Week 2 - Importing and Working with Data

Nicholas Kortessis

2025-01-22

A quick review of R

You should have all completed the tutorials in R using the "swirl" package. Here is a quick refresher of some things you should have learned that are important for going forward.

R is composed of objects, which can come in many forms. The simplest is a single item. Most commonly, we want an object to be a collection of items. These collections come in a number of forms, but we will focus on four

- Vectors
- Arrays
- Lists
- Data Frames

Vectors

The most common object you will encounter in R is a vector. Vectors are simply a set of items enumerated out. Square brackets are used for isolating specific items in a vector. Here are some examples

```
x <- 3*4
x
```

```
## [1] 12
```

```
is.vector(x)
```

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

```
length(x)
```

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

```
x[2] <- 100
x
```

```
## [1]  12 100
```

```
x[5] <- 3
x
```

```
## [1]  12 100  NA  NA   3
```

R is built to work with vectors. Many operations are vectorized, i.e., by default they will happen component-wise when given a vector as an input. Notice that R also recycles vectors if you give R a length that is not the full length of the vector. This feature can sometimes be helpful, but is more often than not a source of problems. Here is an example.

```
(y <- 1:3)
```

```
## [1] 1 2 3
```

```
(z <- 3:7)
```

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

```
z^2
```

```
## [1]  9 16 25 36 49
```

```
y+z
```

```
## Warning in y + z: longer object length is not a multiple of shorter object
## length
```

```
## [1] 4 6 8 7 9
```

Checkpoint 1: What just happened? R gave us a warning. Do you understand why? In a new script, write out the specific equations that gave rise to the output of y+z in this case.

Let's try and resolve this error.

```
(z <- 3:8)
```

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

```
y+z
```

```
## [1]  4  6  8  7  9 11
```

Ah, that's better. No warning.

Checkpoint 2: Why is there no error in this case? What computations are being done in this case? Write you answer in your script as a comment.

Plain vanilla R objects are called "atomic vectors" and an absolute requirement is that all the bits of info they hold are of the same sort, i.e. all numeric or logical or character. If that's not already true upon creation, the elements will be coerced to the same "flavor", using a "lowest common denominator" approach (usually character). The concatenate function c() is often used for making vectors (or lists, more later).

```
(x <- c("cabbage", pi, TRUE, 4.3))
```

```
## [1] "cabbage"          "3.14159265358979" "TRUE"             "4.3"
```

```
length(x)
```

```
## [1] 4
```

```
mode(x)
```

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

```
class(x)
```

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

Vector "Flavors"

Every R object has a type, a mode, and a class (among other things!) and it can be bewildering to navigate these different facets of an R object. For now, assume we are talking about atomic vectors, such as vectors of numbers, character strings, or logical values. Even a single number is stored as a vector of length 1, so atomic vectors are really the most basic sort of R object we need to understand. This table presents a technically

correct typology of the most common atomic vectors, along with a simplified-yet-useful adjunct which we'll call "flavor".

"flavor"	type reported by `typeof()`	`mode()`	`class()`
character	character	character	character
logical	logical	logical	logical
numeric	integer or double	numeric	integer or double
factor	integer	numeric	factor

Thinking about objects according to their "flavor" above will work fairly well for most purposes most of the time, at least when you're first getting started. Notice that most rows in the table are the same across columns, meaning there isn't any difference between "flavor", "type", "mode", and "class" of an object. One important exception is **factors**. **Factors can sometimes be a pain when manipulating data frames and plotting.**

Let's see some examples.

```
n <- 8
set.seed(1) # This ensures we all have the same values
(w <- round(rnorm(n), 2))
```

```
## [1] -0.63  0.18 -0.84  1.60  0.33 -0.82  0.49  0.74
```

```
typeof(w)
```

```
## [1] "double"
```

```
mode(w)
```

```
## [1] "numeric"
```

```
class(w)
```

```
## [1] "numeric"
```

```
(x <- 1:n)
```

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

Checkpoint 3: In your script, find the `typeof()`, `mode()`, and `class()` for `x` and list them.

Let's get the third item in x.

```
x[3]
```

```
## [1] 3
```

Now let's do something different.

```
x[-3]
```

```
## [1] 1 2 4 5 6 7 8
```

Checkpoint 4: What did this do? What would x[-8] look like?

Let's make some character-based objects.

```
(y <- LETTERS[1:n])
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H"
```

Checkpoint 5: What is the `typeof()`, `mode()`, and `class()` of `y`?

Let's make a logical-based object.

```
(z <- runif(n) > 0.3)
```

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

Checkpoint 6: What is the `typeof()`, `mode()`, and `class()` of `z`?

