Week 1 - Intro to R

C.A. Kiahtipes

8/15/2023

Week 1: Introduction to R

We will use the R statistical computing environment as a learning tool for this course, with the goal that you will find coding skills and quantitative reasoning to be valuable intellectual assets in the future. Thus, we will find ourselves developing our skills using computational tools as well as our understanding of how quantitative reasoning is applied within paleoecology.

Working in R: Troubleshooting Resources

There is a large network of R users from different backgrounds who produce and maintain web-based resources for this platform. There are a number of cheatsheets that summarize important sets of commands and provide examples. Don't be afraid to save these and use them as a quick reference later.

https://iqss.github.io/dss-workshops/R/Rintro/base-r-cheat-sheet.pdf

Working in R: File Management

In order to access files on your computer, R needs to know where to begin its search. This is your "working directory". Weekly labs are available as .zip files or github repositories that can be checked out. Make sure that R thinks that it is located inside of this folder. Keep in mind that you may successfully open a script or markdown document, but not be able to run it because R does not know where to look for the files which support these scripts.

```r
getwd()
```

```
## [1] "/Users/christopherkiahtipes/Dropbox/Temporary/Quant_Paleo_Fall23/Quant_Prob_Paleo"
```

Working in R: Using the console.

The R statistical computing environment operates with only a console window. This text window is where R gives you feedback and where you enter commands. The R Studio wrapper makes working in R easier by combining the console with other windows showing you important information or documents. You can use R or R Studio to follow along. You can copy-paste the code directly into the console, re-type the commands, or create an R script with your notes and code that can be run anytime.

Working in R: Annotations and coding principles.

One reason that so many scientists choose to use command-line coding languages like R for their analyses is because it creates additional transparency for their research projects. Although there is a long way to go in the publishing world, the proliferation of open-access databases, data sets, and techniques for analyzing them promises to improve the reproducibility of our work and shore up the integrity of sceintific inquiry. The problems with the integrity of the academy and scholarly research at large extends beyond the realm of computer code. But for my part I do believe that this approach *does* improve the accessibility of our work and it makes it easier for us to produce consistent and well-documented analytical results. A bonus with R is that it gives us an immense capacity to communicate our results via plotting.

We improve the quality of our work and its reproducibility in the lab by maintaining lab notebooks, which also allows us to diagnose problems and establish effective routines. For quantitative analysis, we take the same approach by maintaining a text file with our code and annotations explaining what the code is supposed to do. The benefit of this approach is making your analyses more consistent and giving you the chance to make systematic improvements to your work through trial and error.

In R, we make annotations using the hash symbol "#".

```
#Using the hash symbol lets us enter non-coding text into any document.
```

Let's look at an R script which is a basic text document that you can use to track your R code. This is basically a .txt file, but R and R Studio know how to read these with some additional details that make it easier to code.

IMG OF SCRIPT

How to use an R script for lab notes and code improvement.

Working in R: Simple Grammar, Complex Outcomes

Objects One of the reasons that R is appealing to the research community is that it uses a simple and discrete structure to execute complex commands. R only recognizes three different kinds of input: object calls, function calls, and operators (=, <-, etc.). Objects character strings to which we assign some data. When we do this, we tell R that whenever it sees that character string, it should return the contents of the object. We define objects by assigning information to a set of characters. To start, we'll use the "<-" (arrow) operator to assign the value 5 to "x" and the value 6 to "y". In the code chunk below, we define two objects, "x" and "y", giving them the values 5 and 6 respectively. Then, we call the objects by putting their name in the console and hitting .

```
x <- 5
y <- 6

x
```

```
## [1] 5
```

```
y
```

```
## [1] 6
```

This can also be achieved by using the equals sign ("="). These symbols are differentiated to make it easier for users to develop custom functions, where it can make sense to differentiate between object and variable assignments. To many R users, this distinction means nothing and their work is not impacted by ignoring "<-".

```
x = 5
y = 6

x
```

```
## [1] 5
```

```
y
```

```
## [1] 6
```

