R Primer

BIOS 20172

Monday 2023-03-27

In BIOS 20172, the learning curve for R can be quite steep. Most people are also new to programming in general (I was when I took the course), and there can be some concepts that you might not pick up until after a few tries.

This primer will serve as a quick review for your first lab this quarter as well as your coding lab from BIOS 20170 last quarter. It's also my attempt at addressing some of the questions from the check-ins that are not within the lecture domain, but I think are still important questions for me to try to answer. Based on where there's less coverage in the lab, I tried to fill in some gaps here (e.g. will not cover matrices since that was covered more in-depth in Lab 1). As I am a lecture TA, please treat this primer more as some personal guidance to complement your labs+lab presentations/videos, not as an official resource. Use at your own discretion – I wrote this up pretty quickly after the check-ins deadline on Sunday and there may be some information that does not overlap with what you need to know from the labs; **only the material from your lab TAs will end up on the coding quizzes.**

If you have any questions about coding, lab TA office hours are held every day between Tues. and Thurs as well as Sun. You're also welcome to email me with any questions at thuang5@uchicago.edu.

Navigating file paths

Let's start by just checking where we are in the computer. We can do this by searching up what **directory** (folder) we're in. If it's not the right folder, we can then change it. Knowing what directory we're in allows us to be able to navigate **from that directory** where other files are, such as data files that we want to read in.

```
#getting working directory that this file is in:
getwd()
```

```
## [1] "/Users/tiffanie/Documents/TA_BIOS_20172/primers"
```

Your "home" directory is usually just your username. So here everything up to my name is like my base location. From there, we are in the Documents directory, and within that, the TA_BIOS_20172 directory, and within *that*, the primers directory. On my laptop they just look like a primers folder within a TA_BIOS_20172 folder within the Documents folder. Let's say I want to set it to my downloads folder instead:

```
setwd("/Users/tiffanie/Downloads")
getwd()
```

```
## [1] "/Users/tiffanie/Downloads"
```

This is called an **absolute file path** because no matter where my file is before I set this working directory, once I set it with this file path, we will always end up moving to Downloads. This is because I am telling R exactly where I want to go, all the way from my "home" directory, starting with "/Users/tiffanie. . . ".

There's a shorter way to do the same thing – again, this is why it's important for us to know our current directory! We can use our current location to tell R to move us according to a **relative file path**. It's exactly what it sounds like – a new location **relative** to our current one. This is useful if you're calling a file that is stored nearby.

When you use `getwd()` in your lab file, you will most likely see an output like

`[1] "/Users/tiffanie/Downloads/Lab1-BIOS20172_Spring2023"`.

This just means your file is in the lab folder, which is great! Locating your data file such as `Zika.fasta` is now easier, because **both are in the same folder/directory**. You would read in the file like so: `Zika <- read.fasta("Zika.fasta")`. The file path is very simple because we're using a **relative file path** where no other directory navigation is needed – they're both in the same folder and so you just need to call the file name.

Layout of R Studio

When you open up your .Rmd files in RStudio for the first time, the layout can be slightly overwhelming. Below your Markdown file editor is a series of tabs, the first one being "Console." Treat your console like scratch paper – ex. you can do simple math that you don't need to keep in your code file or `getwd()` if you momentarily forget which current working directory you're in, etc. The "Terminal" tab won't be used much in this class; this is just an extension of your computer system shell (in Macbooks, extension of Terminal). On your top right side is the environment, where objects that you store values in might appear, such as a dataframe you read in, a vector you create, etc. Below that is something like a help desk. The "Help" tab is very useful for looking up syntax for built-in functions or packages that you install, if you're not familiar with it yet. Previews of any visualizations you create will also pop up here under the "Plots" tab.

Main types of data values

The 3 most important types of data values in R include character (in quotes; a.k.a. string in Python), numeric (integers, doubles/decimals), and logical (a.k.a. boolean in Python). Note that you can convert between data types using built-in functions like `as.integer()` or `as.character()`.

```
#in above order:
"2"
```

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

```
2.29
```

```
## [1] 2.29
```

```
TRUE
```

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

```
#if you're not sure, you can check:
typeof("2")
```

```
## [1] "character"
```

R as a calculator: basic arithmetic operators

R can be used to simplify many of your math calculations, especially if you have a lot of data to work with. A few familiar operators for you incude:

Operation	Symbol in R
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	^
Remainder	%%

```
2+2
```

```
## [1] 4
```

```
2-2
```

```
## [1] 0
```

```
2*2
```

```
## [1] 4
```

```
2/2
```

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

```
2^2
```

```
## [1] 4
```

```
#modulo operator gives remainder of floor division;  $2/2 = 0 R 0$, $5/2= 2 R 1$
2%%2
```

```
## [1] 0
```

```
5%%2
```

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

```
#2(2) #This outputs an error - why?
```

Logical data values

These appear as either `TRUE` or `FALSE`, in uppercase letters. They're interesting to use because they also function as binary values – 1 and 0 specifically. This is useful later on if you incorporate logial data values in conditional statements where you also need to do some math (like proportions, sums, etc.). For now, just know that they also have numerical values of 1 or 0.

```
TRUE*5
```

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

```
TRUE*FALSE
```

```
## [1] 0
```

Logical expressions

We work with logical data values in the context of logical expressions. Like the arithmetic operators, you'll find most of them intuitive/similar to what we write on paper.

