UMN EEB Quant/Comp Repo

Table of contents

Hello! **3**

I Iteration: Loops, Functions, and the purr package **4**

1 Intro to Iteration **5**

2 Types of Loops **6**

2.0.1 for loops 6

2.0.2 while loops 9

2.0.3 repeat loops 11

3 When to Loop? **12**

4 Functions **14**

5 Functionals with purr **15**

6 Parallel Computing with map **16**

7 Vectorization **17**

Hello!

This site/book will serve as a repository for resources related to computational and quantitative workshop 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, 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. Benefits of functions
    2. How to write 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 =
                         seed_mass_g = 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_g")

## 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 = 8.1472, df = 56.101, p-value = 4.374e-11
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
 3.058511 5.052860
sample estimates:
mean in group control mean in group drought
            11.928681              7.872996


$dry_biomass_g

    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 5.7138, df = 57.104, p-value = 4.191e-07
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
  6.868612 14.280137
sample estimates:
mean in group control mean in group drought
             49.03588              38.46151


$seed_mass_g

    Welch Two Sample t-test

data:  plant_data[, i] by plant_data[, "treatment"]
t = 10.054, df = 57.947, p-value = 2.565e-14
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
  7.753374 11.608154
sample estimates:
mean in group control mean in group drought
             30.18555              20.50478
```

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 = 8.1472, df = 56.101, p-value = 4.374e-11
alternative hypothesis: true difference in means between group control and group drought is n
95 percent confidence interval:
 3.058511 5.052860
sample estimates:
mean in group control mean in group drought
            11.928681              7.872996
```

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.

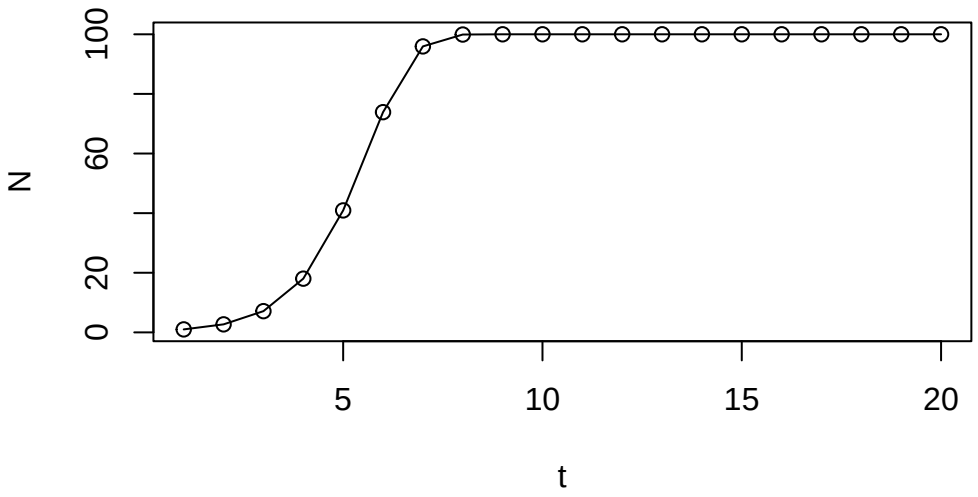
As you can see, that pesky N_t is involved in the calculation of N_{t+1}, or in other words, the process of calculation for one iteration/timestep depends on the outcome of the previous iteration/timestep. Thus we need to use a loop:

```
## 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

5 Functionals with purr

6 Parallel Computing with map

7 Vectorization