Objects range from simple single digit assignments or character strings to tables, lists, or arrays of data. We will walk before we run, so let's build up the dimensionality of our object. We can make ranges of numbers with ":". Note this can move in either direction.

```
x <- 1:100
y <- 100:1
```

```
x
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

```
y
```

```
##   [1] 100  99  98  97  96  95  94  93  92  91  90  89  88  87  86  85  84  83
##  [19]  82  81  80  79  78  77  76  75  74  73  72  71  70  69  68  67  66  65
##  [37]  64  63  62  61  60  59  58  57  56  55  54  53  52  51  50  49  48  47
##  [55]  46  45  44  43  42  41  40  39  38  37  36  35  34  33  32  31  30  29
##  [73]  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11
##  [91]  10   9   8   7   6   5   4   3   2   1
```

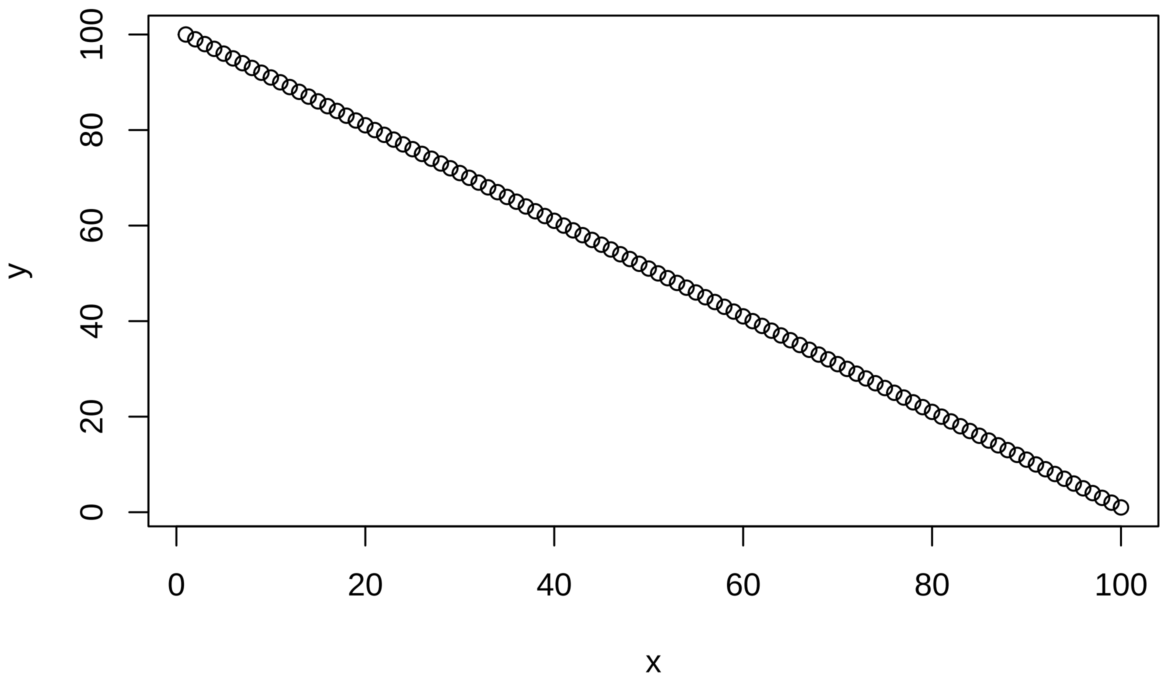
This is a good time to introduce one of R's most important and powerful functions, plot(). All R functions are designed to take some object and perform a task, so they are called by a combination of a character string "plot" and parentheses where objects are called (). On top of this, R's functions also have a number of arguments built into them that allow you to manipulate how the function operates. What is nice about functions is that all arguments have default settings and they are not required in most cases. We can verify this by looking at the help entry for plot with the following.