Expression	Operator
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=
equal to	==
not equal to	!=
"and" – both true	&&
"or" – either one true	||

Any expression can be checked like this:

```
1+3==2 #FALSE if statement is false
```

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

```
1+3!=2 #TRUE if statement is true
```

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

```
1+3>2
```

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

```
1+3<2
```

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

```
1+3>=4
```

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

```
1+3<=4
```

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

```r
#"AND"
1+3>2 && 1+3==4 #TRUE if both statements are true
```

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

```r
1+3>2 && 1+3==2 #FALSE if not all statements are true
```

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

```r
#"OR"
1+3>2 || 1+3==4
```

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

```r
1+3>2 || 1+3==2 #TRUE if at least one statement true
```

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

```r
1+3>5 || 1+3==3 #FALSE if none of the statements are true
```

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

If/else statements

To make a statement **acting** upon something satisfying a condition that you want to fulfill/you *don't* want to fulfill, we can use if/else statements. The typical structure you'll see is:

```
if(something_is_true){
 do this
} else{
 do that
}
```

The curly brackets are important to remember because they will help you organize your code into little chunks for different conditional expressions. Here's an example:

```r
if(2+3==5){ #if this condition is TRUE
  print("yes, this is true")
} else{ #if this condition is something other than TRUE
  print("no, this is false")
}
```

```
## [1] "yes, this is true"
```

Another example:

```r
if(2>10 || 5<10){
  print("The computer is lying.") #if true
} else{
  print("The computer is telling the truth.") #if false
}
```

```
## [1] "The computer is lying."
```

There are many built-in functions.

Some basic math functions might include log, sin, arctan, and square root of a number. The input, or **argument**, that these functions take can be a number or a number stored in something (like a calculation or a vector of numbers).

```
log(4)
```

```
## [1] 1.386294
```

```
sin(-pi)
```

```
## [1] -1.224647e-16
```

```
atan(3*pi)
```

```
## [1] 1.465089
```

```
sqrt(49)
```

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

It sometimes gets confusing to remember the syntax for so many built-in functions. Luckily, R is very user-friendly in this aspect – if you type in your console `?function_name` into the console, the function's documentation will pop up. You'll get explanations and examples for what kind of arguments you can put into the function. For example:

```
?log
```

Concatenation is another built-in function that we like to use to form **vectors**. We call it with `c( argument of choice )`. This will just glue together all the values you input as a vector.

```
c(1,2,3)
```

```
## [1] 1 2 3
```

We can store things with values like dataframes, vectors, lists, functions, etc. in a named object using `<-`. Note that your output will not print, as you are storing values in a name, but not calling the name.

```
some_numbers <- c(1,2,3) #no output printed
```

You can see the output when you call the named object:

```
some_numbers
```

```
## [1] 1 2 3
```

Indexing

By now, you already know a lot about R because of practice+application in the the past two labs. If you're like me and knowing every syntactical detail puts you more at ease when looking at code, here's a brief overview of indexing:

```
#access (index) elements of an object
some_numbers[1] #index 1
```

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

```
some_numbers[c(1,3)] #index multiple specific elements; non-consecutive
```

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

```
some_numbers[1:2] #slicing - first desired index:last desired index (consecutive)
```

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

```
some_numbers[-2] #access every element except the one at index 2
```

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

When indexing, you use square brackets `[ insert index ]` to access specific elements. When we are using a function, you would use parentheses `( insert argument )`. Curly brackets are often used to organize blocks of your code, such as when you're writing a long function or loop with multiple lines: `{ insert block of code }`.

(This doesn't pertain to those who are programming for the first time.) If you are familiar with any other languages (e.g. Python), you know that when indexing you usually begin at index 0 rather than index 1. Here in R we will start in 1, and the last element that we slice is **inclusive** (rather than indexing 1:10 giving you the second to tenth element, it will give you the first to tenth element).

