you can change the arr.ind variable inside your which() command.

> which(A==4)

[1] 4

> which(A==4,arr.ind=T)

row col

[1,] 2 2

**5.) Scripts, Style, Data, and File Management**

When you have code that you would like to save for reference or to continue work on later, you should save that code as an R script. R script files have the .R extension. In R Studio, create a new script using the File>New File> R Script menu option. The **** button in the top left corner will also open a new script. In later labs in this course, R scripts will be provided. Pay close attention to their format. A good script has a descriptive title, organized sections, descriptions of what the sections do, and comments next to the code. In R, use the # symbol to denote comments. Whenever R sees a #, it will ignore everything typed on the remainder of the line. Short comments should go after individual lines, and longer comments should go before and/or after blocks of code. Pay attention to spacing when writing code, and add more space to increase clarity. These strategies are important both for understanding your own old work and for sharing work with others.

When opening data files in R, you can always proceed by entering the whole address of the file. That is, all the folders that the file is in plus the file name. However, if you are both opening several data files, and exporting several files like code and graphics, it is often easier to use R’s working directory function. Your working directory is a folder where you will keep all your files for a particular project. It is strongly recommended that you use a new folder for each project, or at least a new folder for this course. In the future, you may want to reference your project files and code, and may have trouble finding it if you leave it in the Documents or Downloads folder. Using folders also helps determine which files you want to copy over when you get a new computer.

In R Studio, you can set the working directory by using the Session>Set Working Directory>Choose Directory menu option. Then just navigate to the folder that you want to use and select it. If you do this, the code for setting the working directory will be automatically generated in the console. You should paste this code into your script, so that your script is ready to run when you open it again. If you are using base R, you may type out the directory itself, or follow these rules (for Mojave). In the Finder, use View>Show Path Bar. Then navigate to the folder you wish to use for your working directory, right click on the folder in the path bar, select Copy “[Folder]” as Pathname. Then use the setwd() function in R to set the working directory. Make sure to use quotes around the path:

> setwd(“/Users/yourname/Documents/foldername”)

If you are using a lot of files, such as population data, community data, phylogenetic data, drafts of figures, etc, it is likely that you will have many versions. Try to name these in a systematic way so you will always know which file is the most current.

**6.) Logic, Loops, and Conditions**

Logical statements, loops, and conditional statements are some of the most important tools in programming. Many programs are based on these foundational tools, so it is important to get familiar with how to use these tools in R.

Logical is a class of objects that can be either TRUE or FALSE. There are various logical operators that can combine logical objects, such as “and”, “or”, “not”, and “is equal”. The “==” sign means “is equal to”, and will return a logical object. You can express “is greater than” with > and “is less than” with <.

> x=7

> x==7

[1] TRUE

> x==8

[1] FALSE

The ! means “not”, which simply reverses TRUE and FALSE.

> ! TRUE

[1] FALSE

The & means “and”, and the | means “or”. The | can be typed by using SHIFT + BACKSLASH. See these examples:

> x=7

> x>8

[1] FALSE

> x==7

[1] TRUE

> x>8 | x==7

[1] TRUE

> x>8 & x==7

[1] FALSE

Table 1: A table of common logical symbols and their meanings.

|  |  |  |
| --- | --- | --- |
| Logical symbol | Meaning | Notes |
| == | Is equal to | Be careful not to confuse with the normal “=” |
| > | Is greater than |  |
| < | Is less than |  |
| >= | Is greater than or equal to |  |
| <= | Is less than or equal to |  |
| | | Or | Both must be FALSE for an or statement to be FALSE, otherwise the statement is TRUE. |
| & | And | Both must be TRUE for an or statement to be TRUE, otherwise the statement is FALSE. |

Conditionals, or “if” statements, ask the computer to do something if a certain value is TRUE. For instance, I can simulate a game where a fair coin is flipped, using an if statement, and the sample() function. In this game, if you win, you gain 20% of your money, and if you lose, you lose 20% of your money.

> money=10

> win=sample(c(T,F),1) #chooses either TRUE or FALSE 1 time

> win

[1] TRUE

> if(win) {

money=money\*1.2

} else {

money=money\*0.8

}

> money

[1] 12

Loops are a way to make computers do a lot of repetitive tasks quickly. There are two kinds of loops which are commonly used: “for loops” and “while loops”.

