UMN EEB Quant/Comp Repo

Table of contents

Hello!

This site/book will serve as a repository for resources related to computational and quantitative workshops held for the University of Minnesota's Ecology, Evolution, and Behavior Graduate Program through its "Friday Noon Seminar" series.

It is a "living document," and thus will be subject to change and be updated over time.

Contact: Chris Wojan, wojan002 (at umn.edu)

Part I

Iteration: Loops, Functions, and the purr package

1 Intro to Iteration

This workshop will be focused on **iteration**, or performing the same code numerous times, potentially on some range of values, in R. It assumes that you know most of the basics of R programming.

R has various ways to iterate, including looping, using functions/functionals, and vectorization. We will go over each to some extent, with examples included. A general guideline for iterating is that if you are copying and pasting code multiple times, you may be better served with a loop or a function. We will cover the following:

1. Loops
    1. Types of loops: for, while, and repeat
    2. When to use which types of loops
    3. Potential downsides of using loops
    4. When use a loop is still necessary anyway
2. Functions and Functionals
    1. How to write functions
    2. Benefits of functions
    3. How to iterate functions with the purr package
    4. How to iterate even faster with parallel computing
3. Vectorization
    1. Benefits of vectorization
    2. Limitations

2 Types of Loops

The first method of iteration that most people learn is looping, or using flow control to rerun a block of code some number of times, or over a range of values. Many of you will be familiar with this, so we will just review it a bit.

R has different type of loops, including **for**, **while**, and **repeat**.

2.0.1 for loops

for loops are the most commonly used loop, as they are versatile to many contexts. They generally follow the form:

```
for (indexing_variable in range_of_values) {
  some_function(some_argument = indexing_variable)
}
```

In the first line, an indexing variable is created to keep track of iterations, and it iterates through the range of values provided.

Then, some code is provided within brackets to be performed for each value in the range of values. Usually, the indexing variable will be referenced somehow in this code, but it doesn't need to be if you are just doing the same exact thing several times.

Here is a really silly working example:

```
## square the numbers 1 through 5
for (i in 1:5) {
  print(i^2)
}
```

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

```r
## of course you could just do this instead and get a more useable vector
(1:5)^2
```

```
[1]  1  4  9 16 25
```

Now let's look at a more interesting example. Let's say you are interested in the effect of some independent variable on multiple dependent variables. First let's simulate some fake data of greenhouse plant growth under two watering conditions:

```r
## create a data frame with 60 total plants and
## -two treatment levels, control and drought
## -plant heights, masses, and seed masses sampling from a normal distribution
plant_data <- data.frame(treatment = rep(c("control","drought"), each = 30),
                         height_cm = rnorm(n = 60, mean = rep(c(12,8), each = 30), sd = 2),
                         dry_biomass_g = rnorm(60, mean = rep(c(50, 40), each = 30), sd = 6)
                         seed_mass_mg = rnorm(60, mean = rep(c(30, 20), each = 30), sd = 4))
```

Now we are probably interested in how the watering influences these variables (i.e., with a t-test), but we don't want to write the code, and then copy and paste it twice. Here's how we use a for loop:

```r
## create a vector of dependent variables to loop over
dep_vars <- c("height_cm", "dry_biomass_g", "seed_mass_mg")

## pre allocate a list to store results
result_list <- vector(mode = "list", length = length(dep_vars))

## let's name the list as well
names(result_list) <- dep_vars

## set up a loop to iterate through variables
for (i in dep_vars) {
  ## store t test results in our list
  result_list[[i]] <- t.test(plant_data[,i] ~ plant_data[,"treatment"])
}

## check out the results!
result_list
```

```
$height_cm
```

```
    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 7.7608, df = 56.6, p-value = 1.794e-10
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
 2.746624 4.657314
sample estimates:
mean in group control mean in group drought
            11.803027              8.101058


$dry_biomass_g

    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 6.3942, df = 56.457, p-value = 3.344e-08
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
  7.343268 14.041765
sample estimates:
mean in group control mean in group drought
             51.25064              40.55813


$seed_mass_mg

    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 10.608, df = 54.776, p-value = 6.799e-15
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
  7.52219 11.02684
sample estimates:
mean in group control mean in group drought
             29.07882              19.80430
```

When you are writing a for loop (especially ones more complex than the above), it can be helpful to first develop the code within the loop to more easily debug it. For example:

```r
## set the iterator variable to one of the values to be looped through
i <- dep_vars[1]

## then try coding what you want to do repeatedly
test_result <- t.test(plant_data[,i] ~ plant_data[,"treatment"])
test_result
```

```
    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 7.7608, df = 56.6, p-value = 1.794e-10
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
 2.746624 4.657314
sample estimates:
mean in group control mean in group drought
            11.803027              8.101058
```

This can be helpful because R, like most programming languages, is very particular about syntax. If you had tried to use the $ operator to select columns, you would have gotten an error:

```r
## this doesn't work because i is a character value in quotes, which is incompatible with $
t.test(plat_data$i ~ plant_data$treatment)
```

There are some "common pitfalls" to writing for loops, which I hope I have avoided here. You can read about them in more detail here:

https://adv-r.hadley.nz/control-flow.html#common-pitfalls

2.0.2 while loops

while loops are a bit less common, but can be useful in certain contexts (mostly simulation). They follow this form:

```r
while (some_condition == TRUE) {
  some_function()
}
```

The first line provides a conditional statement to be evaluated at each iteration, if it is met the loop will continue to run, if it is not met the loop will close.

There are two uses for while loop that I have found. The first is when there is some degree of randomness in how many iterations need to be run. Let's say for example you want to simulate a population of organisms that aggregate in groups, but you want to allow the group sizes to vary while the global population size remains the about the same. In this case, you could use a while loop:

```
## first let's set the population size to 0
N <- 0

## and create an empty vector for group size
groups <- NULL

## and a counter variable
counter <- 1

## now let's start the while loop
while (N < 100) {
  ## randomly generate a group size as a pull from a poisson distribution w/ a mean of 5
  groups[counter] <- rpois(n = 1, lambda = 5)
  ## calculate total pop size
  N <- sum(groups)
  ## increase counter
  counter <- counter + 1
}
```

If you run that code multiple times, you'll see that you will end up with different numbers of groups with different numbers of members. This could be useful for simulation models or parametric bootstrapping. However, you may also notice that the total population size here isn't exactly constant. To keep it constant, you could do something like this:

```
## first let's set the population size to 0
N <- 0

## and create an empty vector for group size
groups <- NULL

## and a counter variable
counter <- 1

## now let's start the while loop
```

```
while (N < 100) {
  ## calculate remaining population to fill
  to_fill <- 100 - sum(groups)
  ## randomly generate a group size as a pull from a poisson distribution w/ a mean of 5
  ## but if it is larger than the remaining population to fill, just fill it up exactly
  groups[counter] <- min(rpois(n = 1, lambda = 5), to_fill)
  ## calculate total pop size
  N <- sum(groups)
  ## increase counter
  counter <- counter + 1
}
```

Nifty!