We can create `some_numbers` the same way using the function `seq()`, which just forms a sequence with 3 arguments: first number, last number, interval (how much space in between each step). The function `identical()` checks if `some_numbers` and `some_numbers2` is identical.

```
some_numbers2 <- seq(1,3,1)
identical(some_numbers,some_numbers2)
```

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

We can also use this function to look at how different data types will not be read as the same even though to us they might *look* the same:

```
identical("2",2)
```

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

Writing functions

While R has a lot of built-in functions, there are often times when it would be more convenient for you to make your own so that you can refer back to that function every time you need to perform the same calculations, access the same elements, etc. For a function, you assign to your custom function a name using the assignment operator, like we did when we assigned a vector a name using `function_name <- function(any arguments here)`. You can specify what inputs, or **arguments**, your function would like to take by making up any name for one or multiple arguments. If you have multiple arguments, they are separated by commas. If you don't need any arguments (you are explicitly coding any values you need into the function's body), you can also leave the space within the parentheses blank – e.g. write it as `function()`.

An example of a general outline for writing a custom function is:

```
function_name <- function(argument1, arg2, arg3){ #argument name can be anything
    body_of_function1 <- argument1+2
    body_of_function2 <- body_of_function1*arg2
    desired_outputs <- body_of_function2/arg3
    return(desired_outputs)
  }
```

Let's break down this function. It takes 3 arguments (can be named anything – all you need to remember is that **order matters!**) – `argument1`, `arg2`, and `arg3`. You can replace these arguments with any numeric values that you want to apply the function on and at the end, your desired outputs will be returned.

Here, the function first adds 2 to whatever value `argument1` is replaced with. This is stored as `body_of_function1`. The second line of the body then takes `body_of_function1` and multiplies it by your 2nd argument, `arg2`, storing this new value into `body_of_function2`. Then `body_of_function2` is divided by `arg3`, your 3rd argument and stored in `desired_outputs`. At the end, `desired_outputs` will be returned to you.

We call this function using the same format that you wrote the arguments in:

```
function_name(argument1=10,arg2=15,arg3=2)
```

```
## [1] 90
```

```
#order matters...
function_name(10,15,2)
```

```
## [1] 90
```

```
function_name(15,2,10)
```

```
## [1] 3.4
```

```
#...unless you specify the argument assignments:
function_name(arg2=15,arg3=2,argument1=10)
```

```
## [1] 90
```

For loops

For loops are used when you want to **iterate** over a data structure such as a list, array, vector, dataframe column, etc. What does this mean? We will run through **individually** and use each **element** in that object

that we're iterating over in a for loop. For example, iterating over a list of numbers from 1 through 10 will let us go through each number at a time, starting at 1. We can then choose to do whatever we want with each "loop"/new element that we iterate through.

The general structure might look like:

```
for(iterating_variable in list){
 do this
}
```

The iterating variable can be any name you want. Many times you'll just use `for(i in named_object)`, but `i` can be named anything you'd like, especially if you are creating a **nested for loop** where you have multiple for loops within each other and need different names for different iterating variables.

Here's an example of me iterating over a list and telling it to do something each loop, or **iteration**:

```
for(i in 1:5){ #for every element that we iterate through in a list of 1-5,
  print(i) #print that indexed element; when i=1, print(i). then repeat with i=2
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
#loop runs again for each element in list
```

Same example, this time with a list of words:

```
for(i in c("mon","tues","wed")){
  print(i)
}
```

```
## [1] "mon"
## [1] "tues"
## [1] "wed"
```

Nested for loop (one inside the other) example:

```
weather <- c("rainy","cloudy","sunny")
time_of_day <- c("morning","afternoon","evening")

for(i in weather){ #larger loop; goes through weather
  for(j in time_of_day){ #nested loop; exhaust each element in time_of_day
    print(c(i,j))
  }
}
```

```
## [1] "rainy"   "morning"
## [1] "rainy"     "afternoon"
## [1] "rainy"   "evening"
## [1] "cloudy"  "morning"
## [1] "cloudy"    "afternoon"
```

```
## [1] "cloudy"  "evening"
## [1] "sunny"   "morning"
## [1] "sunny"     "afternoon"
## [1] "sunny"   "evening"
```

Now with logical expressions! We can use other things we've learned like if/else statements in these for loops.

```
values <- runif(100,min=0,max=100) #generates n amount of random number values
large_number <- 0 #set this object at 0
small_number <- 0

for(i in values){ #looping through each number in our vector "values"
  #begin loop -- looking at 1 number
  if(i>20){ #is this number greater than 20?
    large_number <- large_number+1 #we add one to large_number each time i>20 is True
  }else{
    small_number <- small_number+1 #if i>20 is NOT true, we add 1 to small_number
  }
}

#let's take a look at small_number and large_number:
print(c(small_number,large_number))
```

```
## [1] 20 80
```

