DEMO - `for` loops for fish

ER Deyle

Fall 2024; Marine Semester Block 3

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

(The above is part of the template text when you create a new Rmarkdown file in Rstudio. It's a Very Useful Link, so I've left it. To create your own template, go to File -> New File -> R Markdown ... There will be some options; a lot of these can actually be changed later. Start with a Document -> HTML).

`for` Loops

Loops are an important part of programing. 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
}
```

The code inside the loop can reference the current value of ITERATOR. This is often very useful. Below is a `for` loop that print the integers 1-10 in order. Here, the function `print()` just tells R to "echo" the argument in the output. If you do this in the Console, the integers will sequentially print in the console. When you do this in an Rmarkdown file, the output will print in the document when you generate it with "Knit". *For this demo, follow along by typing in Console.*

```
for(ITERATOR in 1:10){
  print(ITERATOR)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Exercise 1

What if we only wanted to print the numbers 5-10?

```
# CODE HERE
```

Variables and `for` Loops

By the end of this demo, we're going to be using `for` loops to iteratively simulate population dynamics. The model equations describe how the population changes at each time interval. Let's start with something very simple, where the change in the population is a constant each interval of time.

```
Population_size <- 0
Constant_growth <- 2

for(ITERATOR in 1:10){
  Population_size = Population_size + Constant_growth
}

print(Population_size)
```

```
## [1] 20
```

Exercise 2

Of course, since we are increasing by 2 a total of 10 times, we shouldn't be surprised the answer is 20... Multiplication is just iterated addition! Create a `for` loop to perform multiplication of two numbers `A_=5` and `B_=3` using only the addition operation, `+`.

```
# CODE HERE
```

Vectors and `for` Loops

Above, we were able to generate the size of our population after 10 iterations of constant growth. However, we're often going to be interested in the dynamics through time of a model, not just the end-result. Of course, we could put in a `print()` command to see each value in the output by taking the `print()` command at the end and putting it inside the `for` loop braces.

```
Population_size <- 0
Constant_growth <- 2

for(ITERATOR in 1:10){
  Population_size = Population_size + Constant_growth
  print(Population_size)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

Even better, however, is to store the entire sequence of values as a vector. To R, vectors are a sequence of numbers. We've already created some, like the integers 1-10.

```
My_vector <- 1:10
print(My_vector)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Individual vector values can be extracted with a syntax of indexing. To get the 5th value in the vector, for example, we write:

```
print(My_vector[5])
```

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

We can put values together in a vector by hand using the "combine" function, `c()`:

```
My_new_vector <- c(1,1,2,3,5,8,13,21)
print(My_new_vector)
```

```
## [1]  1  1  2  3  5  8 13 21
```

We can even use a second vector of index values to extract multiple values from a vector variable at once. I.e. instead of one at a time,

```
print(My_new_vector[1])
```

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

```
print(My_new_vector[3])
```

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

```
print(My_new_vector[5])
```

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

We can use `c()` to grab all three at once:

```
print(My_new_vector[c(1,3,5)])
```

```
## [1] 1 2 5
```

If we want to change an individual value, we can re-assign individual vector values using indexing on the left-hand-side.

```
My_new_vector[1] <- 100
print(My_new_vector)
```

```
## [1] 100   1   2   3   5   8  13  21
```

In R, if you try to assign a value to a vector position past the current "end", it will make the vector longer to accommodate the new value. This can be useful. Currently our vector is 8 elements long.

```
print(length(My_new_vector))
```

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

If we assign a value to the ninth element...

```
My_new_vector[9] <- 500
print(length(My_new_vector))
```

```
## [1] 9
```

and the vector is now

```
print(My_new_vector)
```

```
## [1] 100   1   2   3   5   8  13  21 500
```

Now that was just lengthening the vector by 1. What happens if we assign a value at index 20?

```
My_new_vector[20] <- 1000
print(length(My_new_vector))
```

```
## [1] 20
```

Now we have a 20-element vector. But we didn't tell it values for most of those new elements. What did R do?

```
print(My_new_vector)
```

```
##  [1]  100    1    2    3    5    8   13   21  500   NA   NA   NA   NA   NA   NA
## [16]   NA   NA   NA   NA 1000
```

R filled it with `NA`, which generally speaking is R's way of telling itself it doesn't know what's supposed to be there.

Back to our example. We were generating a sequence of population sizes under constant growth.

```
Population_size <- 0
Constant_growth <- 2

for(ITERATOR in 1:10){
  Population_size = Population_size + Constant_growth
  print(Population_size)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

To go from printing each loop to saving the whole sequence we need to add two things. (1) We need a place to put the values and (2) we need to replace the `print()` in the loop with a statement assigning the current value of `Population_size`. To create a blank vector, we use the `vector()` command. Type `?vector` to look at the syntax. When we use it with no argument, a vector of 0 length is created.

```
vector_Population_size <- vector()
Population_size <- 0
Consant_growth <- 2

for(ITERATOR in 1:10){
  Population_size = Population_size + Constant_growth
  vector_Population_size[ITERATOR] <- Population_size
}

print(vector_Population_size)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

Now we're at a good place to mention something you all might already know. There's a LOT of little choices you can make along the way in coding that can get you CORRECTLY to the same result. Sometimes choices have an effect on efficiency; sometimes they have an effect on readability; sometimes they have an effect on your own aesthetic feelings about your work!

The following code is mathematically equivalent to the above chunk.

```
vector_Population_size <- vector()
Population_size <- 0
Constant_growth <- 2

for(ITERATOR in 1:10){
  Population_size = Population_size + Constant_growth
  vector_Population_size <- c(vector_Population_size,Population_size)
}

print(vector_Population_size)
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

Of course there's also ways to INCORRECTLY get the same result.

```
print(c(2,4,6,8,10,12,14,16,18,20))
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

There's also ways where calculations that ARE equivalent in principal actually give you slightly different results because of the way computers work.

```
x <- 0.4-0.1
y <- 0.3

print(x==y)
```

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

```
print(x+0.1==y+0.1)
```

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

Tangent: What's going on there? Well, R is representing these decimals as long strings of 0s and 1s, and something in that representation is not quite right. When R calculates `0.4-0.1` it gets something very very very close to the correct answer, `0.3`, but it's slightly off.

```
print(x-y)
```

```
## [1] 5.551115e-17
```

If you're curious about the solution, look up the Help entry `?all.equal`

Exercise 3

Create a `for` loop for cell division, 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.

```
# CODE HERE
```