The other use of a while loop that I have found is for a brute force algorithm. Basically, if you have do something complex but don't have the time or mental energy to figure out the precise algorithm to do it, you can do it the "stupid" way. This is basically like trying every combination on a lock until it opens. Computer scientists frown on these methods for being highly inefficient, but we are ~~computer~~ scientists. For ecology/evolution/behavior contexts, you might use a brute force while loops to try random combinations of data or sample sets of observations with many characteristics until you find a combination/set that meets highly specific criteria that you are looking for (e.g., subsampling a large dataset such that you sample randomly but also cover a wide range of treatment levels, sampling dates, etc.).

2.0.3 repeat loops

repeat loops are a very basic sort of loop that simply do something over an over again until a condition is met, then they close. Here's the form:

```
repeat {
  some_function()
  if (some_condition == TRUE) break
}
```

To me they kind of seem like a while loop formatted differently. I suppose you could add multiple "break" points for different end conditions, but I believe you can add "break" commands to any type of loop.

Let me know if you know of cases where repeat loops are effective!

3 When to Loop?

Now you may have often heard that you should avoid using for loops because they are slow. This isn't strictly true; it depends how you write them. It has to do with how R uses your computer's working memory, but the details are probably not worth getting into. Generally if you avoid the "common pitfalls" linked in the last section, you should be writing loops that are plenty fast. Also, in my personal opinion, I'm not sure how important speed is for most of our work - since we're not developing software or data pipelines for business purposes, we are more interested in the end result and less interested in how long it took our program to get there. All this to say, don't worry about the for loops you use being inefficient. A good guideline might be: do what you need to do now to get your results, and maybe do it a little better next time.

However, there are other reasons not to use for loops that I think are more compelling, and we'll talk about them when we talk about functions. But sometimes you just have to use loops. How can you know?

Basically -

You need to use a loop when the process of one iteration depends on the outcome of previous iteration/s.

The while loop we used in the last section is technically an example of a case where a loop is necessary, but let's look at another clearer ecological example: discrete population growth.

Let's use one common formulation of discrete logistic growth:

$$N_{t+1} = N_t * e^{r(\frac{K-N_t}{K})}$$

Where N is population size, r is growth rate, and K is carrying capacity.

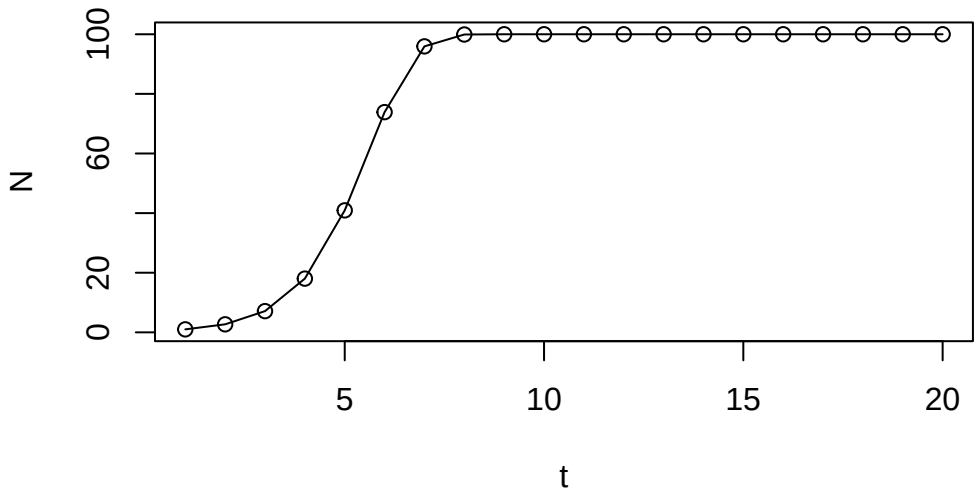
As you can see, that pesky N_t is involved in the calculation of N_{t+1}, or in other words, the process of calculation for one iteration/timestep depends on the outcome of the previous iteration/timestep. Thus we need to use a loop:

```
## set growth rate, r, and carrying capacity K
r <- 1
K <- 100
```

```r
## set number of timesteps, t, and preallocate population size vector, N
t <- 20
N <- vector(mode = "numeric", length = t)
N[1] <- 1

## loop through timesteps and calculate population size
## note: we are considering t = 1 as initial pop size
for (i in 1:(t - 1)) {
  N[i+1]  = N[i] * exp(r * ((K-N[i])/K))
}

## plot the output
plot(1:t, N, type = "o", xlab = "t")
```



This relatively simple loop is what powers this web app:

https://cwojan.shinyapps.io/discrete_logistic_growth/

Cool!

4 Functions

So we just went over the contexts when you need to use loops, but what else can you do when looping isn't necessary (i.e., the process of one iteration *does not* depend on previous iterations)?

You can write your own functions and use functionals to iterate them! We'll talk about functionals in the next section, but for now let's go over how to write functions and why.

4.0.1 How to Write Functions

You are all familiar with functions in general, we've been using them all throughout this workshop! However, the ones we've been using are premade for us. Now, we will write our own.

A function is made up of a couple of things: a name, one or more arguments, and body of code. You can make one with the following form:

```
new_function <- function(some_argument, some_default = "default"){
  output <- what_the_function_does(some_argument, default)
  return(output)
}
```

We start by providing our function name, "new_function", and we assign the output of the "function" function to that name as an object. (That sentence had a alot of "functions" in it…). Within the parentheses of the "function" function we can provide what arguments we want our new_function to have. We can set a default value for an argument by setting an argument = to some value or data structure. Finally, we provide what the function does in the brackets after the end parentheses. We can set what new_function's output will be with the return() statement.

Now let's do a working example, using our fake plant data that we used with a for loop earlier. Here's the code to generate it again:

```r
## create a data frame with 60 total plants and
## -two treatment levels, control and drought
## -plant heights, masses, and seed masses sampling from a normal distribution
plant_data <- data.frame(treatment = rep(c("control","drought"), each = 30),
                         height_cm = rnorm(n = 60, mean = rep(c(12,8), each = 30), sd = 2),
                         dry_biomass_g = rnorm(60, mean = rep(c(50, 40), each = 30), sd = 6)
                         seed_mass_mg = rnorm(60, mean = rep(c(30, 20), each = 30), sd = 4))
```

Now let's create a function to do a t-test for a given dependent variable. But this time instead of creating a list of statistical output, let's make it output just a few numbers of interest: the effect size (mean difference in this case), the 95% confidence interval, and the p value.

(this can be done with functions in the broom package, but it is a simple task that's good for demonstration purposes).