Last, let's make a factor-based object.

```
(v <- factor(rep(LETTERS[9:12], each = 2)))
```

```
## [1] I I J J K K L L
## Levels: I J K L
```

```
v[3]
```

```
## [1] J
## Levels: I J K L
```

Use `str()` and any other functions you wish to inspect these objects. `str()` tells you the type of object, how many elements it has, and gives the first few elements of the object.

```
str(w)
```

```
##  num [1:8] -0.63 0.18 -0.84 1.6 0.33 -0.82 0.49 0.74
```

```
str(x)
```

```
##  int [1:8] 1 2 3 4 5 6 7 8
```

```
str(y)
```

```
##  chr [1:8] "A" "B" "C" "D" "E" "F" "G" "H"
```

```
str(z)
```

```
##  logi [1:8] TRUE TRUE TRUE TRUE TRUE FALSE ...
```

```
str(v)
```

```
##  Factor w/ 4 levels "I","J","K","L": 1 1 2 2 3 3 4 4
```

Checkpoint 7: Use the function `which()` to return the values (not the indexes) of the `w` object that is greater than 0. Include the code that performs this task in your script.

Arrays (Matrices)

A matrix is like a 2-dimensional vector. All the elements are of the same "flavor", but they are arranged into multiple rows that are 'stacked' together (alternatively, multiple columns pasted together). For many graphics (e.g., heatmaps, contour plots, 3D scatterplots), your data need to be formatted in the form of a matrix. Arrays are generalizations of matrices to more than two dimensions. Matrices are then 2-d arrays. But you could easily put multiple matrices together side-by-side to have a 3-d array. Think of matrices as rectangles and think of a rectangular solids as 3-d arrays. It's hard to visualize 4-d objects, but they nonetheless exist and are straightforward to produce in R. But we are unlikely to have any use for them.

Let's make a simple matrix. A nice feature of R is that you can name the columns and rows, which helps in interpreting the numbers inside.

```
(m1 <- matrix(letters[1:25], nrow = 5, ncol = 5))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "a"  "f"  "k"  "p"  "u"
## [2,] "b"  "g"  "l"  "q"  "v"
## [3,] "c"  "h"  "m"  "r"  "w"
```

```
## [4,] "d"  "i"  "n"  "s"  "x"
## [5,] "e"  "j"  "o"  "t"  "y"
```

This makes a square matrix with 5 rows and 5 columns of the first 25 letters in the alphabet.

Notice how the letters are ordered by column. We could do the same thing but order by row if we wanted.

```
(m1 <- matrix(letters[1:25], nrow = 5, ncol = 5, byrow = TRUE))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "a"  "b"  "c"  "d"  "e"
## [2,] "f"  "g"  "h"  "i"  "j"
## [3,] "k"  "l"  "m"  "n"  "o"
## [4,] "p"  "q"  "r"  "s"  "t"
## [5,] "u"  "v"  "w"  "x"  "y"
```

Sometimes it is helpful to check how many rows and columns matrices have.

```
dim(m1)
```

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

Rows are always indicated first. If we want to extract an element from this matrix, we need to indicate both a row and a column.

```
m1[2,3] #read: go to second row and third column of m1.
```

```
## [1] "h"
```

```
m1[1,1]
```

```
## [1] "a"
```

```
m1[2,1]
```

```
## [1] "f"
```

```
m1[1,2]
```

```
## [1] "b"
```

We could also pull out vectors of elements. For example, let's pull out the first row. We do this my just not specifying the column we want. R interprets this as giving us all the columns.

```
m1[1,]
```

```
## [1] "a" "b" "c" "d" "e"
```

Or we could ask for the first row and all columns except the third!

```
m1[1,-3]
```

```
## [1] "a" "b" "d" "e"
```

And we can do the same extracting columns as vectors.

```
m1[,1]
```

```
## [1] "a" "f" "k" "p" "u"
```

```
m1[-3,1]
```

```
## [1] "a" "f" "p" "u"
```

We can also ask for multiple rows (or columns).

```
m1[2:3, 3:5]
```

```
##      [,1] [,2] [,3]
## [1,] "h"  "i"  "j"
## [2,] "m"  "n"  "o"
```

This code asked for the 2nd and 3rd rows and the 3rd, 4th, and 5th column elements of those rows.

Checkpoint 8: What is one line of code to extract all but the last two rows and columns of m1? Write this in your script.

```
(m2 <- matrix(letters[1:25], nrow = 20, ncol = 5))
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,] "a"  "u"  "p"  "k"  "f"
##  [2,] "b"  "v"  "q"  "l"  "g"
##  [3,] "c"  "w"  "r"  "m"  "h"
##  [4,] "d"  "x"  "s"  "n"  "i"
##  [5,] "e"  "y"  "t"  "o"  "j"
##  [6,] "f"  "a"  "u"  "p"  "k"
##  [7,] "g"  "b"  "v"  "q"  "l"
##  [8,] "h"  "c"  "w"  "r"  "m"
##  [9,] "i"  "d"  "x"  "s"  "n"
## [10,] "j"  "e"  "y"  "t"  "o"
## [11,] "k"  "f"  "a"  "u"  "p"
## [12,] "l"  "g"  "b"  "v"  "q"
## [13,] "m"  "h"  "c"  "w"  "r"
## [14,] "n"  "i"  "d"  "x"  "s"
## [15,] "o"  "j"  "e"  "y"  "t"
## [16,] "p"  "k"  "f"  "a"  "u"
## [17,] "q"  "l"  "g"  "b"  "v"
## [18,] "r"  "m"  "h"  "c"  "w"
## [19,] "s"  "n"  "i"  "d"  "x"
## [20,] "t"  "o"  "j"  "e"  "y"
```

```
dim(m2)
```

```
## [1] 20  5
```

