Intro to R

2023-05-25

Contents

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
library(ggpubr)
library("FactoMineR")
library("factoextra")
```

```
## Welcome! Want to learn more? See two factoextra-related books at https://goo.gl/ve3WBa
```

August 30 | Introduction to R

August 30 | Introduction to R

- R is both a language and an interface for statistical analysis, programming, and graphics. R has become a standard interface for statistical analysis in biological sciences due in part to its openness, ability to be extended by users, and it vibrant user base. As a statistical analysis platform, R has its own grammar and in this activity you will begin to understand how to use and interpret R.

To get R and Rstudio (you'll need both), you can go here

Please have this by next class (Aug 30). This will be one of your assignments.

August 30 | Introduction to R

- R itself consists of an underlying engine that takes commands and provides feedback on these commands. Each command you give the R engine is either an:
- Expression

An expression is a statement that you give the R engine. R will evaluate the expression, give you the answer and not keep any reference to it for future use. Some examples include:

```
2 + 6
```

```
## [1] 8
```

```
sqrt(5)
```

```
## [1] 2.236068
```

```
3 * (pi/2) - 1
```

```
## [1] 3.712389
```

August 30 | Introduction to R

-Assignment

An assignment causes R to evaluate the expression and stores the result in a variable. This is important because you can use the variable in the future. An example of an assignment is:

```
x <- 2 + 6
myCoolVariable <- sqrt(5)
another_one_number23 <- 3 * (pi/2) - 1
x
```

```
## [1] 8
```

```
myCoolVariable
```

```
## [1] 2.236068
```

```
another_one_number23
```

```
## [1] 3.712389
```

August 30 | Introduction to R

- Functions

There are thousands of potential functions in R and its associated packages. To use these functions, you need to understand the basic taxonomy of a function. A function has two parts: - A unique name, and - The stuff (e.g., variables) passed to it within the parentheses.

August 30 | Introduction to R

Not all functions need any additional variables. For example, the function ls() shows which variables R currently has in memory and does not require any parameters. If you forget to put the parentheses on the function and only use its name, by default R will show you the code that is inside the function (unless it is a compiled function). This is because each function is also a variable. This is why you should not use function names for your variable names (see below for more on naming).

August 30 | Introduction to R

To find the definition of a function, the arguments passed to it, details of the implementation, and some examples, you can use the ? shortcut. To find the definition for the sqrt() function type ?sqrt and R will provide you the documentation for that function.

August 30 | Introduction to R

Functions may have more than one parameter passed to it. Often if there are a lot of parameters given then there will be some default values provided. For example, the log() function provides logarithms. The definition of the log function show log(x, base=exp(1)) (say from ?log). Playing around with the function shows:

```
log(2)
```

```
## [1] 0.6931472
```

```
log(2, base = 2)
```

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

```
log(2, base = 10)
```

```
## [1] 0.30103
```

August 30 | Introduction to R

R recognizes over a dozen different types of data. All of the data types are characterized by what R calls classes. To determine the type of any variable you can use the built-in function class(x). This will tell you what kind of variable x is. What follows are some of the more common data types.

August 30 | Introduction to R

- Numeric

Numeric types represent the majority of numerical valued items you will deal with. When you assign a number to a variable in R it will most likely be a numeric type. Numeric data types can either be displayed with or without decimal places depending if the value(s) include a decimal portion. In fact, R will make any assignment of a numerical value a numeric by default. For example:

August 30 | Introduction to R

```
x <- 4
class(x)
```

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

```
x
```

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

```
x <- numeric(4)
x
```

```
## [1] 0 0 0 0
```

August 30 | Introduction to R

```
x[1] = 2.4
x
```

```
## [1] 2.4 0.0 0.0 0.0
```

August 30 | Introduction to R

Notice this is an all or nothing deal here, each element of a vector must be the same type and the de- fault value for a numeric data types is zero. Also notice (especially those who have some experience in programming other languages) that dimensions in vectors (and matrices) start at 1 rather than 0. Operations on numeric types proceed as you would expect but since the numeric type is the default type, you don't really have to go around using the as.numeric(x) function. For example:

August 30 | Introduction to R

```
is.numeric(2.4)
```

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

```
as.numeric(2) + 0.4
```

```
## [1] 2.4
```

```
2 + 0.4
```

```
## [1] 2.4
```

August 30 | Introduction to R

- Numeric

Word of Caution, It is important to point out here that you need to be rather careful when dealing with floating point numbers due in part to the way in which computers store these numbers and how they are presented to us in the R interface as well as when we need to perform logical operations on them. Consider the following case. The ancient Egyptians had an approach to calculating pie as the ratio of 256/81.