Instead of jumping right into the function, I'm going to figure out how to pull out those values, and then make the function (for easier debugging)

```r
## let's imagine a function that takes three arguments:
## dat for the data frame, treatment for the categorical column,
## and measurement for the dependent variable of interest
## let's set them to play with:
dat <- plant_data
treatment <- "treatment"
measurement <- "height_cm"

## now let's put those into a t_test
test_result <- t.test(dat[,measurement] ~ dat[,treatment])

## we can pull out values with $, for example
test_result$p.value
```

```
[1] 5.985477e-09
```

```r
## so make it a dataframe
clean_result <- data.frame(mean_diff = diff(test_result$estimate),
                           ci_lwr = test_result$conf.int[1],
                           ci_upr = test_result$conf.int[2],
                           p_val = test_result$p.value)
clean_result
```

```
                        mean_diff   ci_lwr   ci_upr        p_val
mean in group drought -3.979658 2.816046 5.143269 5.985477e-09
```

That looks like what we want! There is a inaccurate row name, but we can ignore that because it won't really affect anything. Now let's write it as a function!

```
## let's name our function quick_t_test and give it our three arguments
quick_t_test <- function(dat, treatment, measurement){
  ## do t test
  test_result <- t.test(dat[,measurement] ~ dat[,treatment])
  ## pull out summary
  clean_result <- data.frame(mean_diff = diff(test_result$estimate),
                             ci_lwr = test_result$conf.int[1],
                             ci_upr = test_result$conf.int[2],
                             p_val = test_result$p.value)
  ## return the clean_result as output from the function
  return(clean_result)
}
```

Note: the "return" command isn't strictly necessary here, because custom R functions will default to returning the last object created. However, I like to be explicit, especially when you create multiple object in a function.

Now let's test out our function!

```
## use quick t test on our fake data
quick_t_test(dat = plant_data, treatment = "treatment", measurement = "height_cm")
```

```
                        mean_diff   ci_lwr   ci_upr        p_val
mean in group drought -3.979658 2.816046 5.143269 5.985477e-09
```

```
## try another variable
quick_t_test(dat = plant_data, treatment = "treatment", measurement = "dry_biomass_g")
```

```
                        mean_diff   ci_lwr   ci_upr        p_val
mean in group drought -11.89196 8.969179 14.81473 4.145386e-11
```

It works! In the next section, we will apply this function iteratively with functionals, and the true power of functions will be apparent.

One more thing to note about functions is the distinction between local and global variables. Within the body of the function, any variable you create is local to the function, so only exists

while the function is running. After it completes, you will only have access to whatever you've returned. Global variable are what exist outside of the function in your environment. You can refer to them inside a function, but it makes the function less versatile.

4.0.2 Benefits of Writing Functions

Functions can be an efficient way to write your code for a variety of reasons. As mentioned before, some people prefer iterating functions with functionals for processing speed reasons, but I don't think that is as important for us, and in many cases there may not be a speed difference. Instead, I think writing functions is helpful because:

1. They can be easier to read. Looking back at code with a complicated nested for loop can be tough to parse, but code organized into functions can be easier to get a handle on.
2. They can be faster to write. I find myself getting things done quicker when I use functions - loops seemed to take longer to write and troubleshoot.
3. They can be transportable. Once you've written a generic function, you can copy and paste it into any old script where it would be useful, while it would be harder to do that with a loop.
4. They can make organizing large projects easier. Often people will create one script with all the functions for a certain analysis, and then other scripts where the analysis is run and output produced. This can be easier to keep track of and lead to shorter scripts to scroll through.
5. They can make your code modular. You can make your script a system of components that work together, functions can call other functions, and bugs are usually isolated to individual components.

5 Functionals with purrr

Now that we've written a function, let's use it efficiently with functionals. Functionals are functions that take other functions as one of their arguments, and then can apply that function a number of times or to a range of values (much like a loop!). Base R has the "apply" family of functionals, including lapply and sapply, but I've found those difficult to learn because they have inconsistent syntax and behavior. Instead, we will use the "map" functionals from the "purrr" package. It's part of the tidyverse, so you likely already have it installed. If not, use this code:

```
install.packages("purrr")
```

Now, we can load the package, and make sure we have our fake plant data and quick_t_test function in our environment.

```
library(purrr)

plant_data <- data.frame(treatment = rep(c("control","drought"), each = 30),
                         height_cm = rnorm(n = 60, mean = rep(c(12,8), each = 30), sd = 2),
                         dry_biomass_g = rnorm(60, mean = rep(c(50, 40), each = 30), sd = 6)
                         seed_mass_mg = rnorm(60, mean = rep(c(30, 20), each = 30), sd = 4))

quick_t_test <- function(dat, treatment, measurement){
  ## do t test
  test_result <- t.test(dat[,measurement] ~ dat[,treatment])
  ## pull out summary
  clean_result <- data.frame(mean_diff = diff(test_result$estimate),
                             ci_lwr = test_result$conf.int[1],
                             ci_upr = test_result$conf.int[2],
                             p_val = test_result$p.value)
  ## return the clean_result as output from the function
  return(clean_result)
}
```

5.0.1 The Basic map Functional

Now let's use the map functional to iterate our quick_t_test function over each of our dependent variables. We simply provide it what we want to iterate over and then we provide the function we want it to do as arguments. Let's go over a simple example first, making vectors of randomly generated numbers of different lengths, 1 through 5.

```
map_results <- map(.x = 1:5, ## iterate over .x
                   .f = rnorm) ## and do .f

## we get a list of vectors
map_results
```

```
[[1]]
[1] 0.2226997

[[2]]
[1] -1.151857 -1.376171

[[3]]
[1]  0.09818845 -1.38530632  2.33209016

[[4]]
[1]  1.1318644  0.1850341  0.2060751 -1.0630056

[[5]]
[1]  0.11868107  0.59746616 -0.94843995 -0.82485666 -0.07456446
```

One thing that is implicitly going on here is that 1:5 is being plugged into the first argument of rnorm by default, which is the number of observations to generate. The mean and sd default to 0 and 1 respectively (as indicated in the help doc for rnorm). However, our function has three arguments with no defaults, so we will need to supply those arguments and values in the map functional to do a quick t test for each of three variables:

```
## quick t test each dependent variable
ttest_results <- map(.x = c("height_cm", "dry_biomass_g", "seed_mass_mg"),
                     .f = quick_t_test,
                     dat = plant_data, treatment = "treatment")

## show results
ttest_results
```

```
[[1]]
                      mean_diff   ci_lwr   ci_upr        p_val
mean in group drought -3.725711 2.773919 4.677503 1.163476e-10

[[2]]
                      mean_diff   ci_lwr  ci_upr       p_val
mean in group drought -11.40987 8.445345 14.3744 2.50065e-10

[[3]]
                      mean_diff  ci_lwr   ci_upr        p_val
mean in group drought -9.473533 7.34689 11.60018 2.311656e-12
```

However, this list isn't super nice to look at. We could bind the rows into a data frame (with list_rbind or dplyr::bind_rows), or we could just made a data frame in the first place with map_dfr, which will always output a data frame (well, a tibble). Note that map_dfr is technically "superseded" because it requires the dplyr package to be installed, but it still works fine if you do.

```
## quick t test each dependent variable
ttest_df <- map_dfr(.x = c("height_cm", "dry_biomass_g", "seed_mass_mg"),
                    .f = quick_t_test,
                    dat = plant_data, treatment = "treatment")

## add a label column
ttest_df$dep_var <- c("height_cm", "dry_biomass_g", "seed_mass_mg")

## show results
ttest_df
```

```
                          mean_diff   ci_lwr    ci_upr        p_val
mean in group drought...1  -3.725711 2.773919  4.677503 1.163476e-10
mean in group drought...2 -11.409872 8.445345 14.374398 2.500650e-10
mean in group drought...3  -9.473533 7.346890 11.600176 2.311656e-12
                                dep_var
mean in group drought...1     height_cm
mean in group drought...2 dry_biomass_g
mean in group drought...3  seed_mass_mg
```

Much nicer right? (just ignore those row names, we could remove them if we wanted). There are map functions that specifically output all sorts of data structures, like vectors of specific data types (e.g., map_chr for strings, map_int for integers). There is even a map functional

that provides no output but iteratively performs a function: walk(). You could use walk when you want to iteratively draw plots or save files.