```
help(plot)
```

```
## Help on topic 'plot' was found in the following packages:
## 
##   Package               Library
##   graphics              /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/library
##   base                  /Library/Frameworks/R.framework/Resources/library
## 
## 
## Using the first match ...
```

Plot is really powerful and we will use it a lot. One advantage to simple plotting is that you can quickly visualize your data, examine it for errors, and look for important patterns. Below, we call the plot function and feed it our objects. R does a lot here, scaling the plot window automatically to the ranges and applying the object names as the labels for the axes. We will work to expand our plotting capacity alongside our quantitative skills in this course and the two will nicely compliment each other. Visualizing our expectations, problems, and data is a key tool for reinforcing and deepening our knowledge.

```
plot(x,y)
```



We can also create objects from non-sequential numbers. In order to do so, we will use an essential function within R: concatenate. The function call is "c()", where we take multiple data entries that are delivered in the order listed. In the example below, we assign ten numbers to "x".

```
x = c(9,1,7,3,2,8,4,5,6,10)

x
```

```
## [1]  9  1  7  3  2  8  4  5  6 10
```

We have been using the object names "x" and "y" here, but we can create any object name we wish so long as we follow some rules. The object name cannot start with numbers or special characters. The object name cannot include spaces or special characters aside from underscores ("_").

There are some basic principles for object naming. We should always give the objects obvious, yet descriptive names. Some of this is learned by experience, but in general I do not use names longer than 10 characters.

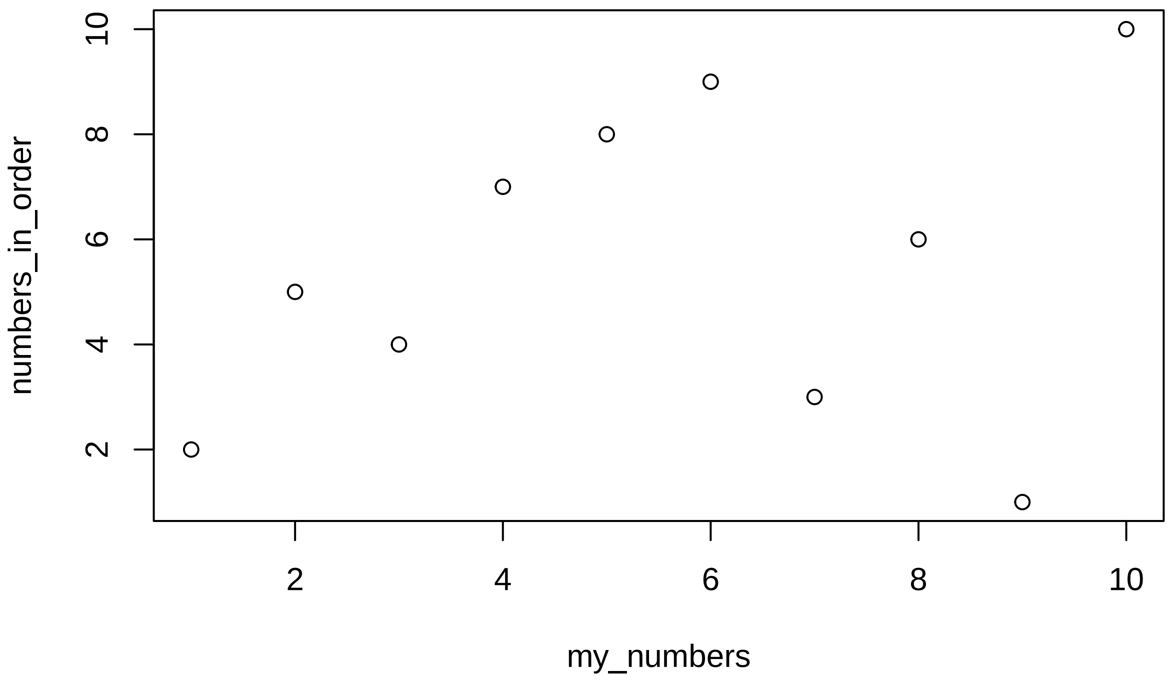
```
my_numbers = c(9,1,7,3,2,8,4,5,6,10)

numbers_in_order = 1:10 #This is a generally bad object name, with too many characters.