```
e.pi <- 256/81
e.pi
```

```
## [1] 3.160494
```

August 30 | Introduction to R

- Numeric

Word of Caution cont'd: Very nice and apparently pretty close to 3.1416 so that they could get work done. Now, as we all know, the value of pie is the ratio of a circle's circumference to its diameter. We also know that it is a transcendeental number (e.g., on that cannot be produced using finite algebraic operations) and its decimal values never repeat.

```
print(e.pi, digits = 20)
```

```
## [1] 3.1604938271604936517
```

August 30 | Introduction to R

- Numeric

Word of Caution cont'd: There is another issue that you need to be careful with. You need to be considerate of how a computer stores numerical values. Consider the following:

```
x <- 0.3/3
x
```

```
## [1] 0.1
```

```
print(x, digits = 20)
```

```
## [1] 0.09999999999999991673
```

August 30 | Introduction to R

- Numeric

Why the difference? A computer deals in binary (0/1) representations and as such has a limited ability for precision, particularly for very large or very small numbers. Usually this does not cause much of a problem, but when you begin to work at crafting analyses, you should be aware of this drawback.

August 30 | Introduction to R

- Character

The character data type is the one that handles letters and letter-like representations of numbers. For example, observe the following:

```
x <- "If you can read this, you are beginning to take a step into a larger world."
class(x)
```

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

```
length(x)
```

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

August 30 | Introduction to R

- Character

Notice here how the variable x has a length of one, even though there are 37 characters within that string. If you want to know the number of characters, you need to use the nchar() function, otherwise it will tell you the 'vector length' (see below) of the variable.

```
y <- 23
class(y)
```

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

```
z <- as.character(y)
z
```

```
## [1] "23"
```

```
class(z)
```

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

August 30 | Introduction to R

- Character

Notice how the variable y was initially designated as a numeric type but if we use the as.character(y) function, we can coerce it into a non-numeric representation of the number. Combining character variables can be done using the paste() function to 'paste together' a string of char- acters (n.b., notice the optional sep argument).

August 30 | Introduction to R

```
w = "cannot"
x = "I"
y = "can"
z = "code in R"

paste(x, w, z)
```

```
## [1] "I cannot code in R"
```

```
paste(x, y, z)
```

```
## [1] "I can code in R"
```

August 30 | Introduction to R

- Constants

Constants are variables that have a particular value associated with them that cannot be changed. They are mostly here for convenience so that we do not have to go look up values for common things. Below are listed some common constants that you will probably encounter as you play with R.

August 30 | Introduction to R

- Constants

```
knitr::include_graphics("Constants_table.png")
```

Table 1: Common constants you will run across in R

Constant	Description
pi	The mathematical constant, π representing the ratio of a circles circumference to its diameter.
NULL	The absence of a type. This is the oubliette, complete nothingness, `/dev/null` Richmond on a Wednesday night... This is commonly used by functions that return undefined responses.
nan	Not a number.
Inf	Infinity (∞) as well as -Inf for $-\infty$.
NA	Typically used to represent something that is not there or missing. You can use it for missing data if you like.

August 30 | Introduction to R

- Constants

For the non-numerical constants, there are commands such as is.NULL(), is.nan(), is.infinite (and its cousin is.finite()), and is.na() to help you figure out if particular items are of that constant type if you like. At times this can be handy such when you have missing data and you want to set it to some meaningful value (e.g, is.na(X) <- 32 will set all N A values in X to 32). We'll get into this more in depth at a later time.

August 30 | Introduction to R

- Logical

Logical data types are boolean variables with a value of TRUE or FALSE. Obviously, these two values are the opposites of each other (e.g., not TRUE is FALSE, etc.). You will encounter logical data types in two primary situations;

- When you are writing a conditional statement that requires you to know the truth about something (e.g., if x == 0 you probably shouldn't try to divide by x because for some reason mathematicians haven't figured out how to divide by zero yet...), or
- If you are tying to select some subset of your data by using a particular condition (e.g.,select all entries where color == "blue").

The interesting thing about logical variables is that numbers can be coerced into a logical variable. For example the number zero, as an integer, numeric, complex, or raw data type, is considered to be FALSE whereas any non-zero value is considered TRUE.

August 30 | Introduction to R

- Factors

Factors are a particular kind of data that is used in statistics associated with treatments. You can think of a factor as a categorical treatment type that you are using in your experiments (e.g., Male vs. Female or Treatment A vs. Treatment B vs. Treatment C). Factors can be ordered or unordered depending upon how you are setting up you experiment. Most factors are given in as characters so that naming isn't a problem. Below is an example of five observa- tions where the categorical variable sex of the organism is recorded.

August 30 | Introduction to R

- Factors