This worked by going through all 25 letters and then repeating them again 4 times. That's 25 x 4 = 100 letters, which is the same as 20 rows x 5 columns = 100 letters. But what happens if the number of elements we ask for is not a multiple of the number of elements in the matrix we describe?

```
(m2a <- matrix(letters[1:25], nrow = 21, ncol = 5))
```

```
## Warning in matrix(letters[1:25], nrow = 21, ncol = 5): data length [25] is not
## a sub-multiple or multiple of the number of rows [21]
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,] "a"  "v"  "r"  "n"  "j"
##  [2,] "b"  "w"  "s"  "o"  "k"
##  [3,] "c"  "x"  "t"  "p"  "l"
##  [4,] "d"  "y"  "u"  "q"  "m"
##  [5,] "e"  "a"  "v"  "r"  "n"
##  [6,] "f"  "b"  "w"  "s"  "o"
##  [7,] "g"  "c"  "x"  "t"  "p"
##  [8,] "h"  "d"  "y"  "u"  "q"
##  [9,] "i"  "e"  "a"  "v"  "r"
```

```
## [10,] "j"  "f"  "b"  "w"  "s"
## [11,] "k"  "g"  "c"  "x"  "t"
## [12,] "l"  "h"  "d"  "y"  "u"
## [13,] "m"  "i"  "e"  "a"  "v"
## [14,] "n"  "j"  "f"  "b"  "w"
## [15,] "o"  "k"  "g"  "c"  "x"
## [16,] "p"  "l"  "h"  "d"  "y"
## [17,] "q"  "m"  "i"  "e"  "a"
## [18,] "r"  "n"  "j"  "f"  "b"
## [19,] "s"  "o"  "k"  "g"  "c"
## [20,] "t"  "p"  "l"  "h"  "d"
## [21,] "u"  "q"  "m"  "i"  "e"
```

R does it anyway, but at least warns us that something might be wrong. It just goes until the matrix is filled out, recycling letters as it goes along. The extra 5 elements we asked for tacked on an extra "a,b,c,d,e" before the matrix was filled.

You can see these matrices get big (and this is a small one). If we just want to see the top, we use

```
head(m2)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,] "a"  "u"  "p"  "k"  "f"
## [2,] "b"  "v"  "q"  "l"  "g"
## [3,] "c"  "w"  "r"  "m"  "h"
## [4,] "d"  "x"  "s"  "n"  "i"
## [5,] "e"  "y"  "t"  "o"  "j"
## [6,] "f"  "a"  "u"  "p"  "k"
```

And if we just want to see the bottom, we use

```
tail(m2)
```

```
##       [,1] [,2] [,3] [,4] [,5]
## [15,] "o"  "j"  "e"  "y"  "t"
## [16,] "p"  "k"  "f"  "a"  "u"
## [17,] "q"  "l"  "g"  "b"  "v"
## [18,] "r"  "m"  "h"  "c"  "w"
## [19,] "s"  "n"  "i"  "d"  "x"
## [20,] "t"  "o"  "j"  "e"  "y"
```

Let's add some row and column names. For these, we will use the `paste()` function. `paste` is useful because it can create a vector of new words like so:

```
rownames(m2) <- paste("row", 1:20, sep = "")
colnames(m2) <- paste("col", 1:5, sep = "_")
dimnames(m2)
```

```
## [[1]]
##  [1] "row1"  "row2"  "row3"  "row4"  "row5"  "row6"  "row7"  "row8"  "row9"
## [10] "row10" "row11" "row12" "row13" "row14" "row15" "row16" "row17" "row18"
## [19] "row19" "row20"
##
## [[2]]
## [1] "col_1" "col_2" "col_3" "col_4" "col_5"
```

```
head(m2)
```

```
##      col_1 col_2 col_3 col_4 col_5
```

```
## row1 "a"   "u"   "p"   "k"   "f"
## row2 "b"   "v"   "q"   "l"   "g"
## row3 "c"   "w"   "r"   "m"   "h"
## row4 "d"   "x"   "s"   "n"   "i"
## row5 "e"   "y"   "t"   "o"   "j"
## row6 "f"   "a"   "u"   "p"   "k"
```

Checkpoint 9: What questions do you have about matrices?

