DEMO - `for` loops for fish

ER Deyle

Fall 2023; Marine Semester Block 3

`for` Loops, Again!

As we already discussed, loops are an important part of programming. They provide a structure to repeat a calculation many times. This is something computers are good at, so it's a good thing to learn how to ask the computer to do! There is more than one syntax for looping like this; we're starting with one that is basically ubiquitous across programming languages, the `for` loop. The basic syntax of a `for` loop is something like:

for every value of ITERATOR in VALUE_SET
do this code
end

In R, the syntax is

```
for(ITERATOR in VALUE_SET){
  YOUR_CODE_HERE
}
```

In the first lab, we use a `for` loop to repeat the same randomized process many times. In this case, the calculation every time through the loop is unchanged, aside from the result of the random number generator (RNG). The only variable that was changed was the vector of outputs, which were filled in with a new estimated total population size every time through the loop.

```
N_ <- 5000
m_ <- 500
K_ <- 500

v_N_estimate <- vector(length=100)
for(iter_i in 1:length(v_N_estimate)){

  fish_vector <- vector(length=N_)

  individuals_1 <- sample(N_,m_)
  fish_vector[individuals_1] <- 1

  individuals_2 <- sample(N_,K_)
  k_i <- sum(fish_vector[individuals_2])

  v_N_estimate[iter_i] <- m_*K_/k_i
}
```

This is a fairly simple piece of code, but the Lab Assignment asked questions that required further repetation, varying things like the fraction of the total population sampled each in round (`m` and `K`). The code we need is the same, just with changes in the parameters. This is where writing custom functions can help make organized, modular code. We can create a little module that does the necessary calculation for an aribitrary set of parameters.

Similarly, in the next lab, we're going to be using loops to simulate the passage of time and changes happening in every step. Writing a function that calculates that change at each step given the current variables will be very useful!

Custom functions in R

The syntax for writing your own function in R is as follows:

```
NAME_FOR_FUNCTION <- function(INPUTS){
  CALCULATIONS
  return(OUTPUT)
}
```

Just like with a `for` loop, the chunk of code to execute is enclosed in curly braces. Now, there are some shortcuts you can take; I point them out here just so you're aware reading someone else's code.

- The use of `return()` is not strictly necessary. R will just return the value of the last calculation it did. However, this can create unintended behavior sometimes, and in my opinion at least makes the code a bit harder to read.
- If the function only needs to perform a single-line calculation, the curly braces aren't necessary.

Suppose we wanted to add 1 to a number. Over and over. We write a function that takes a number and returns that number plus 1. We can write this with the short-cuts above, since it's a simple calculations

```
f_bad_habits <- function(x) x+1
```

and check that it works

```
f_bad_habits(1)
```

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

However, we can take a few more lines and write something that builds good habits!

```
f_good_habits <- function(x){
  y = x+1
  return(y)
}
```

Now if we can ask the value of a few numbers.

```
f_good_habits(1)
```

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

```
f_good_habits(5)
```

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

```
f_good_habits(1.5)
```

```
## [1] 2.5
```

We can even ask it the value for a bunch of numbers at once*

```
f_good_habits(c(1,5,1.5))
```

```
## [1] 2.0 6.0 2.5
```