```
sex <- factor(c("Male", "Male", "Female", "Female", "Unknown"))
levels(sex)
```

```
## [1] "Female"  "Male"    "Unknown"
```

```
table(sex)
```

```
## sex
##  Female    Male Unknown
##       2       2       1
```

```
sex[5] <- "Male"
```

August 30 | Introduction to R

- Factors

Here the table() function takes the vector of factors and makes a summary table from it. Also notice that the levels() function tells us that there is still an "Unknown" level for the variable even though there is no longer a sample that has been classified as "Unknown" (it just currently has zero of them in the data set).

August 30 | Introduction to R

- Collections

While alluded to previously, working with single numbers, factors, or local types are OK, but we often work with collections of data and R has some built-in objects that handle different assortments of basic data types.

August 30 | Introduction to R

A collection of several values, all of the same type, are held in a vector. In fact, all base data types can be created as vectors using the c() function (c for combine).

August 30 | Introduction to R

-Vectors

```
x <- c(1, 2, 3)

y <- c(TRUE, TRUE, FALSE)
y
```

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

```
z <- c("I", "can", "code", "in", "R")
z
```

```
## [1] "I"    "can"  "code" "in"   "R"
```

August 30 | Introduction to R

To access an element in a vector, R uses square brackets ([]) as demonstrated here:

```
x
```

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

```
x[1] <- 2
x[3] <- 1
x
```

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

```
x[2]
```

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

August 30 | Introduction to R

Sequences of numbers are so common in analyses that there are several helper functions that assist you.

```
x <- 1:6
x
```

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

```
y <- seq(1, 6)
y
```

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

```
z <- seq(1, 20, by = 2)
z
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

```
rep(6, 4)
```

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

August 30 | Introduction to R

The notion x : y provides a vector of whole numbers from x to y. In a similar fashion the function seq(x,y,by=z) provides a sequence of numbers from x to y but can also have the optional parameter by= to determine how the sequence is made (in this case the by 2s for all the odd numbers from 1 to 20). The function rep(x,y) repeats x a total of y times.

August 30 | Introduction to R

- Matrices

Matrices are 2-dimensional vectors and can be created using the default constructor matrix() function. However, since they have 2-dimensions, you must tell R the size of the matrix that you are interested in creating by passing it a number for nrow and ncol for the number of rows and columns.

August 30 | Introduction to R

- Matrices

```
matrix(nrow = 2, ncol = 2)
```

```
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

```
matrix(23, nrow = 2, ncol = 2)
```

```
##      [,1] [,2]
## [1,]   23   23
## [2,]   23   23
```

August 30 | Introduction to R

- Matrices Matrices can be created from vectors as well.

August 30 | Introduction to R

```
x <- c(1, 2, 3, 4)
x
```

```
## [1] 1 2 3 4
```

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

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

```
is.matrix(x)
```

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

```
matrix(x)
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

```
y <- matrix(x, nrow= 2)

is.matrix(y)
```

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

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

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

August 30 | Introduction to R

There is a slight gotcha here if you are not careful.

```
x <- 1:4
matrix(x, nrow = 4, ncol= 2)
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
## [3,]    3    3
## [4,]    4    4
```

```
matrix(x, nrow = 3)
```

```
## Warning in matrix(x, nrow = 3): data length [4] is not a sub-multiple or
## multiple of the number of rows [3]
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    1
## [3,]    3    2
```

```
matrix(seq(1, 8), nrow = 4)
```

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

August 30 | Introduction to R

Notice here that R added the values of x to the matrix until it got to the end. However, it did not fill the matrix so it started over again. In the first case the size of x was a multiple of the size of the matrix whereas in the second case it wasn't but it still assigned the values (and gave a warning). Finally, as shown in the last case, if they are perfect multiples, then it fills up the matrix in a column-wise fashion. To access values in a matrix you use the square brackets just as was done for the vector types. However, for matrices, you have to use two indices rather than one.