my_numbers
```

```
## [1]  9  1  7  3  2  8  4  5  6 10
```

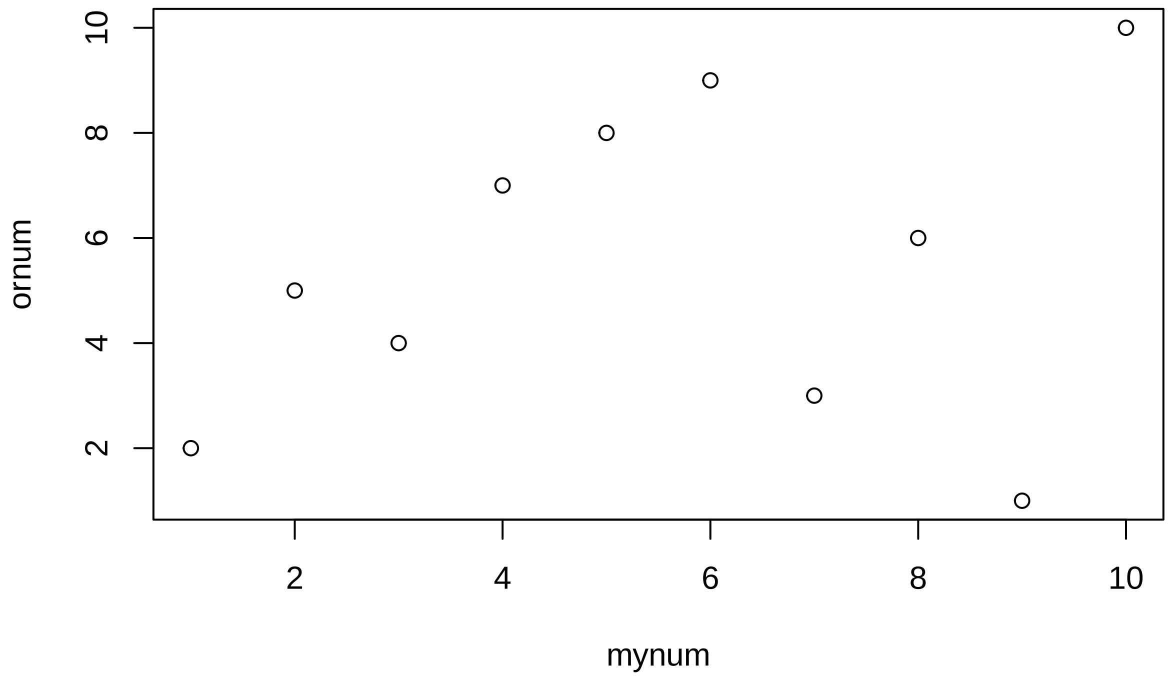
```
numbers_in_order
```

```
## [1]  1  2  3  4  5  6  7  8  9 10
```

```
plot(my_numbers,numbers_in_order)
```



```
mynum = c(9,1,7,3,2,8,4,5,6,10)
ornum = 1:10

plot(mynum, ornum)
```



We see here that objects are more than the data we've stored in them. They include the order of the data as well as the object name. This information shapes the output that we get from functions like plot() and the ways that we can manipulate our data. Below, we will explore the different kinds of information that can be encoded into vector objects.

Working in R: Vector Objects and Object Classes

The objects we have been creating thus far are all *vectors*. Vectors are a single dimension of data with a length greater than or equal to one. So, a single number ("x = 8" or "y = A") or a string of numbers ("x = 1:5)" or "y = c('A', 'E', 'C', 'F', 'B')") are both considered vectors. We can interrogate this with "is.vector()".

