Directories, Reading in Data, and Essential Programming Skills - Functions, If-Statements, For Loops

Emma & Cole

Fall 2020

Essential programming skills

As biologists, many of us come to programming and coding by necessity, not choice. As such, we often jump right into learning how to manipulate data and do things with it such as statistics and visualizations. While in many cases that's ideal, it sometimes means we don't get as much of a solid grounding in some foundational programming skills that can be incredibly useful. There are two main skills that we will cover today, neither of them to exhaustion, but hopefully enough to get you what you need. These skills are writing **functions** and **for loops**

Functions

What is a function in R? Well, we've actually already met them! When we call `head()` on our dataframe, that is a function. To R, there is no difference between a function that comes pre-built and a function that we make ourselves.

Let's start by talking about how we define a function. We use the `function()` call, with two round brackets followed by a curly brace. What is inside the round brackets are the things that are needs to run the actual function. Inside the curly braces, this is where we put what happens to the things we specified in the round brackets during the function actually being executed. That looks like this:

```
my_function_name = function(argument_1, argument_2) {

  operations_do_something

  return_some_output
}
```

The last component is somewhat optional but it's called a return statement. A return statement allows us to dictate to our what is actually giving us as the output of the function.

When we write a function we first have to define the function in our environment before we can use it this means we need to name it, like above.Once we've named it, we can call it again by writing the function name, the arguments it requires inside the round brackets, and then running it either in a script, or in our console. It might be useful for a second, to pause and think about why we would bother writing a function in the first place. Recall from last week's lecture, the concept of using a function to build a house. What we were trying to do with this analogy is illustrate a time when you might want to perform a very similar set of operations multiple times but with different inputs. For example, when running a statistical test (some thing we won't talk about in this workshop series), we often want to run the same test on many data. To do this, it would be incredibly wasteful to write an individual function, or set of functions, to do this every time we wanted to run the same function each time. Instead, developers and software engineers have written a set of functions that exist in the R environment, for use by people like us all the time. If the function we want doesn't exist,

we can write it ourselves! You'll come to see that often the function you want doesn't exist, so it's handy to be able to write your own, even if it's very simple. All right let's get going.

Simple examples:

Here, we're going to write a few functions to do some basic calculations. These functions will allow us to get the whole concept of the thing.

```
# Example 1:
f1 <- function(x, y){ #your arguments, in this case x and y,
#  which can be any two numbers, go in the round brackets
  x+y #inside the curly braces is where your operations go
}

f1(x=3, y=4)
```

```
## [1] 7
```

```
f1(3,4)
```

```
## [1] 7
```

```
# Example 2:
f2 <- function(x,y){
  z1 <- 2*x+y
  z2 <- x + 2*y

  return(c(z1, z2)) #this optional, but very useful component tells R what to give us back
}

f2(5,6)
```

```
## [1] 16 17
```

Real data example:

Using our Penguins data, let's use our new knowledge of logical operators, to write a function that returns to us the number of rows in the dataframe that relate to a particular species. In our example, we're using the species 'Gentoo'.

```
dir = 'C:/Users/ematkinson/Documents/GitHub/Introduction-to-Programming-for-Biology/data'
setwd(dir)
penguins_data = read.csv('penguins.csv')


f3 <- function(species){
  r <- nrow(penguins_data[penguins_data$species==species,])
  return(r)
}

f3("Gentoo")
```

```
## [1] 124
```

Hopefully what we're doing here makes sense. To break it down, we're defining a function, called `f3`. We're saying that the function requires an argument, which doesn't have to be called 'species', that's just what we're going to use it for. We could put 'x' inside the brackets and it would work the same. Inside the curly braces, we're performing some operations. Here, we're subsetting the data (something you'll learn more about

in week 3), by telling R to only return the parts of the data where the value in the column `Species` matches what we gave the funciton as the lowercase 'species' argument. We then ask R to return the number of rows of that subsetted data to us.

An Aside About Function Definitions I, (Cole), would like to make a quick aside here. If any of you ever take an introductory programming course, and show the professor how you were taught in this workshop series to define a function, any good professor of programming would come to my office and light my computer on fire. What we've taught you here, is an EXTREMELY quick and dirty version of how to define and use functions. That is not to say that quick and dirty functions don't have a place, especially in the tool kit of a biologist, they are often VERY useful, and that's why we've taught you this! This is only to say that there is a whole section of study in computer science on how to effectively build and document functions. If you ever look at the documentation for a function in R by using the `?`, as I hope you do, then you will see an explanation of a) what the function does, b) how the function is supposed to act, c) who made the function, d) a few examples, and sometimes a variety of other things depending if it's base or a package etc. In different languages (including R), there are important conventions to follow when building functions that will be used by others in packages etc., and we have gone over none of them here, just the skeleton of what a function is. I only say this because I don't want you to have the impression that this is the be-all-end-all of functions. In our case, when building your own functions, it is **essential** to document what you are doing by commenting your code. This will allow you to come back to your code after some period of time, and understand what the function is doing. If you were interested in the proper way to make functions, how to document your functions, and how to make functions that extend far beyond the small examples that we've given you, I encourage you to dive into the world wide web (it has some great stuff and some not great stuff), consider taking one of the many Computer Science course options here at the University of Alberta, or perhaps consider taking or auditing a MOOC (Massive Open Online Course). You don't need to know all the theory behind how functions are made from a software development point of view, for our purposes it is definitely sufficient to know how to make a quick function to do something that you might want to do multiple times with different data. This is why we took the time to go a little bit into the world of functions. I just don't want you thinking this is the gold standard for howYou build functions, or how you document them.