August 30 | Introduction to R

```
X <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
X[1, 3]
```

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

```
X[2,2] <- 3.2
X
```

```
##      [,1] [,2] [,3]
## [1,]    1  3.0    5
## [2,]    2  3.2    6
```

```
X[1, ]
```

```
## [1] 1 3 5
```

```
X[ , 3]
```

```
## [1] 5 6
```

August 30 | Introduction to R

The last two operations provide a hint as to some of the power associated with manipulating matrices. These are slice operations where only one index is given (e.g., X[1,]) provide a vector as a result for the entire row or column.

August 30 | Introduction to R

It is at this point that we get into some compounded information. R no longer just works in the space you were looking at above. It use to be useful to simply manipulate data in base R as you just saw, but it gets more and more cumbersome as your data complexity intensifies.

On top of that, R also uses "packages" to work with increasingly complex data to make it more computationally effecient as well as make it easier to look at visually.

August 30 | Introduction to R

Given that we'll be looking at decently large datasets by the end of the semester, and for your future adventures in R, it will be necessary to get some packages to help with data manipulation as well as visualization.

In our case we'll start with the following:

- Tidyverse
- ggpubr

August 30 | Introduction to R

I highly encourage you to implement Tidyverse with all your data. Lists and regular data.frames are kind of a thing of the past. data.frames in Tidyverse are called "tibbles". "Tibbles" are essentially data frames with a meta data frame that runs in the background so it is easy for you to call data. Instead of having to call the specific row or column or call the row or column by $ and the name with the data frame name every time; because of the meta data frame of a tibble, you can just call the variable. This is super useful with trimming data in various forms (filter, select, mutate, etc.).

August 30 | Introduction to R

Ok, so now that we've gone through everything necessary to get started, your assignment coming into class is to get as class to having R and Rstudio installed on your computer. If you get past that, then load in Tidyverse and ggpubr onto R in Rstudio via the install.packages(" ") command.

In class we will: - Make sure you're caught up on your R things from above - Take a look at some sample data for Phab from VADEQ. - Take a look at some of our data we just collected as practice.

August 30 | Introduction to R

- From above, if you haven't installed R and Rstudio, let's work on that.
- If you haven't gotten tidyverse and ggpubr installed, let's work on that.

August 30 | Introduction to R

Before getting started, I highly recommend taking a look at the ggpubr website as well as the writer's blog for more information and options about how the ggpubr package works. It's a wrapper package that makes generating publication ready plots a little easier than using just ggplot2 from the tidyverse.

August 30 | Introduction to R

Once you have that, load in the data I have shared with you that's publicly available 2018 DEQ phab data.

```
test <- read_csv("DEQ_2018_Probmondata.csv")
```

```
## Rows: 34 Columns: 206
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (16): DataSource, StationID, StationID_Trend, state, status, comment, B...
## dbl (173): Year, LongitudeDD, LatitudeDD, stratum, designweight, weightcateg...
## lgl  (17): ARSENICppm, BERYLLIUMppm, CADMIUMppm, CHROMIUMppm, COPPERppm, LEA...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
Phab_var <- read_csv("Phab_variables.csv")
```

```
## Rows: 50 Columns: 2
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Variable, Physical_Hab_character
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# remember you can use variables to call any data frame etc. anything you want, so you don't have to us
```

August 30 | Introduction to R

Remember that you can use tidyverse to manipulate your data. See this tidyverse cheatsheet for reference. In our case, we'll need to match the variables to the variables we'll collect in our own Phab dataset (approximately), and then remove any character class variables we shouldn't include so we only look at variables with is.numeric() in class.

```
deq_filter <- test %>% select(Phab_var$Variable)

deq_pca <- deq_filter %>% select(-StationID, -Basin, -SubBasin, -BayShed, -EcoRegion)
# filter out character class variables
```

August 30 | Introduction to R

Remember, there are a lot of ways to pipe variables in tidyverse. See the cheat sheet. You don't always have to do one pipe at a time or use just one type of pipe in a single chunk of code. It will make your code effecient and is the reason why tidyverse in R is so useful.