```
x = 8

y = 'A'

is.vector(x)
```

```
## [1] TRUE
```

```
is.vector(y)
```

```
## [1] TRUE
```

```
x = 1:5

y = c('A','C','F','G','B')

is.vector(x)
```

```
## [1] TRUE
```

```
is.vector(y)
```

```
## [1] TRUE
```

Above, we have introduced a *character* vector alongside the *integer* vectors we had defined previously. Using either double (" ") or single quotes (' '), we can encode data that is read as character strings rather than numbers. We can interrogate the classes of objects using the class() function. Understanding these base R functions (default operations that come with R) can help with data handling and manipulation later.

```
class(x)
```

```
## [1] "integer"
```

```
class(y)
```

```
## [1] "character"
```

Notice here that these last two code chunks have used two kinds of functions to query these objects and they return different classes of data to us. R objects are filled with data and R recognizes three kinds of data input: numeric, character, and logical. Vector objects assume that all of the data is the same type. If we attempt to make a vector with multiple data types, all types are converted to characters.

```
myint <- 1:10
mychr <- c('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J')
mylgi <- c(TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, FALSE, FALSE)

class(myint)
```

```
## [1] "integer"
```

```
class(mychr)
```

```
## [1] "character"
```

```
class(mylgi)
```

```
## [1] "logical"
```

```
#Below is an example of a vector that breaks the rules and will be forced into character format.

mymix <- c(1, "A", TRUE)

#Check the class of "mymix".
```

We can extract individual entries from a vector by using square brackets after the object call ([]). We can retrieve individual entries or segments of an object by their position within the object.

```
myint[1]
```

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

```
myint[3:8]
```

```
## [1] 3 4 5 6 7 8
```

```
mychr[c(7, 1, 2)] #We can use concatenate to extract a non-consecutive set of values from this object.
```

```
## [1] "G" "A" "B"
```

Objects can interact with each other and we can use one object to modify the other. As we go about subsetting data or assessing how our categorical assignments match up with reality, we will regularly modify objects with other objects. By encoding our categorizations into objects, they can be applied uniformly to our analyses as well as our plots. Below, we modify a vector of character data with a vector of logical data. When we're looking for specific values within an object, we use two equals signs (==). In the code below, we tell R to give us the values of z where the values of t are equal to "TRUE". In normal human language, we are using t to tell R which values of z

```
z = c("This", "is", "not", "a", "character", "vector")
t = c(TRUE, TRUE, FALSE, TRUE, TRUE, TRUE)

#These can interact and we can subset the one by the other by using [].

z[t == TRUE]
```

```
## [1] "This"      "is"        "a"         "character" "vector"
```

We can also subset data using various operators applied to the original objects. R uses a number of typical mathematical operators: +, -, /, *, >, <, >=, <=, and ==. Other operators worth knowing are and ("&") as well as or ("|"). This could use better coding examples.

```
z[myint[3:8] >= 6]
```

```
## [1] "a"         "character" "vector"
```

```
z[t == TRUE & myint[3:8] == 8]
```

```
## [1] "vector"
```

```
#We can even define new objects using these data subsets

zint = z[myint[3:8] >= 6]

zint
```

```
## [1] "a"         "character" "vector"
```

Base R includes a function call to create a vector. The arguments within it allow you to specify the mode of the data (numeric, character, or logical) and the length of the vector.

```
x <- vector(mode = "numeric", length = 100)
```

We can fill this vector with data using brackets to navigate the positions within the vector.

```
x[1:10] = 1:10 #Fill positions 1 - 10 with the values 1:10

x[c(11, 15, 17, 23, 29)] = c(111, 115, 117, 123, 129) #Fill positions 11, 15, 17, 23, 29 with 111, 115,

x[x == 0] = 88 #We're filling every value of 0 with the value 88. Here we're putting one value in multi

x
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10 111  88  88  88 115  88 117  88
##  [19]  88  88  88  88 123  88  88  88  88  88 129  88  88  88  88  88  88  88
##  [37]  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88
##  [55]  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88
##  [73]  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88  88
##  [91]  88  88  88  88  88  88  88  88  88  88
```

We can find out something about how R works here. Let's revise the data in our object.