One thing that is also helpful is that you can write a function within a map functional, if you just need to do something over and over again but don't need to use the function elsewhere. So we could have done this:

```
ttest_df2 <- map_dfr(.x = c("height_cm", "dry_biomass_g", "seed_mass_mg"),
                     .f = function(x){
                       ## do t test
                       test_result <- t.test(plant_data[,x] ~
                                             plant_data[,"treatment"])
                       ## pull out summary
                       clean_result <- data.frame(mean_diff = diff(test_result$estimate),
                                                  ci_lwr = test_result$conf.int[1],
                                                  ci_upr = test_result$conf.int[2],
                                                  p_val = test_result$p.value)
                       ## return the clean_result as output from the function
                       return(clean_result)
                      })
```

We could have also done this with our pre-made function to deal with the arguments more explicitly:

```
ttest_df3 <- map_dfr(.x = c("height_cm", "dry_biomass_g", "seed_mass_mg"),
                     .f = function(x){
                       quick_t_test(dat = plant_data, treatment = "treatment",
                                    measurement = x)
                     })
```

5.0.2 Iterating over Two Ranges with map2

You can also iterate over two ranges of values with map2 functionals, or many with pmap functionals. Let's try it with our data by calculating the covariance between each pair of our dependent variables (there are likely other functions for this, but again, this is just a demo):

```
## first let's find the pairs of our variables with combn
var_coms <- combn(x = c("height_cm", "dry_biomass_g", "seed_mass_mg"), m = 2)
var_coms
```

```
     [,1]            [,2]           [,3]
[1,] "height_cm"     "height_cm"    "dry_biomass_g"
[2,] "dry_biomass_g" "seed_mass_mg" "seed_mass_mg"
```

```
## we can use the two rows as our iterating ranges

## use map2_dbl to coerce to numeric vector
var_covs <- map2_dbl(.x = var_coms[1,], .y = var_coms[2,],
                     .f = function(x, y){ ## use a wrapper function to set the data frame
                      cov(plant_data[,x], plant_data[y])
                     })
var_covs
```

```
[1]  8.747415  7.834075 33.858710
```

Here we didn't just use "cov" because we wanted to pull the data from the plant_data data frame, so we used a "wrapper" function so that we could more precisely set the arguments of cov().

We can put all of our info together like so:

```
plant_covs <- data.frame(var1 = var_coms[1,],
                         var2 = var_coms[2,],
                         cov = var_covs)

plant_covs
```

```
           var1          var2       cov
1     height_cm dry_biomass_g  8.747415
2     height_cm  seed_mass_mg  7.834075
3 dry_biomass_g  seed_mass_mg 33.858710
```

We could have also plugged the map functional right into our data.frame construction. If you are familiar with dplyr and use mutate, you can mutate new columns using map as well!

5.0.3 Concluding Remarks (that sounds too formal...)

Since the above demonstrations are relatively simple, you may be skeptical of the advantages of writing custom functions to iterate with functionals like map. And if you prefer to use loops, just use loops! You can do a lot of cool stuff with for loops. However, I think it's always nice to add more techniques to your "coding toolbox," as you can solve more problems.

Particularly, as your code gets more complex and interconnected, and your project gets larger, functions and functionals can really come in handy for keeping things organized. For example, I have a simulation project where I have a function to generate simulated landscapes, functions

to look at those landscapes, a function to simulate animal movement over those landscapes, and a function that applies parasitic interactions with those animals (the animal movement function is called within this function). This way I can pinpoint where bugs might be easily and transfer my landscape generation or movement functions to other projects. Perhaps most importantly, I can use map to try generating landscapes and simulating movement under a range of different parameters. This general organizational concept could be applied to more analytical projects where you are aalyzing and plotting different datasets in different ways.

In the end, you have the most important opinion on how you code, so try using functionals and see how they can be best integrated into your coding!

If you'd like to learn more than this shallow introduction, check this link out:

https://purrr.tidyverse.org/

6 Parallel Computing with map

If you want to go *really fast*, you can improve the performance of map functionals by using the parallel computing versions of them. Basically you can split the iterations you are doing among different R sessions or processing cores to shorten the total time to finish them all.

To demonstrate, I want to first make a function that is slow to run. It takes a while to calculate the distance matrix of a set of coordinates, especially if there's a lot of them. So I will make a function that does just that (don't worry about what actually is being done here, just that it is slow):

```
## load purrr library for later
library(purrr)

## set the only argument to be the size of the coordinate system, with a default of 1000
## store output temporarily and then delete it to clear up memory
long_function <- function(size = 1000){
  temp <- dist(c(1:size, 1:size))
  rm(temp)
}

## runs without errors (defaults to size = 1000)
long_function()
```

Now let's try running this function multiple times with walk (which is a purrr functional that gives no output), and measure how much time it takes. There a fancy ways to do this with performance check packages, but a simple way is to check the time before running, and then after running, and checking the difference (you need to run the code all at once for this to work though).

```
### run this block all at once
## save start time
start <- Sys.time()
## iterate long function for 1000 iterations of the same size value
walk(.x = rep(1000, 1000), .f = long_function)
## save end time
end <- Sys.time()
```

```
## calculate runtime
end - start
```

```
Time difference of 5.069347 secs
```

```
###
```

The time difference varies on my machine, but for me it's usually around or above 5 seconds. Obviously that's not terribly long, but good for demonstration purposes. If we instead chunk those 500 iterations to be performed in prralel by separate R sessions, we should be able to divide that runtime by as many sessions as you have.

For this we need the "furrr" package (hey it rhymes with purrr!), as well as it's dependency "future".

```
install.packages("furrr")

## if it doesn't install future automatically....
install.packages("future")
```

Then we need to set the "plan" to use multiple R sessions. This is basically telling the alternative map functional that we will use in the next step how to evaluate its process, sequentially or in parallel with multiple sessions (or cores, or even machines).

```
## load furrr
library(furrr)
```

```
Loading required package: future
```

```
## set plan to use 5 R sessions
plan(multisession, workers = 5)
```

Now we can use "future_walk" to run the function for 1000 iterations in ~1/5th the time!

```
### run this block all at once
## save start time
start <- Sys.time()
## iterate long function for 1000 iterations of the same size value
future_walk(.x = rep(1000, 1000), .f = long_function)
## save end time
end <- Sys.time()
## calculate runtime
end - start
```

```
Time difference of 2.123009 secs
```

```
###
```

This usually runs in less than 2 seconds! Wheee! Obviously in this case, a >50% reduction in time doesn't add up to much in seconds, but when your function takes hours, you can work much faster with furrr.