August 30 | Introduction to R

In our case, this data is inherently multivariable so, we should reduce the dimesionality of the data to find what about the data drives what we see in the environment and then see what sort of resolution we have

within those variables. This will allow us to see what aspects of physical habitat are "good" versus "bad", and what aspects of the environment are driving the "good" and "bad" qualities.

August 30 | Introduction to R

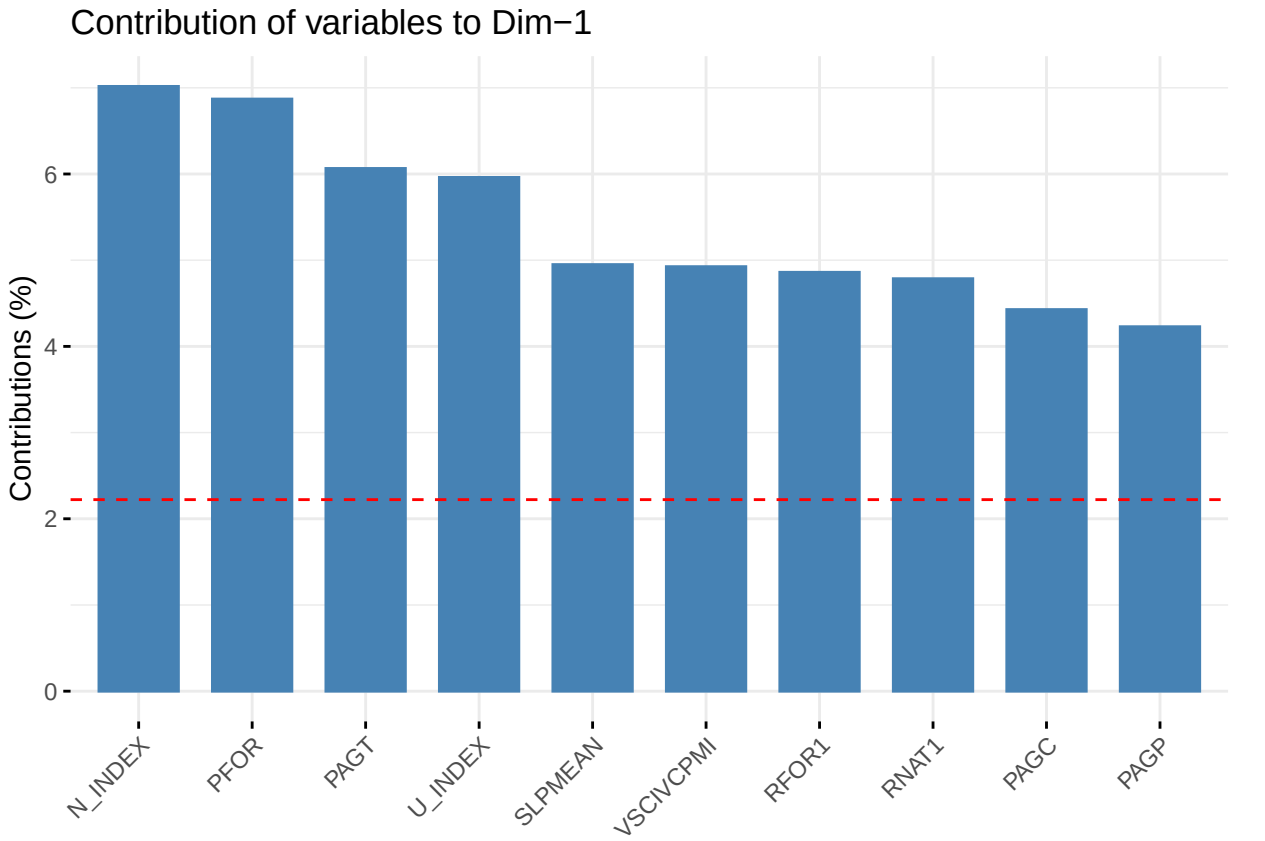
If you all want to see the dimensionality reduction now, we can briefly discuss it however, we will talk at more length about these sorts of statistics later in the semester. I have done the dimensionality reduction for you in this case so we can just look at the resolution of the responsible variables as an exercise for R.