```
x[1:10] = 1:10 #We're putting 10 integers into the first ten positions

x[11:100] = 11:20 #We're putting 10 integers into the other 90 positions.

x
```

```
##   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 11 12 13 14 15
##  [26] 16 17 18 19 20 11 12 13 14 15 16 17 18 19 20 11 12 13 14 15 16 17 18 19 20
##  [51] 11 12 13 14 15 16 17 18 19 20 11 12 13 14 15 16 17 18 19 20 11 12 13 14 15
##  [76] 16 17 18 19 20 11 12 13 14 15 16 17 18 19 20 11 12 13 14 15 16 17 18 19 20
```

In the case where the number of positions available is divisible by the length of the data being added, R will repeat the data to fill the object.

Working in R: Using Lists to Organize Vectors

We can attach vectors to each other into multi-dimensional objects with list(). Instead of juggling many objects in your code, you can create lists of associated objects that can be called under the same name. We will revisit our objects from earlier (myint, mychr, mylgi) and make them into a list.

```
my_list = list(myint, mychr, mylgi)
```

Lists are flexible and can hold vectors of different lengths also.

```
my_list = list(myint, mychr, mylgi, c("another", "character"), c(1,3,5))
```

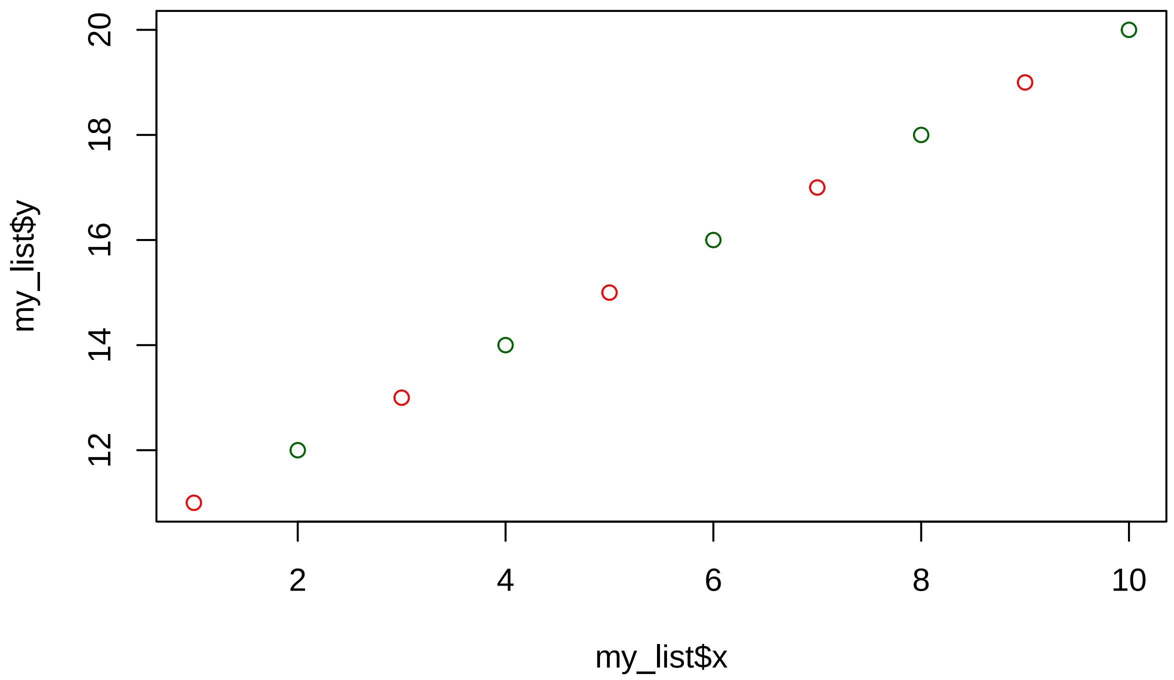
Let's make a more useful list to demonstrate some more of its useful features. We also can see some useful features of R Studio at this point, which highlights color names as we type them in.