This is a very surface level demo, but should be good enough to speed up some of your code. There is a lot of depth to this topic that I'm not equipped to speak on, so do look here for more info:

https://furrr.futureverse.org/index.html

And the plan function in particular is good to understand:

https://www.rdocumentation.org/packages/future/versions/1.33.0/topics/plan

7 Vectorization

So the last thing I want to mention is the fastest way to iterate, and one that you have already done - vectorization. Many of R's base functions are written in C, which is much faster than R, so simple iterations can be done much faster. These functions are referred to as "vectorized", or you are basically performing vector math with them. Here is a simple example: let's find the 1 through 6th powers of 2:

```
## we could write a loop
for (i in 1:6){
  print(2^i)
}
```

```
[1] 2
[1] 4
[1] 8
[1] 16
[1] 32
[1] 64
```

```
## or we could use a functional
library(purrr)
map_int(.x = 1:6, .f = function(x){2^x})
```

```
[1]  2  4  8 16 32 64
```

```
## or we could simply use vector math; the "^" function is written in C, so...
2^(1:6)
```

```
[1]  2  4  8 16 32 64
```

The third solution (which is the fastest) may have been obvious to you, but sometimes it's not so obvious.

Consider simulating an animal that moves randomly around an infinite grid (or perhaps simulating Brownian motion of particles). If you wanted to track that animal/particles position at each of 100 timesteps, it seems like a for loop would be your best bet. The position of the animal at one time step depends in part on its position in the last time step, right? So let's show that:

```
## create a matrix to keep track of positions
animal_coords <- matrix(NA, nrow = 101, ncol = 2)
## start our animal at the origin of the infinite grid
animal_coords[1,] <- c(0,0)

## move it 100 times
for (i in 1:100){
  animal_coords[i+1,1] <- animal_coords[i,1] + sample(x = c(1,0,-1), size = 1)
  animal_coords[i+1,2] <- animal_coords[i,2] + sample(x = c(1,0,-1), size = 1)
}
```

So that works, and it's plenty fast for a single run of 100 timesteps, but increase the timesteps and simulate over some parameter space and it gets lengthy. Luckily, base R has a helpful vectorized function for this: cumsum. It calculates the cumulative sum of a vector at each index. So we can also write our random walk like this:

```
## cumulatively sum random movements for the x and y coordinate
x_coords <- cumsum(c(0, sample(c(1,0,-1), size = 100, replace = TRUE)))
y_coords <- cumsum(c(0, sample(c(1,0,-1), size = 100, replace = TRUE)))

## bind those coords
new_animal_coords <- cbind(x_coords, y_coords)
```

(Note: for random walks across higher dimension arrays, writing a function and iterating over each axis would be advisable).

That method would save you a lot of time if you were running tons of simulations! Basically, we are outsourcing our for loop to C with the cumsum function. When you can, outsource iterations to C! (Although it may take some time digging around base R functions to find what you need).

Part II

git, GitHub, Building Websites, and Copilot

8 Git and GitHub

8.1 Introduction

This page aims to get people started with using version control in their coding and data analysis with R, specifically using git (a computer application) and GitHub (a website).

By far the most useful resource for this aim is the following webpage from statistician Jenny Bryan:

Happy Git With R

On this page is a shorter summary of some of the key points based on specifically how I (Chris Wojan) work with git and GitHub.

8.1.1 Why?

When writing code, three things can be helpful:

1. Backing up your scripts
2. Sharing your code easily
3. Looking at previous versions of code in case something breaks

Using git allows you to save iterative versions of your code (and thus monitor your successive edits). Using GitHub allows you to back up your scripts online, and share your code easily with collaborators without attaching misc. files to emails.

8.2 Version Control with git

Our first step is to get git installed locally on our computers. Since git is designed primarily for computer programming applications, its installation is a bit different than many applications, and the steps vary among operating systems.

8.2.1 MacOS

If you use MacOS (like me), installing git is fairly straightforward, but there are a few ways to do it.

Note: if you think git might already be installed on your computer, you can check by opening the Terminal in RStudio (Tools -> Terminal) or through your OS (e.g. MacOS' Terminal application), and then entering the command:

```
which git
```

I personally have installed Homebrew, which is a tool that allows you to install useful applications and code packages. You can install Homebrew with the traditional Mac .pkg installation process from their GitHub Repository. Alternatively, you can open the Terminal application on your Mac and input the following command:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh
```

With Homebrew installed, you can simply open the MacOS Terminal application or Terminal in RStudio (Tools -> Terminal) and input the following command:

```
brew install git
```

Alternatively, Apple features git in its "xcode" set of command line tools, you can install these (including git) with this command in your Terminal:

```
xcode-select --install
```

8.2.2 Windows

I am personally less adept with git on Windows, but you should be able to download git through a typical installer from these pages:

Git For Windows

or

Windows git Download

Both of those should come with a git Bash Shell, where you can input command line git actions if needed. Windows doesn't have a simple shell/terminal situation like MacOS does unfortunately. However, you should be able to check if you have git installed in Command Prompt, VirtualShell, or something else (?) with

```
which git
```

8.2.3 Linux

If you are using Linux, you know more than me.

8.3 GitHub

While git can be used locally for version control, it is arguably most effective when used in conjunction with an online repository. GitHub is the most popular and well-documented.

You can register an account for free here:

https://github.com

The aforementioned Jenny Bryan recommends choosing a simple username that features your name somehow, and I do too as you'll likely be using this account in conjunction with your scientific publications.

In addition, the username you choose will be the prefix of your personal website if you choose to make one with GitHub.

Also, as a University of Minnesota student, you can get access to Github Education through this page:

GitHub Education

This comes with a variety of features, but the only one I've used is Copilot, which we'll go over in a later page.

With a GitHub account, you can make an unlimited number of "repositories", which are basically folders hierarchies for code (in truth, this is based on how git works without GitHub). These repositories can be private or public, and you can switch visibility after you've made one (e.g., making a repo public when submitting a manuscript). R Projects in RStudio are designed to match and work as git repositories.

8.4 Connecting in RStudio

Now you should have git installed on your computer, and GitHub account where you can back up and share your code. While you can use git in the Terminal/Shell to send files to GitHub, it is easier for beginners (and me) to use RStudio to connect things.

8.4.1 Configuring

There are also some handy R packages to make things easy that we can use. The first is "usethis", which has various utilities for managing coding projects in R.