August 30 | Introduction to R

```
res.pca <- PCA(deq_pca, scale.unit = TRUE, graph = FALSE)
```

```
## Warning in PCA(deq_pca, scale.unit = TRUE, graph = FALSE): Missing values are
## imputed by the mean of the variable: you should use the imputePCA function of
## the missMDA package
```

```
fviz_contrib(res.pca, choice = "var", axes = 1, top = 10)
```




```
# look at contributing variables to PC1
```

This plot shows the major contributing variables to physical habitat quality. Next, we'll take a look at the correlation of the data with the habitat scores.

August 30 | Introduction to R

```
cor(deq_filter$PFOR, deq_filter$VSCIVCPMI)
```

```
## [1] 0.6666747
```

August 30 | Introduction to R

```
cor(deq_filter$PAGT, deq_filter$VSCIVCPMI)
```

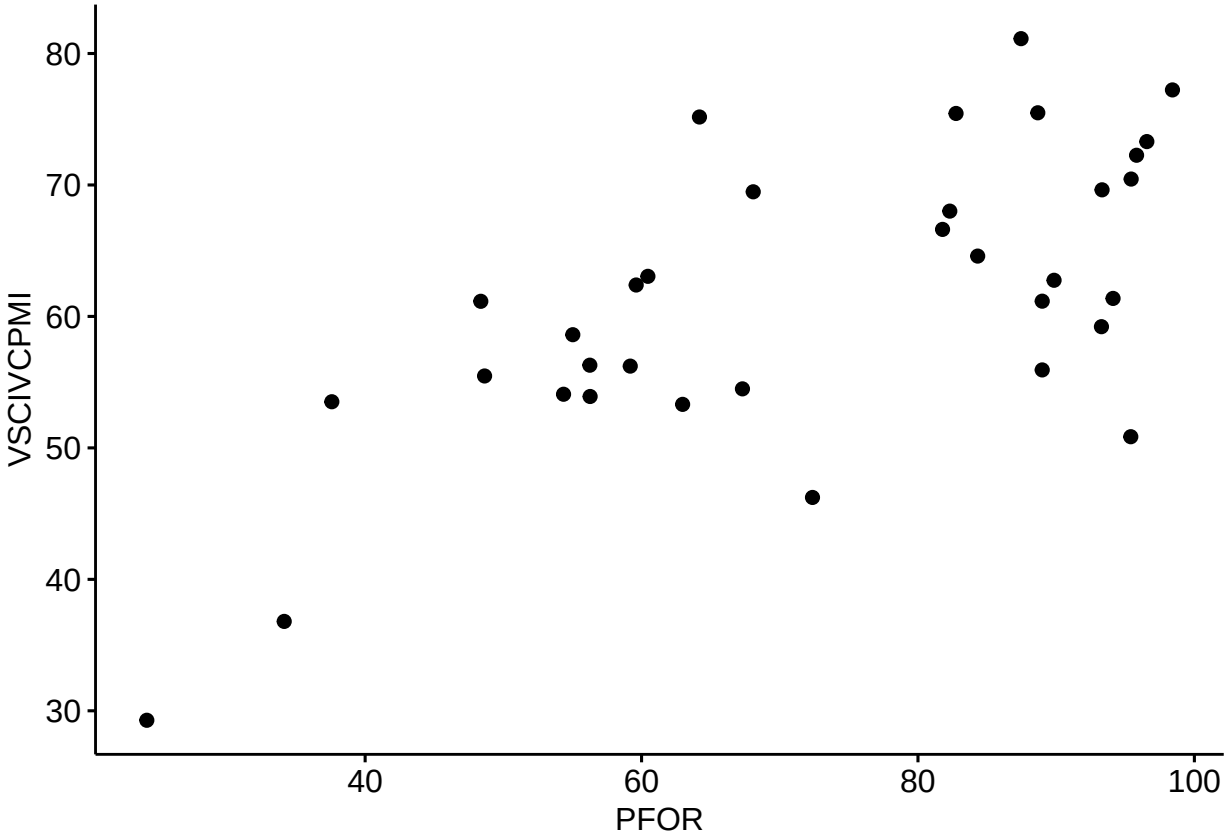
```
## [1] -0.7320828
```

We can see that our VSCI score is positively correlated with forested cover, and negatively correlated with agricultural cover. This may be intuitive but why do you think these two aspects of the physical habitat near the watershed effect the overall scores in the way that they do?