```
my_list = list(x = 1:10, y = 11:20, col = c("red","darkgreen"))

my_list
```

```
## $x
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $y
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
##
## $col
## [1] "red"       "darkgreen"
```

```
plot(my_list, col = my_list$col) #For some reason you do not need to call x and y individually.
```



Lists help us deliver multiple vectors of information at a time, which will be critical for building arrays, which is their equivalent for matrices. There are also arguments within several R functions which expect list objects.

Working in R: Matrix Objects and Data Frames

Matrices are tables of data, which by definition has two dimensions. This adds one more layer of complexity to our vector objects and it is worth discussing how we create and interact with these objects. In most research settings, we primarily interact with matrices. Let's use the base function matrix() to create a two-dimensional data set.

```
my_matrix = matrix(data = c(1:10, 11:20, 111:120)) #We are using the arguments "data = " and "ncol =".

is.matrix(my_matrix) #We can interrogate the object and see if it is a matrix.
```

```
## [1] TRUE
```

Now, at this point we've made a matrix object and determined that it is indeed a matrix. Let's call the object and see our matrix looks like.

```
my_matrix
```

```
##       [,1]
##  [1,]    1
##  [2,]    2
##  [3,]    3
##  [4,]    4
##  [5,]    5
##  [6,]    6
##  [7,]    7
##  [8,]    8
##  [9,]    9
## [10,]   10
## [11,]   11
## [12,]   12
## [13,]   13
## [14,]   14
## [15,]   15
## [16,]   16
## [17,]   17
## [18,]   18
## [19,]   19
## [20,]   20
## [21,]  111
## [22,]  112
## [23,]  113
## [24,]  114
## [25,]  115
## [26,]  116
## [27,]  117
## [28,]  118
## [29,]  119
## [30,]  120
```

Because we used matrix() and fed it 30 integers, but *did not* define the number of rows or columns, R made a matrix with one column. Let's look at the arguments available in matrix and then make a more sensible matrix.

```
help(matrix)
```

The help page shows us that there are five arguments we can give to matrix(), including the number of rows and columns. This will let us make a more useful matrix. We will also introduce objects to give names to our rows and columns.

```
my_cols = c("Species 1", "Species 2", "Species 3")
my_rows = c("Site 1", "Site 2", "Site 3", "Site 4", "Site 5", "Site 6", "Site 7", "Site 8", "Site 9", "

my_names = list(my_rows, my_cols) #We're making a list here with the two vector objects above.

my_matrix = matrix(data = c(1:10, 11:20, 111:120), ncol = 3, dimnames = my_names) #Note we don't have t
```

Just like vectors, we can make empty matrices that we can populate with data later. All we need to do is specify the number of rows and columns.

We interact with matrices in the same way we do vectors, with one minor difference. We use the object's name plus the bracket ([]) symbol, but we need to specify the row and column position, which is done by separating them with a comma ([row, column]). We can see this in action below.

```
my_matrix[,1] = 1:10 #We're taking the columns in order here
my_matrix[,2] = 11:20
my_matrix[,3] = 111:120

#An alternative method

my_matrix[,"Species 1"] = 1:10
my_matrix[,"Species 2"] = 11:20
my_matrix[,"Species 3"] = 111:120
```

The brackets also allow us to call specific sections of the data within any matrix.

```
my_matrix[, 1]
```

```
##  Site 1  Site 2  Site 3  Site 4  Site 5  Site 6  Site 7  Site 8  Site 9 Site 10
##       1       2       3       4       5       6       7       8       9      10
```

```
my_matrix[1, ]
```

```
## Species 1 Species 2 Species 3
##         1        11       111
```

```
my_matrix[2, 2:3]
```

```
## Species 2 Species 3
##        12       112
```

```
my_matrix[c(2,4,8), 3] #When can choose non-consecutive rows here with concatenate.
```

```
## Site 2 Site 4 Site 8
##    112    114    118
```

TEXT IN DEVELOPMENT

Much of what we end up doing in R is defining objects and then using functions to execute a set of commands against those objects.

FAIR Principles

I need to include this, where is the best space to introduce?