For loops ask the computer to do a task a set number of times. Say I wanted to simulate the game with the fair coin from above, but I want to do it for 10 trials, starting with 10 dollars.

> money<-10

> for(i in 1:10) {

win=sample(c(T,F),1)

if(win) {money=money\*1.2} else {money=money\*0.8}

}

> money

[1] 12.23059

Looks like I won money this time! What if I wanted to play this game until I lost half of the money, to see how long that would take. This is the perfect time to use a while loop.

> money=10

> numPlays=0 # I’m going to use this counter to count how many times I’m playing

> while(money>5) {

win=sample(c(T,F),1)

if(win) {money=money\*1.2} else {money=money\*0.8}

numPlays=numPlays+1

}

> numPlays

[1] 84

Wow, that was a lot of trials! Loops are very useful in any programming language, but in R, there are times when it might be better to avoid loops. This is because vectorization may be faster. Vectorization tends to work well in place of for loops. Let’s say again that I want to simulate 10 trials of the coin flipping game. Here’s a way to do that with vectorization

> money=10

> wins=sample(c(1.2,0.8),10,replace=T) # sample with replacement- this matters if you are taking multiple samples

> prod(wins)\*money

[1] 8.153727

**7.) Graphing**

Earlier, we saw that some R functions can output graphs. Later in the course, we will have an entire lab on graphical techniques. This section will go over just the very basics of graphing with a toy data set. Open the data set mice.csv and examine the data set. You can do this using the “Import Dataset” button on the Environment tab in R Studio, or directly from the R Console.

> mice<-read.csv(“mice.csv”,header=T)

> View(mice)

A table should open that shows you that data in the mice file. Let’s explore the data:

> hist(mice$weight)



A histogram counts the frequency of mouse weight in each of R’s pre-defined bins.

> plot(mice$weight,mice$length)



We see a generally positive relationship between mouse length and mouse weight. Let’s now look at average mouse weight by location:

> barplot(tapply(mice$weight,mice$location,mean)) # tapply will tabulate one variable by another variable



Nice! Clearly the mice weigh the most in the kitchen… Say we want to add a title and have our axes labeled:

> barplot(tapply(mice$weight,mice$location,mean),main="Mouse weight by room",ylab="Mean weight (g)",xlab="Room in my house")



The function tapply() is a powerful way to avoid using loops in R, and we will see further use of it later in the course. These are the very basics of graphing. In future labs, you will learn how to add colors, make multi-panel figures, add keys, and customize any figure that you might want to create in R.

**8.) Packages**

There are a lot of packages available for R that are available for R from the CRAN repository. It is easy to add these packages onto R to get additional functionality. The packages are made by small developers, including many researchers who have a specialty in some area that they want to share. Some of our work will use various R packages. When downloading a package, you can use R Studio’s built-in installer, by navigating to the “Packages” tab from the window where plots appear. Click the install button, and type in the package name. Make sure “Install dependencies” is checked. You can also install packages directly in the console. Use the form install.packages(“package”,dependencies=T). For example:

> install.packages(“vegan”,dependencies=T)

We will use the community ecology functions in the vegan package later in the course.

**9.) User defined functions**

Sometimes, it is useful to make new functions that can perform tasks not available in base R. This can help streamline your code and save you time if there is something that you need to do repeatedly. To start with, let’s work with a simple and familiar function, the quadratic formula. Recall from your algebra class that the quadratic formula gives solutions to quadratic equations. The formula is given by:

To define a function in R, we first decide on a name for the function, then define the function’s inputs, the function’s body, and the function’s output.

> QuadForm<-function(a,b,c, ShowIm=F) {

d<-b^2-4\*a\*c

x1<-(-b+sqrt(d))/(2\*a)

x2<-(-b-sqrt(d))/(2\*a)

return(c(x1,x2))

}

Notice the placement of brackets around the body of the function. The style and spacing may look odd, but it is consistent with coding style guidelines. It is recommended that the body of some statement in brackets should be placed in a block starting on a new line after the opening bracket, and the closing bracket be placed on its own line so it is visually clear that the brackets are closed. Let’s try doing the quadratic formula:

> QuadForm(-2,3,10)

[1] -1.608495 3.108495

> QuadForm(2,3,10)

Error in return(x1, x2) : multi-argument returns are not permitted

In addition: Warning message:

In sqrt(b^2 - 4 \* a \* c) : NaNs produced

Why is there an error message? The value in the square root must be negative, and therefore not have real roots. It’s good to know that there are no real solutions, but we may also want the solutions, real or not. Let’s edit our quadratic formula to deal with this issue so it can return imaginary solutions as well- we’ll do this using R’s complex number class. However, let’s add a variable that the function can take in that will tell it whether to return imaginary roots or not.