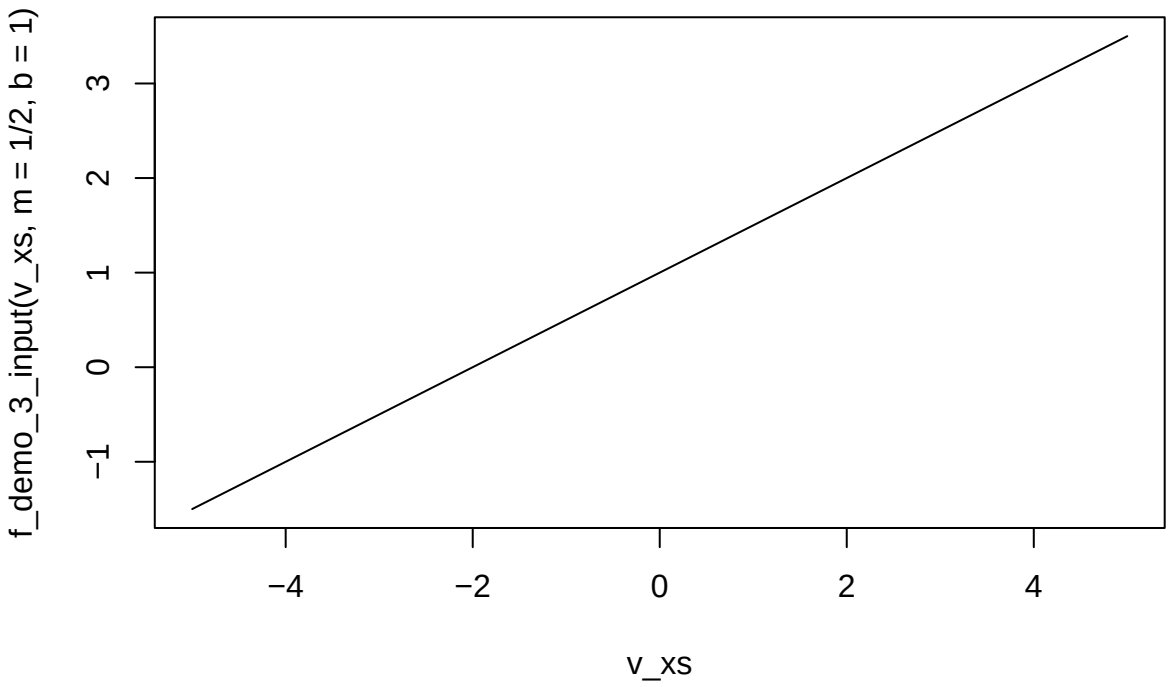
*This is actually something that isn't possible in other languages that is a double edged sword in R. Generally we don't have to tell R what kind of input we're giving a function. R will just try to make sense of it. In some languages, we would have had to specify if the input were going to be an integer, double precision (decimal), a vector, etc. And hey, the answer looks slightly different!

Just like built in R functions aren't limited to one argument, like `cor()`, custom functions are not limited to one argument. If we want to multiply x by a number, then add a number to it, we could write:

```
f_demo_3_input <- function(x,m,b){
  y = m*x+b
  return(y)
}
```

Look it's a line!

```
v_xs <- seq(-5,5,by=0.01)
plot(v_xs,f_demo_3_input(v_xs,m=1/2,b=1),type='l')
```



Functions and `for` Loops

Back to the example from Lab 1. There, we want to write a function that forms the *in silica* mark-recapture experiment a single time.

```
do_1_mark_recapture <- function(total_population,catch_sample_1,catch_sample_2){
  fish_vector <- vector(length=total_population)

  individuals_1 <- sample(total_population,catch_sample_1)
  fish_vector[individuals_1] <- 1

  individuals_2 <- sample(total_population,catch_sample_2)
  number_recatch <- sum(fish_vector[individuals_2])

  return(number_recatch)
}
```

A note on variable spaces and environments. When R is calculating "inside" a function, variables created there are in their own little world. Like this:

```
a_brand_new_variable <- 1
print(a_brand_new_variable)
```

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

```
remove(a_brand_new_variable)
print(a_brand_new_variable)
```

```
## Error in eval(expr, envir, enclos): object 'a_brand_new_variable' not found
```

```
f_print_a_brand_new_variable <- function(){
  a_brand_new_variable <- 1
  print(a_brand_new_variable)
}
f_print_a_brand_new_variable()
```

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

```
print(a_brand_new_variable)
```

```
## Error in eval(expr, envir, enclos): object 'a_brand_new_variable' not found
```

Some languages are very insistent on keeping function calculations completely constrained to the variable "environment" of the function. R is not. If R is evaluating a code expression and comes across a variable not in the function environment, it will go looking.

```
another_new_variable <- 2

f_print_another_new_variable <- function(){
  print(another_new_variable)
}

f_print_another_new_variable()
```

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

What happens if you change the value of the variable in the outside environment?

```
another_new_variable <- 3
f_print_a_brand_new_variable()
```

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

Even though `another_new_variable` was 2 when we defined the function, R doesn't actually care about it's value until we call the function.

Last bit of this tangent. What happens if you change the value of the variable INSIDE the environment?

```
last_new_variable <- 3

f_print_last_new_variable <- function(){
    last_new_variable <- 8
    print(last_new_variable)
}

f_print_last_new_variable()
```

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

In the function environment, the value of `last_new_variable changed`. However, did it change in the base environment?

```
print(last_new_variable)
```

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

It did not change! So, "What happens in the function, stays in the function." Except the output.