Making more complex functions

Now we can try making a more complex function that involves if/else statements. As an example:

```
is_even_num <- function(x){ #create function named is_even_num that takes 1 argument "x"
  if(x%%2==0){ #checks if it's true that the remainder of "x" divided by 2 is 0
    return(TRUE) #returns TRUE if true
  }else{ #otherwise...
    return(FALSE) #it returns FALSE
  }
}

#try calling function w/ different arguments:
is_even_num(4)
```

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

```
is_even_num(5)
```

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

Add a step: incorporating for loops inside a function

For loops are useful within a function, especially if you would like to run a certain for loop that you already wrote on different objects with different values or lengths. Here's an example with a for loop inside that I'll break down after we run it:

```r
new_fxn <- function(num1,num2){
  max_num <- max(num1, num2)
  results <- numeric(max_num) #creates vector of 0s as long as the argument (max_num)
  for(i in 1:max_num){
    if(i==1){
      results[i]=i
    }else{
      results[i]=i*(results[i-1]+1)
    }
  }
  return(results)
}

#try calling the function:
new_fxn(6,2)
```

```
## [1]    1    4   15   64  325 1956
```

This might look complicated, but we can break it down by the concepts we know. The top and bottom of this block of code are just me defining my function to be named `new_fxn`, a custom function that takes 2 arguments: `num1` and `num2`. It returns an output: `results`. That is the *backbone* of our function.

Within the function, we define two things before starting a for loop to run whatever process we want to run: `max_num` and `results`. We assign the maximum value out of the 2 arguments to the named object `max_num`, and a "dummy" vector filled with lots of zeroes into `results`. The function `numeric()` just takes the length of its argument (in this case, `max_num`) and creates a vector of 0s of the same length.

Then we start our for loop. This loop iterates over a list of 1 through the value stored in `max_num`, and for each iteration, we check if the indexed value `i` is equal to 1. If it is, then we store `i` (which is 1) into the ith value of `results`. So if our indexed value was actually equal to 1, we would store the value of `i`, or 1, into the 1st (ith) element of `results`, rewriting the 1st element from 0 to 1, our new value. However, if the indexed element is not equal to 1, we multiply that value, `i`, by the *sum* of the *previous* iteration's stored value in `results` (indexing `i-1`th element of `results`) and 1. Then we store that value into the *current* iteration's ith value of `results`, rewriting the 0 from our "dummy" vector in that index location.

Look at the output: we get six values, which makes sense because our for loop iteration has a length of 6 – it runs from 1 to the maximum value out of 6 and 2, which is 6. For the first iteration value, $i = 1$, we simply store that value back into the 1st index location of `results`. For the next ones, we just perform calculations and store the results of that calculation back into *that* iteration's index value, all the way until we finish our 6th iteration.

While loops

While loops are tricky because you need to make sure that there is an ending point for the loop; otherwise it'll break your computer. We use while loops in instances where we don't know the specific index range of values – whereas in for loops we have a definite beginning and end (such as a list with a specific length). Instead of finishing/**breaking** at the end a specified list length, while loops will break when a *condition* is satisfied; we might not know how long that will take, but we just need to know that **at some point, the condition WILL be satisfied**.

The general structure might look something like:

```
while(condition is true){
 do this
 do something to make condition false eventually
}
```

Here's an example, where condition that is fulfilled and eventually **not** fulfilled is if $x \geq 0$.

```r
x <- 10
while(x>=0){ #while this condition is true,
  print(x) #do this
  x <- x-1 #make modification each loop so that eventually condition is false
}
```

```
## [1] 10
## [1] 9
## [1] 8
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
## [1] 0
```

In this while loop, we set `x` as 10, so each loop, starting from 10, it checks for the condition $x \geq 0$, does an action (prints `x`), then subtracts it by 1 (`x-1`) and stores this new *smaller* value *back* into `x`. In the next rerun of the loop, it repeats the same thing (for 9, 8, 7, etc.) all the way until `x` reaches a value at which it no longer fulfills the condition of being greater than/equal to 0. This happens once `x` becomes -1.

Extra Resources

[1] simpleR: Using R for Intro Stats – Webpage Version

[2] simpleR: Using R for Intro Stats - PDF Version

[3] R-Python "Conversions" (for those more familiar with Python)

[4] R Intro + Practice Problems (note: uses tidyverse which we don't use in this course!)

Hope this was at least somewhat helpful! This primer ended up being longer than expected, but please do not feel pressured to memorize all of this for the coding quiz. You will only be expected to apply similar concepts to what you learned in lab/in Tess's pre-lab videos; any comments I made in this should be seen as more of a personal aside. However, my hope is that this document might supplement+refresh you on some of the things you learned in the coding lab from last quarter/the first lab of this quarter. The labs contain much harder examples than the ones shown, but my goal was to communicate to you how to work your way around R Studio and also how basic R **syntax** works as clearly as possible, since that will lay the foundation for you to write harder code on your own. Good luck on your coding quiz!