August 30 | Introduction to R

Let's plot the same data to visualize the relationship and play with ggpubr a bit.

```
ggscatter(deq_filter, x = "PFOR", y = "VSCIVCPMI")
```

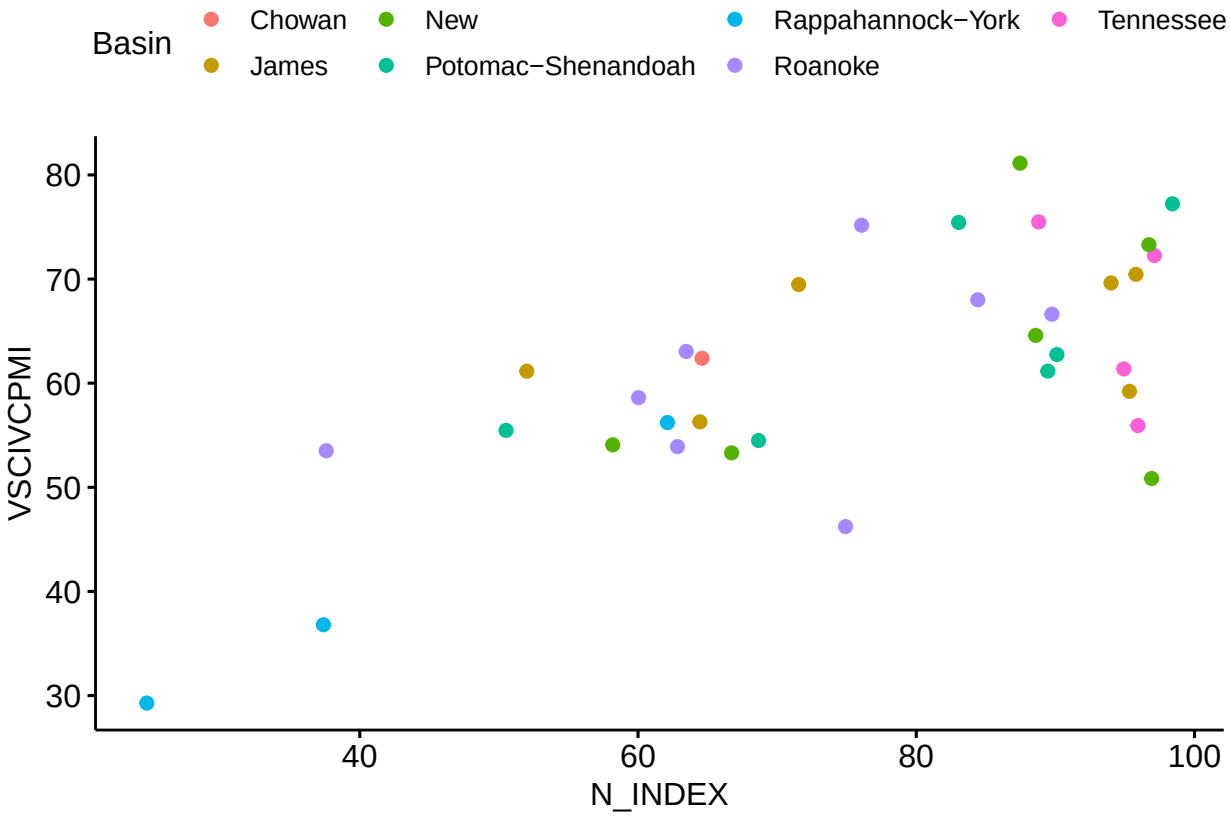


See if you can do this with PAGT as well.