/rant_over/

For Loops

The second very important tool that we want to teach you guys a bit about today, is called the 'for loop'. A for loop is a very common tool in many programming languages. *Note*: we won't go into this in detail but know that in R, there is technically a series of functions in the `apply()` family, that are somewhat preferable for use in the R programming environment. Essentially, they have been written to optimize the strengths of R which are not necessarily predisposed to running for loops, after all R is a statistical programming language, not primarily built for development (where for loops have their origins). However, these `apply()` functions are based on for loops. And while it is best practice to use `apply()` functions, they are much less intuitive, and harder to grasp unless you understand the underlying concept of a for loop. `apply()` functions are really just for loops under the hood, so we think it's important for you to learn them first. If in the future you go on to work in other programming languages such as Python, you will possibly have to use for loops substantially.

But, let's back up. What is the point of using a for loop? And what even is a for loop anyways?? Well, in a similar but distinct way to functions, these structures allow us to repeat the same operation on sets of information, usually data, and can be most effectively employed if you find yourself copying and pasting multiple lines of code and changing only one or maybe two things in each row. As their name suggests, for loops allow you to loop over things and perform the same operation multiple times. The way you can think about is to imagine you're verbally telling R what you want to do. Broadly, "*For* every thing in this series of things, *Loop* through each one, and do something each time".

For loops can sometimes be a bit tricky to wrap your head around at first, just like functions. However, when used in coordination with each other and another tool that we won't have time to cover called `if`

`statements` (Google if you're interested, you'll probably end up using them one day), they can be incredibly powerful. These three structures, `for loops`, `functions`, and `if statements` make up the vast majority of basic programming that you will ever do especially, with regards to programming with biological data. All right that's enough talking, let's actually make some loops!

Simple Examples:

Take a look at each of these examples in turn. Try to, without doing them on your own computer, interpret what their output will be, and why they will do what they do. These are some common things we might want to do to multiple numbers or sets of data, and rather than copying `print(number x)` over and over again, we can just do it in a loop!

```
# Example 1:
for(i in 1:10){
  print(i)
}
```

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

```
# Example 2:
a <- 0
for (i in 1:50){
  a <- a+1
}

## Challenge yourself! What would the output of a be here? Why is it that
a
```

```
## [1] 50
```

Note that in the above example, since we didn't specify otherwise, we're just re-assigning `a` over and over again, until we get to 50.

```
# Example 3:
a <- NULL
for (i in 1:50){
  a[i] <- i
}
a
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

In the above, we create `a` as a vector, and both create and assign locations in `a` sequentially, going from 1 to 50.

```
# Example 4:
a <- seq(1:25)
for(i in 1:length(a)){
  a[i] <- a[i]^2
```

```
}
a
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100 121 144 169 196 225 256 289 324 361
## [20] 400 441 484 529 576 625
```

Here, we've again made `a` into a vector, and we're simply re-assigning each position of the vector with the square of itself at each position.

Real data example:

Okay, let's do something with real data now. An important precursor here is that the `unique()` function returns all the unique values of a factor-type variable. (Some homework for you could be looking up what a factor means if you've forgotten from the first week!). Also a reminder, if you can't remember what a function does, you can always check by using the `?` in R. For example, I can ask for help on the `unique()` function like this:

```
?unique
```

```
## starting httpd help server ... done
```

While it doesn't print the help in a markdown document like this (sad), you can check this on your own machine.

Right, moving on! So, back to for loops but in the context of real data. For example, let's ask how many observations in our dataset are of each species. Here's the long way of doing that. Recall above, we defined `f3()` as a function which, given the species of interest, would give us the number of rows (observations). Okay, here we go:

```
f3("Gentoo")
```

```
## [1] 124
```

```
f3("Adelie")
```

```
## [1] 152
```

```
f3("Chinstrap")
```

```
## [1] 68
```

And lo and behold, it does the thing we asked. This doesn't look like too much work does it? Well, you're right it's not. However, try doing this for a dataframe with 200-odd species, and it will become super un-fun super quickly.

Not to fear! There's a better way, and it's looping, in this case, with the `unique()` function to help us out.

```
for (i in unique(penguins_data$species)){
  rows <- f3(i)
  print(rows)
}
```

```
## [1] 152
## [1] 124
## [1] 68
```

We can even make it $_{\text{fancy}}$ and ask it to print us a full real sentence.

```
for (i in unique(penguins_data$species)){
  rows <- f3(i)
  print(paste("The number of observations for ", i,"is ", rows, sep=""))
}
```

```
## [1] "The number of observations for Adelieis 152"
## [1] "The number of observations for Gentoois 124"
## [1] "The number of observations for Chinstrapis 68"
```

Fun!

Recap of session 2:

We talked about a lot this week!

1. How to read in data using the `read.csv()` function
2. We discussed logical operators and how to index columns
3. We went over **functions** and **for loops**, why they're important, and how we use them

Whew, that was a lot! Don't feel worried if you don't fully understand the functions and for loops yet. You'll need to use them (yes, actually practice!) to fully understand them and for it all to make sense. The tough thing about programming is it's very much something that gets easier as you do it more, and if you stop doing it for a long period of time, it's harder to quickly remember how to do things. However, hopefully these markdown documents can serve as a reminder if you step away from R for a while but come backt to it.