Lists

Beginning R users may not find lists very useful, but as you become more advanced, you will find them invaluable. They are also the format of the output of many of R's statistical commands. At least for this class, it is important to know how to access lists. Let's first create one.

```
a <- list(c("hola mundo", "hello world", "hallo welt"),
          pi,
          c(rep(TRUE, 5), rep(FALSE, 5)),
          4.4)
a
```

```
## [[1]]
## [1] "hola mundo"  "hello world" "hallo welt"
##
## [[2]]
## [1] 3.141593
##
## [[3]]
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
##
## [[4]]
## [1] 4.4
```

```
length(a)
```

```
## [1] 4
```

```
mode(a)
```

```
## [1] "list"
```

```
class(a)
```

```
## [1] "list"
```

```
a[1] # This works but should be avoided
```

```
## [[1]]
## [1] "hola mundo"  "hello world" "hallo welt"
```

```
a[[1]] # This is the correct way to index lists
```

```
## [1] "hola mundo"  "hello world" "hallo welt"
```

Think of lists like grocery lists. They contain items in different categories: frozen foods, produce, dry goods, etc. Within each section, there might be a list of items. Here, we have four groups. The first group has 3 items, the second has 1 item, the third has 10 items, and the last has 1 item.

Imagine we want the 2nd item of the first group. You might think we try this

```
a[1][2]
```

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

But clearly that doesn't work. Here is the correct way to do it

```
a[[1]][2]
```

```
## [1] "hello world"
```

The double bracket indicates the group, the single bracket indicates the item within the group.

Dataframes

A dataframe is like a matrix, but each row corresponds to an individual in the population that is sampled, and each column corresponds to characteristics for that individual. Dataframes will form the foundation of most of the work we will be doing with R. In thinking about how to collect data for your own research:

1. When you design the experiment, you should know what kind of statistical test will be used to analyze the data.

2. So that you can can employ the pre-determined statistical test, you should know what kind of format your data should be in.

3. When you collect your data, you should collect it in a manner so that it will be entered in the correct format.

The format of dataframes are counter-intuitive to most people when they collect data for their experiments (or at least it was to me). For example, take an experiment in which clones were grown in three three temperature treatments: low, medium, and high and you measured their height. Many people might be tempted to write down the data in their lab notebook, or enter it in Excel, like this:

CloneID	LowHeight	MedHeight	HighHeight
1	12.4	15.6	16.2
2	10.4	11.1	12.0

For whatever reason, this is the format most people default to. One explanation is that this format makes it easy to calculate means in Excel. But for statistics, it's garbage. Here, note that we have 6 individuals with characteristics that are measured about them. Their characteristics are: their ID, their temperature treatment, and their height. We can organize our data into individuals on each row and characteristics in each column. Like this

CloneID	Temperature	Height
1	Low	12.4
1	Med	15.6
1	High	16.2
2	Low	10.4
2	Med	11.1
2	High	12.0

Your workflow should be to collect data in something like a lab notebook, which can be stored as a hard copy. Then, you can transcribe that data into a spreadsheet such as excel, google sheets, or numbers. These should be transcribed in plain text form, **without any bolding, cell coloring, or fomulas.** Such data files can be saved in one of two ways: as a .csv file (a comma separated file) or a .txt file (a text file). These don't

require much space and are easily interpretable by computing software, such as R. Both work equally well, although I prefer .csv files and they seem to be the general preference of statisticians and data scientists. So we will use them most of the time here.

Checkpoint 10: Create this data table in your favorite spreadsheet program and save it as "MyFirstData.csv". Somewhere on your computer, create a folder named "BIO380". Within that folder, create another named "Wk2". Place your data file in this "Wk2" folder.

Now, we get to what should be a very simple task but often proves to be a challenge: importing and loading this data into R.

Importing Data - It's Not as Simple as You Think

Before bringing data into R you have to tell R where the data are stored. Your computer has a vast amount of space, a bit like a giant office building with many floors, rooms, cabinets, drawers, and folders. For R to load data on your computer, you have to give it precise instructions to look exactly where the data are. R is looking for files currently on your computer. The place where it is looking is called your *working directory*. Want to see where it is looking right now? Type `getwd()` for "*get working directory*"

```
getwd()
```

```
## [1] "/Users/nicholaskortessis/Library/CloudStorage/GoogleDrive-kortessn@wfu.edu/My Drive/Import/Wake
```

This is my working directory while I was making this lab. Your's will be differnet. This says on my computer, look in Users, find the user 'nicholaskortessis', go to the folder 'Library' and in there "CloudStorage" and (you get the point).

If you want to change the working directory, you can use the function `setwd()` for, you guessed it, "set working directory". Let's tell R to look in my downloads folder.

```
setwd("/Users/nicholaskortessis/Downloads")
# Now let's see where R is looking
getwd()
```

```
## [1] "/Users/nicholaskortessis/Downloads"
```

I can clearly see it's looking in my Downloads folder.

Remember those folders I told you to make where you saved your data? Yep, tell R to look there using the `setwd()` function. The actual path to that folder will depend on where it lives on your computer, and so the code will be specific to your computer. Note the use of the forward slash to designate folders; you can use either a forward slash ("/") OR a backslash ("\""). Once you have done it correctly, you should see your newly created data file in the "Files" tab in your environment window.