> QuadForm<-function(a,b,c, ShowIm=F) {

d<-b^2-4\*a\*c

if(ShowIm&d<0) {

x1<-(-b+sqrt(as.complex(d)))/(2\*a)

x2<-(-b-sqrt(as.complex(d)))/(2\*a)

} else {

x1<-(-b+sqrt(d))/(2\*a)

x2<-(-b-sqrt(d))/(2\*a)

}

return(c(x1,x2))

}

We should be ready now. We just have to make sure to turn the internal variable “ShowIm” to True if we want the imaginary solutions too.

> QuadForm(2,3,10,ShowIm=T)

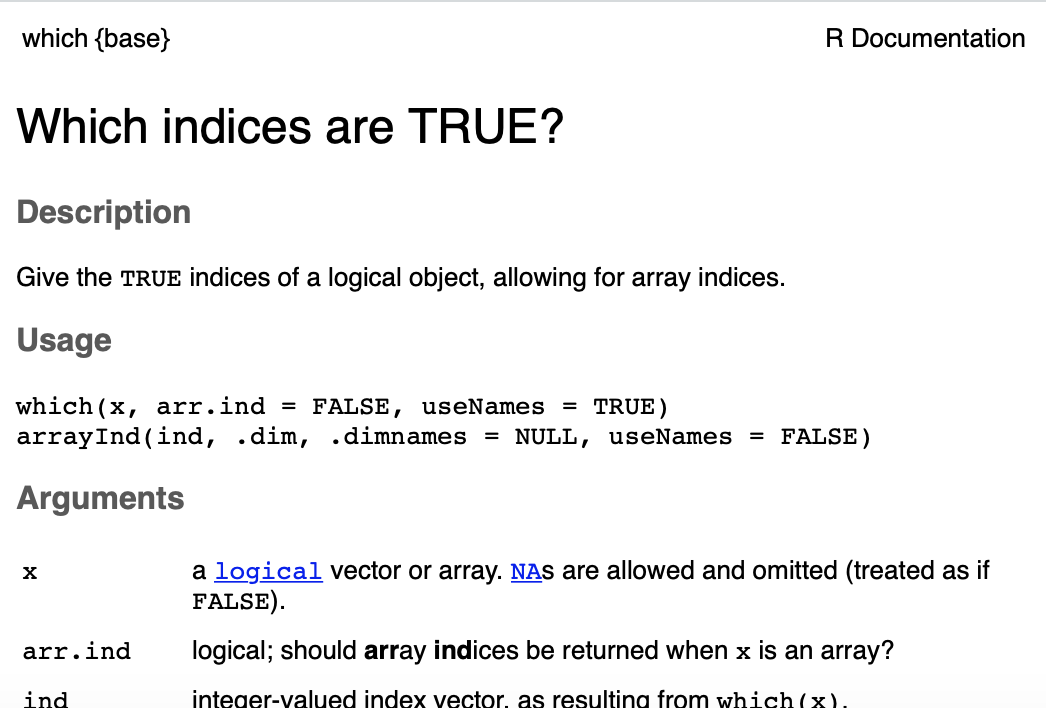
[1] -0.75+2.106537i -0.75-2.106537i

Great! Now we can see imaginary roots! It’s possible to build more complex functions, of course, but keep in mind that it is best to start simple, and then troubleshoot, before adding more complexities to the function.

**10.) Help**

R has a built-in help system that can tell you how to use functions in base R and in R packages that you download. To activate the help system, simply use the help() function. When activated, a window appears with the help topic.

> help(which)



The help file describes what inputs, or arguments a function takes (in this case one input x is required), and any default internal function variables (in this case arr.ind and useNames have default variables that can be changed). The help file also has a description of the function and examples of its usage. These can be very helpful, but also intimidating for users who aren’t familiar with the jargon. Don’t worry, the vocabulary will become more familiar over time.