Was all this a bit confusing? That's fair. And all this confusion is avoidable by being deliberate with variable names. Variables like "n" and "N" might be tempting to use for lots of different things in a single set of code: number of individuals in a population, number of times to repeat a calculation. If you don't give unique names, you can end up with some very buggy code! IF you want to use very short variable names, my advice is to only use them as function input variables. However, even this can be tricky, as calling a function inside a function inside a function can cause R to go looking "back up stream"...

Population Dynamics

In the first `for` loop demonstration, we briefly thought about a population that's linearly increasing through time. The growth in the population is a constant at each time step. This is a very simple model of population change, but we can use it to illustrate a code structure that will work for more complicated models.

```
f_demo_model_1 <- function(N,c=2){
  delta_N <- c
  return(delta_N)
}
```

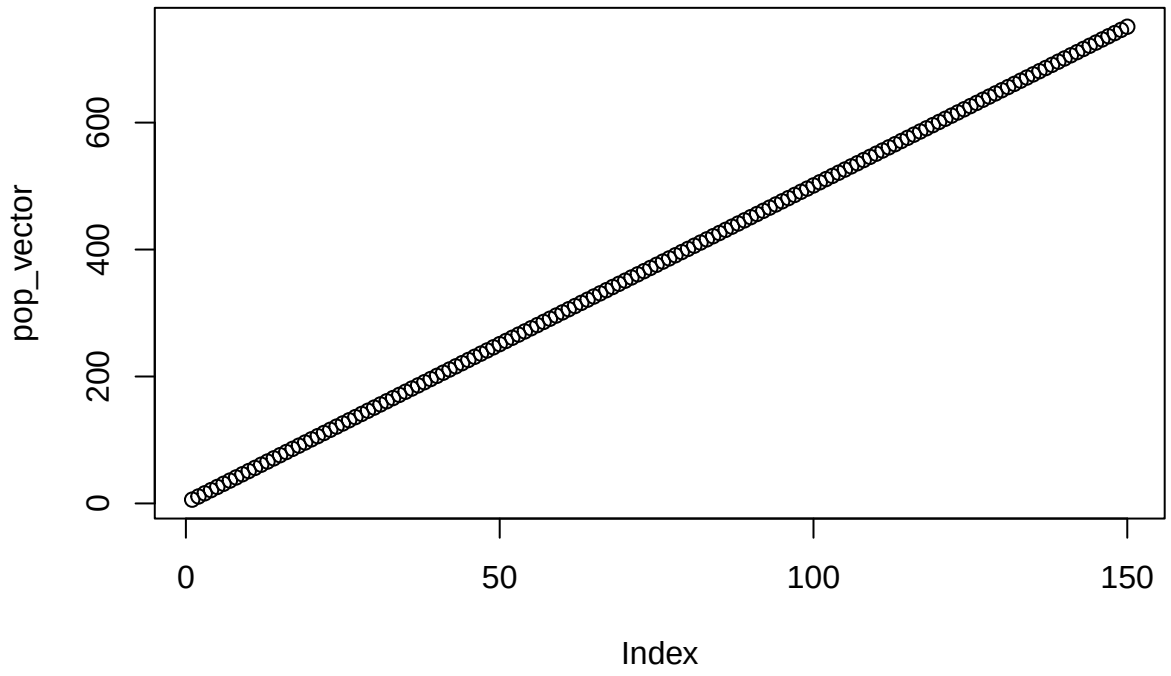
[We'll fill this in together]

```
total_steps <- 150
pop_vector <- numeric(total_steps)

N_0 <- 1

for(iter_time in 1:total_steps){
  N_0 <- f_demo_model_1(N_0,c=5) + N_0
  pop_vector[iter_time] <- N_0
}

plot(pop_vector)
```



Exercise 3

Create a `for` loop for cellular reproduction, where each cell produce two daughter cells in each time step. Begin with 1 cell, and allow for 10 generations. HINT: you can start with the code above.

```
f_cell_reproduction <- function(N_0){
  # return population growth rate for N_0 parent cells
  delta_N <- N_0
  return(delta_N)
}

total_generation <- 10
cell_vector <- numeric(total_generation)
N_0 <- 1

for(iter_generation in 1:total_generation){
  N_0 <- f_cell_reproduction(N_0) + N_0
  cell_vector[iter_generation] <- N_0
}

plot(cell_vector)
```