Checkpoint 11: Once you have your working directory listed appropriately, call Dr. Kortessis over and show him before continuing. This is a great time to get help. In your R script, be sure to write out the working directory.

To upload different file formats into a dataframe to R, you can see the options at '?read.table'. Even if you have .csv or .delim files, you can always use "read.table" and specify the options you want. In this class, we will be working mostly with delimited files with headers and .csv files. To import from a .csv file use the command "read.csv".

Before you name an object in R, you should check that the name you want to use does not already exist in the R with the "?" command. For example, for a long time, I had tendency to name my dataframes "df." It turns out that is a bad idea because df already does duty for R as a function!

Checkpoint 12: Write in your R script what the function `df()` does by using the help command `?df`.

The first data frame we will be working with is the one you just made. Let's try to load it in R and save it as an object. We can use the function `read.csv()` to read csv files like this one. All we need is the name of the file. Below, I've been sure to set my working directory since I changed it above. But once you set it within a script, it stays there until you tell R to look elsewhere.

```
setwd("/Users/nicholaskortessis/Library/CloudStorage/GoogleDrive-kortessn@wfu.edu/My Drive/Import/Wake F
MyData.df <- read.csv(file = 'MyFirstData.csv')
MyData.df
```

```
##   CloneID Temperature Height
## 1       1         Low   12.4
## 2       1         Med   15.6
## 3       1        High   16.2
## 4       2         Low   10.4
## 5       2         Med   11.1
## 6       2        High   12.0
```

```
dim(MyData.df)
```

```
## [1] 6 3
```

You can see that this is a dataframe much like a matrix with 6 rows and 3 columns. Moreover, RStudio tells us how to interpret each item within this dataframe by giving its flavor. CloneID is an integer, Temperature is a character, and Height is a double (that is a number that is more than just an integer).

If we want just the height, we can call that using the dollar sign symbol and do things with it, like this:

```
MyData.df$Height
```

```
## [1] 12.4 15.6 16.2 10.4 11.1 12.0
```

```
mean(MyData.df$Height) # mean
```

```
## [1] 12.95
```

```
sd(MyData.df$Height) # standard deviation
```

```
## [1] 2.396456
```

```
quantile(MyData.df$Height, 0.5) # median (50% quantile)
```

```
##  50%
## 12.2
```

You can see how this is much better than calculating these things by hand. Alright, let's go work with a real dataset.

In some tutorials, they teach you to 'attach' the dataframe to your R session. THIS IS A BAD PRACTICE THAT YOU SHOULD NOT USE, and we will not use it in class. As you progress in your career, you may have to work with multiple dataframes simultaneously within the same project. Attaching datasets can cause problems in this scenario and generally makes code less interpretable.

We'll now look at a dataframe example from an R book (The R Book by Crawley) called 'Worms.txt'. This data file is on the canvas course page. Download it and save it in your Wk2 Folder within your BIO380 folder.

This dataframe is a text file, so open it in your favorite text editor (I use Notepad) before you try to open it in RStudio. In this study, the density of worms ("worm.density") was measured in several different sites (column "Field.Name"), and some features of the habitats were also monitored or noted (columns "Area", "Slope", "Vegetation", "Soil.pH", "Damp").

