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.

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
```

```
## 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:

```
## 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 =
                         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:

```
## 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 = 6.7383, df = 57.118, p-value = 8.596e-09
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
 2.474106 4.566219
sample estimates:
mean in group control mean in group drought
            11.415920              7.895758


$dry_biomass_g

    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 5.1325, df = 56.384, p-value = 3.669e-06
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
  5.112431 11.656355
sample estimates:
mean in group control mean in group drought
             46.93341              38.54901


$seed_mass_mg

    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 10.495, df = 57.784, p-value = 5.266e-15
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
  8.418132 12.386737
sample estimates:
mean in group control mean in group drought
             30.17007              19.76764
```

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 = 6.7383, df = 57.118, p-value = 8.596e-09
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
 2.474106 4.566219
sample estimates:
mean in group control mean in group drought
            11.415920              7.895758
```

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
```

```r
## 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:

```r
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.

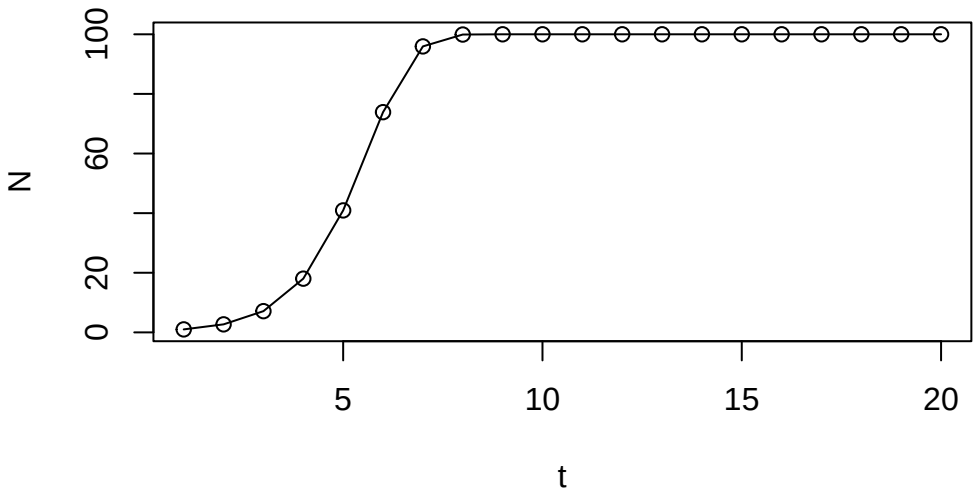
As you can see, that pesky N_t is involved in the calculation of N_{t+1}, or in other words, the process of calculation for one iteration/timestep depends on the outcome of the previous iteration/timestep. Thus we need to use a loop:

```
## 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:

```
## 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 =
                         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)

```
## 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] 1.779708e-12
```

```
## 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 -4.340184 3.368791 5.311577 1.779708e-12
```

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 -4.340184 3.368791 5.311577 1.779708e-12
```

```
## 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 -8.961865 5.632837 12.29089 1.386989e-06
```

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 =
                         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] 0.07932791

[[2]]
[1]  1.360347 -2.436044

[[3]]
[1]  0.1198042 -2.4063339 -0.9247977

[[4]]
[1] -0.30962750 -0.79319497  0.19740748 -0.04044216

[[5]]
[1] -0.8355214 -1.0432741 -0.8747075 -1.3341813  0.2239067
```

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 -4.055065 3.144971 4.96516 4.80569e-12

[[2]]
                      mean_diff   ci_lwr   ci_upr        p_val
mean in group drought -7.919656 4.785241 11.05407 4.801888e-06

[[3]]
                      mean_diff   ci_lwr   ci_upr       p_val
mean in group drought -8.687122 6.373927 11.00032 3.96523e-10
```

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")

## show results
ttest_df
```

```
                         mean_diff   ci_lwr   ci_upr        p_val
mean in group drought...1 -4.055065 3.144971  4.96516 4.805690e-12
mean in group drought...2 -7.919656 4.785241 11.05407 4.801888e-06
mean in group drought...3 -8.687122 6.373927 11.00032 3.965230e-10
```

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 withing 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)
                      })
```

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]  6.262796  6.956641 26.791533
```

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  6.262796
2    height_cm  seed_mass_mg  6.956641
3 dry_biomass_g  seed_mass_mg 26.791533
```

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 4.983276 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 2.103791 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(i^2)
}
```

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

```
## 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).