```
install.packages("usethis")
```

One nifty function is the ability to configure your git installation through R, i.e. setting your name and GitHub account-related email:

```
library(usethis)
use_git_config(user.name = "Jane Doe", user.email = "jane@example.org")
```

I believe this is a necessary step to get things working, but I'm not absolutely positive. It's easy enough though!

Next, there is the very useful "gitcreds" package (this one will likely get periodic use, as you will see).

```
install.packages("gitcreds")
```

This will take us to how we can "log in" to GitHub through RStudio. To do this, we will need a Personal Access Token or PAT to authorize RStudio to do git actions. You can do this through usethis, but I think it is more intuitive to do it on the GitHub website.

To do so, when logged in on GitHub, click your profile picture in the upper right (probably a weird collection of pixels), and scroll down to Settings. Once there, scroll down to "Developer Settings" on the left sidebar. Click that and then Personal Access Tokens again on the left sidebar, and "Tokens (classic)" in the dropdown. There, you can click Generate New Token (again choose "classic").

When generating a new token, you'll be given a bevy of options, the first of which is a note. I usually just call it the month and the year.

Then you set how long until it expires. You can set it to never expire, which GitHub does not recommend, but I just go with 90 days myself (after which you will just repeat the process we are doing now).

You will also set the scope of the token. For our purposes click the header checkboxes for repo, workflow, and user.

Then click generate token, and you'll be given a long sequence of characters that you should copy to your clipboard, as you won't be seeing it again. If you close the window or something beforehand, you can just generate a new token though.

Now, with PAT copied, we can run the following code in R:

```
gitcreds::gitcreds_set()
```

It will then prompt you to set or replace your credentials. You can use your GitHub username, and the PAT you generated as the password.

Now you should be able to communicate from RStudio to GitHub!

8.4.2 Creating a Repo

To test out our connection, let's make a repository! The simplest way, and the way I usually do this is by making a repository on GitHub first.

To do so, be logged in to GitHub and click the green "New" button near repositories on the left of your dashboard page.

The page it takes you to will have a few options. The most important is the repository name. For this purpose it doesn't really matter, as you can delete this repository after testing.

You can also set if it is private or public, add a readme, select a gitignore template (more on this later), and set the license, but you don't need to bother with those for now.

When you hit create, you'll be taken to a new repo page. If you did not create a Readme, it will just be a setup page where you can copy the HTTPS link easily (which you should do!).

Alternatively, with a readme file, you'll have a more traditional repo page, where you can click the green "Code" button to copy the HTTPS link to the repo.

In either case, the link should look like "https://github.com/your-username/your-repo.git".

With this copied, head to RStudio, and click File > New Project, or New Project from the project tab in the upper right.

In the dialog box that pops up, select "Version Control" then "git", and paste in your repository link in the "Repository URL" field. Take note of the directory where this project is being saved.

With the new project opened, you can edit the Readme.md file, or create a new .R file, and save these change locally in RStudio. Do something like this.

You should notice that in the upper right pane of RStudio, there is a git tab. There, any new saved change you have made should show up. You can click the check box under "Staged" to add these changes to a potential "commit". A commit is a locally stored snapshot of a repository that you can create whenever you choose to.

With your changes staged, click the Commit button in the git tab, and you'll see a dialog box where you can add a message for the commit, summarizing what changes have been made. You can also track what changes you've done. Then you can click commit and close the window.

Now you can "push" these changes to your GitHub repository. Remember, commits are local to wherever you are, but can be pushed or pulled elsewhere. You can push your commits from your computer to GitHub, and you can pull commits made elsewhere from GitHub to your computer.

In the git tab, there are "Push" and "Pull" buttons. If you are ever working collaboratively, it can be good practice to pull before you push everytime. If you are just backing up and sharing your own code, you can just push from RStudio to GitHub when needed. Click "Push" to send your commit to GitHub.

If you refresh the repository's page on GitHub, you should see the files you made or edited in RStudio there. Woohoo!

You can also make edits to things on GitHub and then "Pull" them to Rstudio in the git tab, but that isn't a common thing to do at least in my experience. Pulling will mostly be relevant if you are working on a git repo with others or by yourself on multiple computers.

9 Organizing Repositories

9.1 Introduction

In the previous section, we went over getting git installed and connecting with GitHub. As we did that, we introduced the idea of git repositories. Here, we will go into a bit more detail about them.

Repositories can be thought of as a collection of related files, much like a folder or directory on your computer (with or without subfolders). As we have discussed, with git we can save versions of this collection of files at different time points. This is like the the version history on a Google Doc, but beyond the scope of one document to a whole collection of files.

Repositories can take many forms (this website you are reading is one!). In Ecology, Evolution, and Behavior, they are often R Projects, or projects based in Python or MatLAB, or perhaps other languages, or a mixture.

While a git repository at its most basic is simply a local folder, it makes most sense in our context to talk about them as a shared entity between your local computer and the GitHub website.

9.2 Repository Structure

On this page, we'll talk about the structure of a repository as an R Project built with RStudio.

If you are unfamiliar with R Projects in general, they are a useful way to organize code and data related to a given research project. At the very least, one consists of a .Rproj file with the project name inside of a folder on your computer named the same thing. In the folder with the .Rproj file, you can create subfolders, R scripts, and other relevant files.

One of the great thing about an R Project is its directory detection - when you have an R Project open, the working directory that it will read files from and write files to is the same as the folder that the .Rproj file is in. IF you want to read or write to a subfolder, you can simply add "/subfolder_name/" to your read or write function's name or directory argument.

On your computer, a research project's repository might often look something like this:

- project_name
  - project_name.Rproj
  - /code
    - script.R
    - other_script.R
  - /data
    - data.csv
  - /figures
    - figure1.tiff
    - figure2.tiff
  - README.md
  - .gitignore

In such a repo, the scripts would likely read data from the "/data" folder, and write figures to the "/figures" folder.

Your project might also feature a .Rhistory file that logs the R code that you run into a text file.

There are two other important files here, the first is the .gitignore file which we will go into now, and the readme file which we'll talk about afterwards.

9.3 .gitignore

Now you may have noticed that I referred to the above R Project / Repository as *on your computer*. This is because the files in a project on your computer will generally be more inclusive than those on GitHub. GitHub is not meant as a data storage solution, so it is best to limit what is stored on GitHub to only text files - i.e. scripts of code or markdown files.

This asymmetry is where the .gitignore file comes in. It is a simple text file the lists the files and folders in your local repository that you don't want to "push" to GitHub. This file is read by git, which ignores those files for commits and pushes in the git tab of RStudio (or elsewhere, like in the Terminal).

You can edit the .gitignore file in RStudio by opening it through the Files tab in the lower right pane. It will likely have some things already listed if you set a template .gitignore when creating your repo, or from when you created the R project itself. These include:

- .Rproj.user
- .Rhistory
- .RData
- .Ruserdata

You can add files and folders to this document on new lines. I general recommend that you add any subfolders of data or figures, although there may be some small output .csv files that you may want on GitHub in certain situations.

Note: MacOS will create a .DS_Store file to your project/repo if you open it in Finder, so you may as well add ".DS_Store" to your .gitignore file.

Here is an example repository for a manuscript I have in review:

https://github.com/cwojan/spatially_aggregated_parasite

On the repo page, you can see there is an .Rproj file, a set of .qmd files (these are Quarto Markdown files, which is an alternative way to write R code with more documentation), and a .R script. There is also a README.md, a .gitignore, and an /old_misc_files folder.

You can click the .gitignore file and see that I have ignored three folders: /data, /sim_output, and /figures.

/data and /sim_ouput contain .rds files and .csv files featuring simulation data, and some of them are rather large. /figures includes figures in high quality tiff format, so these are also large files. These folders are of course located on my computer in my R Project folder.

You can of course backup these types of folders that you are not sending to GitHub using Google Drive or what have you, but note that if you backup a git repo on Google Drive, your "Recent Files" tab will be filled with weirdly named config files when you do a lot of commits.

9.4 Readme files

The other important file that often goes along with repositories is the Readme file. This is where you can decide how to describe the project to anyone who might need to know about it, including what your aim is, what certain scripts do, etc.

In the example repo above, I provide a broad overview of what the researrch project is, and how someone would go about recreating my simulations with the code provided.

I then used GitHub Copilot to generate a more detailed description, which I'm not sure on the usefulness of.

In any case, a Readme file is always a good idea for repositories that are going to be shared with collaborators or are part of a manuscript publication.

10 Building GitHub Websites with RStudio

10.1 Introduction

One really cool thing you can do with a GitHub repository is turn it into a "GitHub Page" on the web, and RStudio provides built-in, intuitive ways to do so.

Before we get started, it is important to be familiar with "Quarto Markdown Files", which were referenced in the last section. These are similar to R Markdown files which you may be familiar with, but are compatible with other languages (like Python), and are generally more supported by modern documentation and interfaces these days.

With Quarto documents, you can document and organize your code in ways that can be more clunky with pure comments. In addition, you can render these files to HTML and PDF formats, which allows you to share descriptive text with embedded code and figures. It also allows you to create HTML pages that can be published to the web without actually knowing HTML, through RStudio.

We will talk about two ways to do that: making your own personal webpage with your github account, and publishing a web book. The website you are reading is a web book built with Quarto documents in RStudio and published through GitHub Pages.

Before we get into it, the Quarto file format is extremely well-documented in detail here:

https://quarto.org/

You can also find more info on GitHub Pages here:

https://pages.github.com/

What follows is a brief overview of how I (Chris Wojan) have approached using Quarto to publish websites.

10.2 Personal Webpages

While you can make websites for an unlimited number of your project repositories, with your free GitHub Account you can make one special website that is simply *username*.github.io. Here is mine:

https://cwojan.github.io/

And here is the repository that underlies it:

https://github.com/cwojan/cwojan.github.io

10.2.1 Setup

The key thing here is that you create a repository on GitHub that is named "[your_username_here].github.io". The simplest way to do this is to make a repository on GitHub itself and name it that, then "clone" it to an RStudio Project on your computer (as we did in an earlier section). However, this will lead to you needing to fill your repository with the requisite files for website publishing with Quarto.

An alternative method, the one I used, is to make a new R Project in RStudio with a Quarto Website template. To do so in RStudio, click New Project > New Directory > Quarto Website. Then set the directory you want to save your site in. In "Directory name", type "[your_username_here].github.io" (with your respective username as the prefix). Also, check "Create a git repository".

The project that is then created has the basic building blocks of a Quarto site:

- _quarto.yaml
  - This file is the backbone and configuration of your site
- styles.css
  - This file will alter the appearance of your site, but will probably remain blank if you don't edit it yourself with your own CSS skills.
- index.qmd
  - this is a Quarto Document that serves as the homepage of your site
- about.qmd
  - this is a second page of your website, that can be "about" you
- .gitignore
- [project_name].Rproj

Before we dig into building your site, let's make sure we can connect it to GitHub, since right now it only exists on your computer.

The simplest way to do this is with the aforementioned "usethis" package in R. Otherwise you'll need to use shell/terminal commands.

In your new project, simply run the following in your R console:

```
usethis::use_github()
```

If your git credentials are set as we did in prior sections, this will create a new repo of the same name as your local repo on GitHub and connect them together. If you named it correctly, this should then work for your special GitHub URL.

To test things out, make some edits to the index.qmd or about.qmd, save and commit them, then push them. Check to make sure they show up on your repo on GitHub.

Now there's a few steps to do before publishing (in addition to of course filling out your website):

First, the simplest publishing method is to publish from a subfolder of rendered HTML in your repository. So first create a subfolder in your repo called "docs". Then add "output-dir: docs" to your _quarto.yaml file, indented under the "project:" heading, after "type: website".

Second, GitHub Pages uses Jekyll as a tool to build sites by default, but since we are using Quarto, we need to tell GitHub not to use Jekyll. To do this we just need to create a blank file called ".nojekyll".

In the Terminal tab of the lower left pane in RStudio, enter

MacOS (and probably Linux too):

```
touch .nojekyll
```

Windows:

```
copy NUL .nojekyll
```

These commands create that file.

Finally, with the .nojekyll file pushed to the repo on GitHub, you will want to go into the repo's Settings (available along the top of the repo's base page), click "Pages", and make sure that the "Build and deployment" section is set to "Source: Deploy from a branch" and "Branch: main /docs". If you build you site now, there won't be anything because your docs folder is empty. You will need to render HTML files to that folder in RStudio before you have a site to publish.

10.2.2 Filling in and Editing your Website

Making your website will begin and revolve around the _quarto.yaml file. You can use it to add pages and change the look and theme of your site.

In that file, there should be a "website:" header, under which there are indented sections relating to the site's title, navigation method, and the pages. You can add pages by creating a new line under the "- about.qmd" line, and then creating a file with that name and editing it.

There is also a "format:" header, under which you can edit the look of the site. The easiest thing to do is edit the text after "theme:". Quarto works with the Bootswatch set of themes you can see here:

https://bootswatch.com/

The default is "cosmo", but you can switch to any that are shown. I use "minty" for my website =).

You can also edit beyond those themes within the yaml or in a .css file, depending on your skills and desire to tweak things.

More details on how to edit how your website works can be found here:

https://quarto.org/docs/websites/

And more info on HTML construction and theming can be found here:

https://quarto.org/docs/output-formats/html-basics.html

https://quarto.org/docs/output-formats/html-themes.html

Feel free to look at my website's repo for very basic examples!

10.2.3 Publishing Your Site

To publish your site, you'll first need to render your collection of Quarto documents to HTML/CSS/JavaScript, which is as easy as a click of a button!

Under the "Build" tab in the upper right pane of RStudio, there should be a "Render Website" button. Click that, and RStudio will start filling the "docs" folder in your repo with rendered website files.

Once that is done, you'll want to push your changes to GitHub. However, there may be many files in your git tab to stage before committing. To save time, you can use the following command in the Terminal to ready all changed files in your repo for committing:

```
git add .
```

"git add" is the shell command for staging files, and "." basically refers to all files.

All the files in your git tab should now have Staged checked, so you can go ahead and click Commit, and add a message like "initial site build" or something like that. Once committed, Push your commit to GitHub.

Then, on GitHub, navigate to the Pages subtab in your repo's settings again, and click the button to publish your site (assuming it knows to publish from the docs folder).

With that, a hyperlink should show up taking you to your site. Nice!

When you make changes to your site's Quarto files or yaml, you'll need to rebuild it before pushing it, but then it should automatically republish those changes in a few minutes.

Note: there are many other ways to publish websites with RStudio and Quarto, but this is just one simple way.

10.3 Other Websites, Web Books

While personal websites are a common desire for folks, others might also consider publishing other repositories.

You can make a GitHub page for any repository you have, so you could make a website for a research project or manuscript with R code built-in to show tables and figures. To do this, you could follow similar steps as for building your personal webpage (create a Quarto Website R Project in RStudio, and then create a connected repo on GitHub with usethis), but simply name it whatever you wish instead of your specific username plus github.io. The setup and publishing steps are virtually identical.

You can also publish web books for educational purposes, like this one. In this case, you can start an R Project in RStudio with the "Quarto Book" template. This comes with many nifty features, like a well organized sidebar of chapters, and cross-referencing abilities. This website is a Quarto Book, and here is another example that I have made for Cedar Creek interns doing research projects:

https://cwojan.github.io/ccesr_intern_hub/

There is a ton of info on writing web book with Quarto here:

https://quarto.org/docs/books/

If you are interested in contributing to this book with helpful computational/quantitative resources for EEB folks, feel free to reach out to me (Chris Wojan) and I can share the repo with you so we can collaborate!

11 GitHub Copilot

11.1 Introduction

When we went over signing up for a GitHub account, I mentioned how you can get access to GitHub Education by verifying your student status at UMN. There are many features to GitHub Education, most of which I have not used. The one I have used, and found somewhat useful, is GitHub Copilot. Copilot is a Large Language Model (commonly referred to as AI) built on the GPT framework specifically fine-tuned for computer programming. You can use it to answer coding questions specific to your GitHub repositories, as well as interactively while you are coding to solve problems.

Of course, there are ethical concerns to AI usage. Environmentally, it takes a lot of energy to train these models (although new developments seem to be reducing that consumption…). Philosophically, there are unresolved questions regarding the morality and legality of how LLMs have used web materials, oftentimes copyrighted materials, to "learn". For many, these concerns are a strong deterrent from using AI, which is totally fair!

In the specific context of Copilot, I can't say what it's environmental footprint has been, but I can say that there are some settings you can use to mitigate some of the concerns regarding the web content used for code suggestions.

11.2 Setting up Copilot

In order to use Copilot, you'll need to first sign up for GitHub Education. Instructions can be found here:

https://docs.github.com/en/education/explore-the-benefits-of-teaching-and-learning-with-github-education/github-education-for-students/apply-to-github-education-as-a-student

You'll need to provide your UMN email address and some proof of enrollment. As I recall, UMN student ID cards don't have enough info on them, so I think I used an "Enrollment Verification" pdf downloaded from MyU (but maybe that was for Spotify…). Anyway, if you are reading this as part of a live workshop, this means you may have to wait a bit for verification before you can actually set up Copilot. Nevertheless, you can return to this site and set things up afterwards.

Once you have GitHub Education, you can follow these steps to set up Copilot with GitHub:

https://docs.github.com/en/enterprise-cloud@latest/copilot/managing-copilot/managing-copilot-as-an-individual-subscriber/managing-your-github-copilot-pro-subscription/getting-free-access-to-copilot-pro-as-a-student-teacher-or-maintainer

Basically, there should be a "Copilot" tab in your account settings where you can activate access.

When you activate, there will be several setting for you to choose. First, there is enabling where Copilot can be used. Personally, I have it set as enabled for GitHub.com, the CLI, and my IDE. Copilot in GitHub.com allows you to chat with it's chatbot functionality on the website, and you can ask it questions about specific repositories. The CLI is basically your shell or terminal, which you may or may not use. Finally, I believe you need to enable Copilot for your IDE in order to link it to your RStudio, but I am not 100% positive.

There are more settings that you'll want to consider. Perhaps the most important is this: "Suggestions matching public code (duplication detection filter)". With this enabled, Copilot will make code suggestions that are identical to code snippets from open-source projects housed on public GitHub repositories. However, these projects will have different licenses for use, and if you just use whatever code suggestions you get willy-nilly without proper attribution, you may be violating those licenses. Whenever you do accept one of these suggestions, Copilot will automatically add the license information to a log file, so you can figure out the ways to properly cite these repositories. Personally, I don't want to deal with that, so I just have this setting disabled. This means that Copilot primarily generates code suggestions based on programming language documentation, tutorials, and your own code (as far as I know).

There are also settings to select whether Copilot uses your coding practices for product functionality evaluation and/or training, which you can decide for yourself. Other settings refer to the use of Bing and other LLMs with Copilot, which don't seem strictly necessary to me for many use cases.

With these settings saved, Copilot should be activated for your GitHub account.

11.3 Copilot in RStudio

The most useful function of Copilot that I have found is using it while I write R code in RStudio.

Instructions on how to set this up can be found here:

https://docs.posit.co/ide/user/ide/guide/tools/copilot.html

The broad strokes are this:

1. Navigate to Tools > Global Options > Copilot in RStudio

2. Enable Copilot
3. Sign in and verify with your GitHub credentials

Afterwards, Copilot should be connected to your RStudio installation.

Copilot in RStudio works in two ways. First it will provide tailored code suggestions as you code. Second, it can answer questions within the script editor.

For the first application, I have found it incredibly useful, but in order to get good suggestions, documenting your script with descriptive comments is key. First, having a broad goal outlined in ## comments at the top of the script is very helpful. Second, writing ## comments before each section of code describing what you are trying to do and which functions you are using to do it will provide Copilot with the context to generate fairly detailed code for you. As you move through your script in this way, Copilot will pick up on the context and often provide suggestions that match the way you have been coding so far. Whenever Copilot gives you a suggestion, just hit the Tab key to accept it.

Asking questions in the script editor has not been very useful to me, but your mileage may vary. Use the following format typed into your .R or .qmd file: # q: *[question]*? and hit enter. Copilot will respond with a # a: answer. Here it had no interest in my small talk:

```
# q: what's up copilot?
# a: I'm just here to help you write code. I'm not a copilot, I'm a code copilot.
```

11.4 Copilot on GitHub

You can also use Copilot on the GitHub website for coding questions in a more intuitive way than asking questions within a script.

On almost every page on GitHub when you are logged in, there should be an icon near the top right that looks like a little face wearing a old-timey airplane pilot cap. If you click that, it will open a chat subwindow on that page.

The nice thing about this functionality is that Copilot can read your repositories, so you don't have to copy and paste certain code to ask specific questions.

I used Copilot on GitHub.com to generate a detailed overview of one of my repositories for a README file.

There are probably many more ways to use it, but I haven't played around with it much yet.