Once you have looked at the dataset in a text editor, set your working directory and load the data as the object `df.w` using the following code.

```
df.w <- read.table('worms.txt', header = T)
df.w
```

```
##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
## 2     Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
## 4        Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
## 7       Church.Field  3.5     3  Grassland     4.2 FALSE            3
## 8            Ashurst  2.1     0     Arable     4.8 FALSE            4
## 9        The.Orchard  1.9     0    Orchard     5.7 FALSE            9
## 10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 11       Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
## 15        Pond.Field  4.1     0     Meadow     5.0  TRUE            6
## 16      Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
## 17         Cheapside  2.2     8      Scrub     4.7  TRUE            4
## 18        Pound.Hill  4.4     2     Arable     4.5 FALSE            5
## 19        Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
## 20         Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

Checkpoint 13: Looking at this script, write down the statistical individuals in this sample and the general characteristics of each individual. Which of them are numerical and which are categorical? Of the numerical, which are continuous and which are discrete? Of the categorical, which are ordinal? Write these answers in your script.

Basic data frame manipulation

To do statistics with this data, we need to understand how to manipulate the dataframe. Let's begin with a summary of the 'flavors' of characteristics. Let's see what flavors the columns are

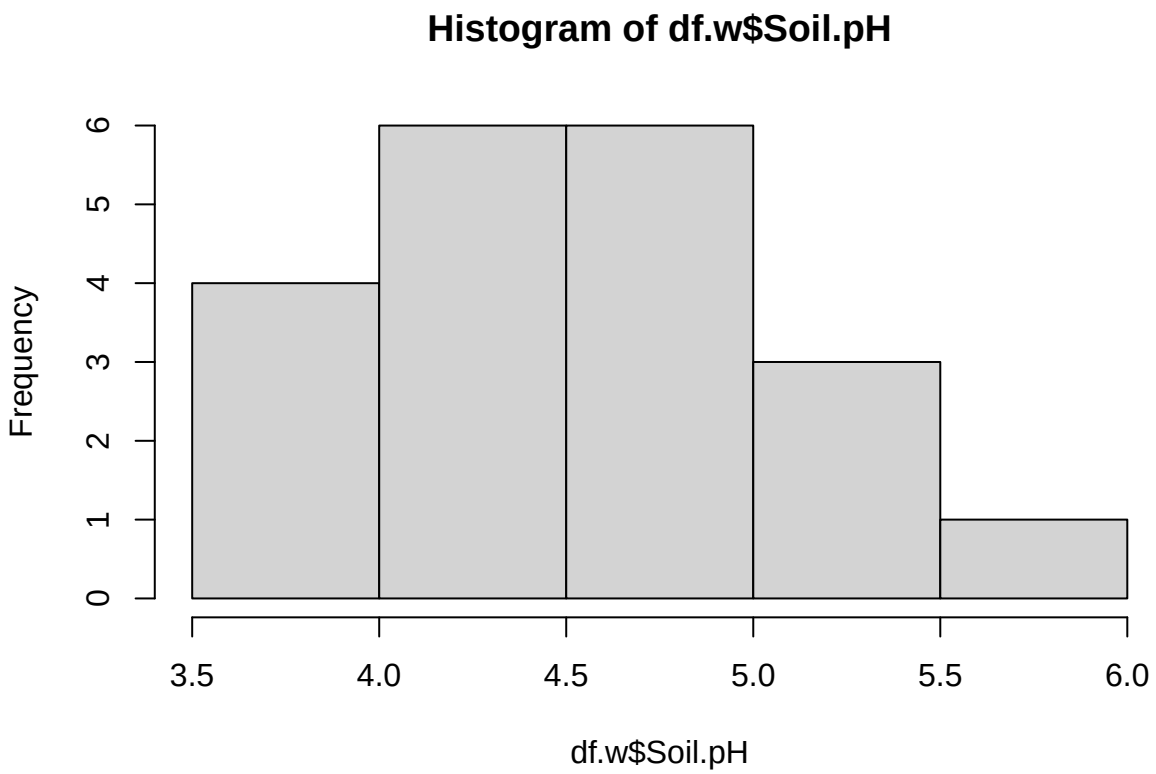
```
str(df.w)
```

```
## 'data.frame':    20 obs. of  7 variables:
##  $ Field.Name  : chr  "Nashs.Field" "Silwood.Bottom" "Nursery.Field" "Rush.Meadow" ...
##  $ Area        : num  3.6 5.1 2.8 2.4 3.8 3.1 3.5 2.1 1.9 1.5 ...
##  $ Slope       : int  11 2 3 5 0 2 3 0 0 4 ...
##  $ Vegetation  : chr  "Grassland" "Arable" "Grassland" "Meadow" ...
##  $ Soil.pH     : num  4.1 5.2 4.3 4.9 4.2 3.9 4.2 4.8 5.7 5 ...
##  $ Damp        : logi  FALSE FALSE FALSE TRUE FALSE FALSE ...
##  $ Worm.density: int  4 7 2 5 6 2 3 4 9 7 ...
```

See the dollar sign again? This means you can access each of these columns by writing `df.w$<column name>`.

Let's look at soil pH.

```
hist(df.w$Soil.pH)
```



How easy was that?

We can also find the names of the columns

```
names(df.w)
```

```
## [1] "Field.Name"   "Area"         "Slope"        "Vegetation"   "Soil.pH"
## [6] "Damp"         "Worm.density"
```

Many times, we are interested in comparing individuals with different characters. For example, what if we wanted to know how average soil pH varied across vegetation types. We can use the function `tapply`

```
?tapply
tapply(df.w$Soil.pH, df.w$Vegetation, mean)
```

```
##    Arable Grassland    Meadow   Orchard     Scrub
##  4.833333  4.100000  4.933333  5.700000  4.800000
```

Look at that. pH is most basic in orchard sites and most acidic in grassland sites.

Checkpoint 14: Write code to calculate the standard deviation of soil pH for each vegetation type using the tapply command.

`tapply` and other `apply` functions are super useful in R.

Sorting dataframes

It is common to want to sort a dataframe by rows, but rare to want to sort by columns. Note that to do this in R we will use the "order()" command rather than the "sort()" command, which is used to sort along one column (i.e., one vector) but not multiple columns (we want to sort the entire dataframe). Because we are sorting by rows (the first subscript) we specify the order of the row subscripts before the comma. Thus, to sort the dataframe on the basis of values in one of the columns (say, Slope), we write:

```
# Order by slope
df.w[order(df.w$Slope), ]
```

```
##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 8            Ashurst  2.1     0     Arable     4.8 FALSE            4
## 9        The.Orchard  1.9     0    Orchard     5.7 FALSE            9
## 15        Pond.Field  4.1     0     Meadow     5.0  TRUE            6
## 16      Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1
## 19        Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
## 2     Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
## 18        Pound.Hill  4.4     2     Arable     4.5 FALSE            5
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
## 7       Church.Field  3.5     3  Grassland     4.2 FALSE            3
## 10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 4        Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
## 17         Cheapside  2.2     8      Scrub     4.7  TRUE            4
## 11       Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
## 20         Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
## 1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
```

```
# Or, in reverse order
df.w[rev(order(df.w$Slope)), ]
```

```
##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
## 20         Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
## 11       Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
## 17         Cheapside  2.2     8      Scrub     4.7  TRUE            4
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
## 4        Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
## 7       Church.Field  3.5     3  Grassland     4.2 FALSE            3
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
## 18        Pound.Hill  4.4     2     Arable     4.5 FALSE            5
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
## 2     Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
## 19        Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1
## 16      Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
## 15        Pond.Field  4.1     0     Meadow     5.0  TRUE            6
## 9        The.Orchard  1.9     0    Orchard     5.7 FALSE            9
## 8            Ashurst  2.1     0     Arable     4.8 FALSE            4
## 5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
```

```
# Or, by Vegatation, then Slope, then Area
df.w[order(df.w$Vegetation, df.w$Slope, df.w$Area),]
```

```
##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 8            Ashurst  2.1     0     Arable     4.8 FALSE            4
```

```
## 18        Pound.Hill  4.4     2     Arable     4.5 FALSE           5
## 2     Silwood.Bottom  5.1     2     Arable     5.2 FALSE           7
## 19        Gravel.Pit  2.9     1  Grassland     3.5 FALSE           1
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE           1
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE           2
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE           2
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE           2
## 7       Church.Field  3.5     3  Grassland     4.2 FALSE           3
## 10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE           7
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE           0
## 1         Nashs.Field  3.6    11  Grassland     4.1 FALSE           4
## 16      Water.Meadow  3.9     0     Meadow     4.9  TRUE           8
## 15        Pond.Field  4.1     0     Meadow     5.0  TRUE           6
## 4        Rush.Meadow  2.4     5     Meadow     4.9  TRUE           5
## 9        The.Orchard  1.9     0    Orchard     5.7 FALSE           9
## 5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE           6
## 17         Cheapside  2.2     8      Scrub     4.7  TRUE           4
## 20         Farm.Wood  0.8    10      Scrub     5.1  TRUE           3
## 11       Garden.Wood  2.9    10      Scrub     5.2 FALSE           8
```

Subsetting dataframes

Many times we want to only see specific individuals within a dataframe. For example, maybe you want do something with only those sites with soil pH below 4.5. Doing that is called "subsetting" and it is one of the most important aspects of data management in R. It will take some time to get used to - but the more you master these techniques, the more powerful your capabilities of data management and analysis. Here, you will learn how to look at specific parts of the dataframe using:

- logical arguments
- sapply
- the `subset` command

Logical Arguments

Let's try and find all the ones with soil pH below 4.5, the acidic sites. First, we can ask whether each site satisfies this condition

```
df.w$Soil.pH < 4.5
```

```
##  [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
## [13]  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
```

Out pops a list of logical statements about whether this condition is TRUE. We can then plug this in as **row indices** in the dataframe. For example, if we have TRUE, TRUE, FALSE applied to a row dimension of a dataframe, R reads this as saying we want rows 1,2, and NOT 3. Thus, plugging in this logical statement into the rows gives us only those individual sites with pH < 4.5

```
df.w[df.w$Soil.pH <4.5,]
```

```
##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1         Nashs.Field  3.6    11  Grassland     4.1 FALSE           4
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE           2
## 5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE           6
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE           2
## 7       Church.Field  3.5     3  Grassland     4.2 FALSE           3
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE           1
```

```
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
## 19        Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
```

Maybe we just want the damp sites

```
df.w[df.w$Damp == TRUE,]
```

```
##       Field.Name Area Slope Vegetation Soil.pH Damp Worm.density
## 4    Rush.Meadow  2.4     5     Meadow     4.9 TRUE            5
## 10 Rookery.Slope  1.5     4  Grassland     5.0 TRUE            7
## 15    Pond.Field  4.1     0     Meadow     5.0 TRUE            6
## 16  Water.Meadow  3.9     0     Meadow     4.9 TRUE            8
## 17     Cheapside  2.2     8      Scrub     4.7 TRUE            4
## 20     Farm.Wood  0.8    10      Scrub     5.1 TRUE            3
```

Or maybe we want the Damp sites with soil pH < 4.5

```
df.w[df.w$Damp == TRUE & df.w$Soil.pH < 4.5]
```

```
## data frame with 0 columns and 20 rows
```

Look at that. R gave us nothing. Must have been an error, right? Not quite. In this case, we've done everything right. There are just no sites that are damp with soil that acidic. Look at the damp sites and look at the acidic soil sites to prove this to yourself.

We can also exclude some things. Let's look at everything but the grassland sites.