August 30 | Introduction to R

Let's see if we can add some color by looking at watersheds.

```
ggscatter(deq_filter, x = "N_INDEX", y = "VSCIVCPMI", color = "Basin")
```



August 30 | Introduction to R

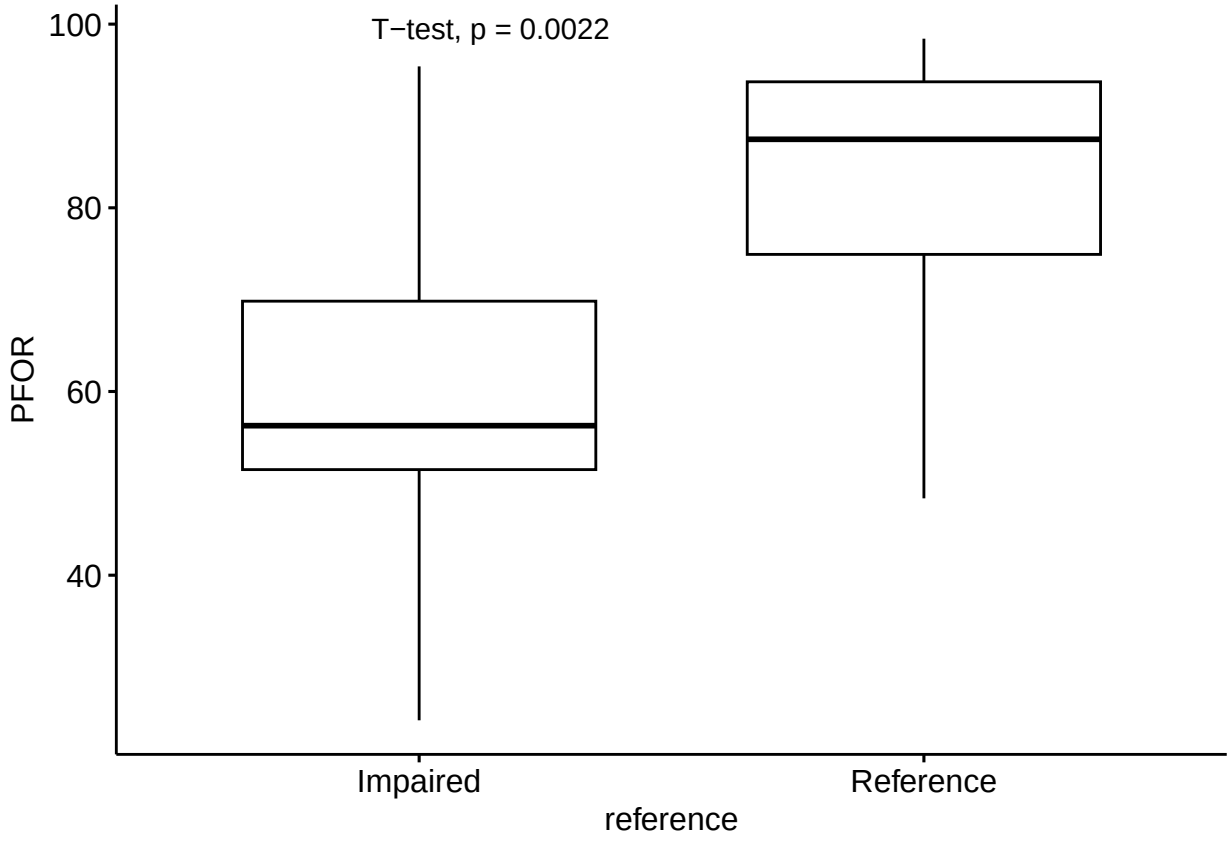
Finally, let's return to our two variables of interest, "PFOR" and "PAGT" and see if they have the power to discern reference and impaired streams on their own. In order to do this, we need to create a new column in our data frame "deq_filter" that shows both reference and impaired streams. I have done some homework here for you too as we know that the cutoff for VSCI scores is 60. Let's create that column using tidyverse.

```
deq_filter <- deq_filter %>% mutate(reference = if_else(VSCIVCPMI > 60, "Reference", "Impaired"))
# mutate will create a new column based off all sorts of conditions.  In our case, we need to create a
```

August 30 | Introduction to R

That's really good practice in applying the logic of R. Let's take a look and see if when we compare impaired versus reference streams, if we have the statistical power necessary to discern between them using a t.test.

```
 p <- ggboxplot(deq_filter, x = "reference", y = "PFOR")
p + stat_compare_means(method = "t.test")
```




```
# calling a plot with a variable is nice because then if you add something like stat_compareremeans, you'
```

August 30 | Introduction to R

See if you can do this for PAGT, and add labels on the axes as well as a title. Keep practicing! To get good at R, like anything else, it takes practice.