```
df.w[df.w$Vegetation != 'Grassland', ]
```

```
##         Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 2   Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
## 4      Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
## 5  Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 8          Ashurst  2.1     0     Arable     4.8 FALSE            4
## 9      The.Orchard  1.9     0    Orchard     5.7 FALSE            9
## 11     Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
## 15      Pond.Field  4.1     0     Meadow     5.0  TRUE            6
## 16    Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
## 17       Cheapside  2.2     8      Scrub     4.7  TRUE            4
## 18      Pound.Hill  4.4     2     Arable     4.5 FALSE            5
## 20       Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

sapply

As they like to say, there is more than one way to skin a cat. This is very true in R. There is no 'correct' way to do things. There are certainly wrong ways to do things, but the 'correct' way is the way that gets you to what you want. Using logicals is one approach.

Another is to use the function `sapply()`. Here is an example where we collect all the numeric characters for our individuals in this dataframe. The function `is.numeric()` is used here to evaluate whether something is a number.

```
sapply(df.w, is.numeric)
```

```
##   Field.Name         Area        Slope   Vegetation      Soil.pH         Damp
##        FALSE         TRUE         TRUE        FALSE         TRUE        FALSE
## Worm.density
##         TRUE
```

```
# Now we put this into the columns to tell we only want the numeric columns
df.w[ , sapply(df.w, is.numeric)]
```

```
##    Area Slope Soil.pH Worm.density
## 1   3.6    11     4.1            4
## 2   5.1     2     5.2            7
## 3   2.8     3     4.3            2
## 4   2.4     5     4.9            5
## 5   3.8     0     4.2            6
## 6   3.1     2     3.9            2
## 7   3.5     3     4.2            3
## 8   2.1     0     4.8            4
## 9   1.9     0     5.7            9
## 10  1.5     4     5.0            7
## 11  2.9    10     5.2            8
## 12  3.3     1     4.1            1
## 13  3.7     2     4.0            2
## 14  1.8     6     3.8            0
## 15  4.1     0     5.0            6
## 16  3.9     0     4.9            8
## 17  2.2     8     4.7            4
## 18  4.4     2     4.5            5
## 19  2.9     1     3.5            1
## 20  0.8    10     5.1            3
```

If we wanted the categorical variables, we could simply ask for those that are NOT numeric

```
df.w[ , !sapply(df.w, is.numeric)]
```

```
##           Field.Name Vegetation  Damp
## 1        Nashs.Field  Grassland FALSE
## 2     Silwood.Bottom     Arable FALSE
## 3      Nursery.Field  Grassland FALSE
## 4        Rush.Meadow     Meadow  TRUE
## 5    Gunness.Thicket      Scrub FALSE
## 6           Oak.Mead  Grassland FALSE
## 7       Church.Field  Grassland FALSE
## 8            Ashurst     Arable FALSE
## 9        The.Orchard    Orchard FALSE
## 10     Rookery.Slope  Grassland  TRUE
## 11       Garden.Wood      Scrub FALSE
## 12      North.Gravel  Grassland FALSE
## 13      South.Gravel  Grassland FALSE
## 14 Observatory.Ridge  Grassland FALSE
## 15        Pond.Field     Meadow  TRUE
## 16      Water.Meadow     Meadow  TRUE
## 17         Cheapside      Scrub  TRUE
## 18        Pound.Hill     Arable FALSE
## 19        Gravel.Pit  Grassland FALSE
## 20         Farm.Wood      Scrub  TRUE
```

Subset

My favorite is the function `subset()` which asks you how you want to subset your dataframe (which individual characteristics to subset from) and which characteristics you want to select.

First, let's look at the Vegetation types and worm density in all the acidic sites (pH < 4.5).

```
subset(df.w, subset = df.w$Soil.pH < 4.5, select = c('Vegetation', 'Worm.density'))
```

```
##    Vegetation Worm.density
## 1   Grassland            4
## 3   Grassland            2
## 5       Scrub            6
## 6   Grassland            2
## 7   Grassland            3
## 12  Grassland            1
## 13  Grassland            2
## 14  Grassland            0
## 19  Grassland            1
```

It's mostly grassland sites that are acidic.

We could also ask for all characteristics of the acidic sites by not specifying anything to 'select'. In that case, the function defaults to taking them all.

```
subset(df.w, subset = df.w$Soil.pH < 4.5)
```

```
##           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
## 1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
## 3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
## 5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
## 6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
## 7       Church.Field  3.5     3  Grassland     4.2 FALSE            3
## 12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1
## 13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
## 14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
## 19        Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
```

I love using subset. It makes sense to me and is very readable in terms of code. It says right there what you want when looking at portions of dataframes.

Checkpoint 15: Now that you have learned how to look at dataframes, write code in your script that answers the following questions.

1. **What is the average worm density in each vegetation type.**
2. **What is the average worm density in damp sites versus not damp sites?**
3. **Are meadows always damp?**
4. **What is the overall average worm density and standard deviation of worm density?**
5. **Make a histogram of worm density in grassland sites.**