Week 1 - Introduction to R

Nicholas Kortessis

First Things First - Download the software

Our first task is to install the programming language R and tools that make it easier to use.

1. Go to an R website (e.g., https://archive.linux.duke.edu/cran/) to do so. OK, but what is R? If you are unfamiliar with R, here is text direct from the source itself:

"R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To download R, please choose your preferred CRAN mirror".

(2) Next please download and install R Studio by visiting the website: https://rstudio.com/products/rstudio/download/

OK, but what is R Studio? Again, from the source:

"RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, and a variety of robust tools for plotting, viewing history, debugging and managing your workspace."

Checkpoint 1: Verify that R and RStudio are both installed by opening RStudio. Can you open and use RStudio?

Notice the default panes:
- Console (entire left)
- Workspace/History (tabbed in upper right)
- Files/Plots/Packages/Help (tabbed in lower right)

FYI: you can change the default location of the panes:
http://www.rstudio.com/ide/docs/using/customizing#pane-layout
Show menu item Tool -> Options

Working in the console

Go into the Console, where we interact with the live R process. Let's do some basic math using R. Type line by line

```
3+4
3-4
3*4
3/4
3^4
```

These are basic arithmetic operations, just like a calculator. The operators are special functions. If you have questions about a function, you can use either `help("function name")` or you can simply write `?function name`.

```
help("+")
?"+"
```

To be fair, these help pages don't seem so helpful at first. As time goes on, you will become more familiar with them and how they work. Some of the most helpful information is at the bottom of the page in "Examples". Copying and pasting these into the command, and experimenting with them often can be very helpful.

One of the beauties of *object-oriented languages* such as R is that information can be stored in objects. Objects come in a diverse array of forms. For example, we might want to store the result of the calculation 3*4. To do so, we just need to give the object a name. For now, let's call it `MyCalc`.

```
MyCalc <- 3*4
MyCalc
```

```
## [1] 12
```

All R statements where you create objects with an "assignment" has this form:

```
ObjectName <- 12
```

and should be read as "assign 12 to the object with the name ObjectName".

You will make lots of assignments and the operator <- is a pain to type. Don't be lazy and use =. An equal sign means something specific, which is that the content on the left and right sides of = are mathematically the same. That's not what is going on here. We are just naming things. As such, stick with `<-` to avoid any confusion in the future. Think about the `<-` as an operation to put a sticker on something. Right now, we are putting stickers on numbers (and sometimes other things!).

Honestly, writing `<-` gets aggravating. So don't! Instead, use RStudio's keyboard shortcut:
- In windows and linux, press alt and the minus sign: alt -
- On Mac OS, press option and the minus sign: option -

Notice that RStudio automagically surrounds <- with spaces, which demonstrates a useful code formatting practice. Code is miserable to read on a good day. Give your eyes a break and use spaces. RStudio offers many handy keyboard shortcuts detailed here.

Naming

It's worth taking a minute to talk about naming objects, since you will be doing a lot of it. There are rules and recommendations.

Rules: Object names cannot start with a digit and cannot contain certain other characters such as a comma or a space.

Recommendation: Pick a convention for demarcating words in names. You'll quickly realize that you'll need multiple words to name something, but you can't use spaces. Here are some common ones.

```
HereIsCamelCasesome.people.use.periodseven_others_use_underscores
```

Let's make another assignment

```
thisIsAReallyLongName <- 2.5
```

To inspect this, try out RStudio's completion facility: type the first few characters, press TAB, add characters until you disambiguate, then press return.

Make another assignment

```
Eight <- 2^3
```

Now let's try to inspect

```
eight
```

Implicit contract with the computer / scripting language: Computer will do tedious computation for you. In return, you will be completely precise in your instructions. Typos matter. Case matters. Get better at typing and FINDING YOUR MISTAKES. Even very experienced coders make mistakes. You'll have to get used to it. What separates experienced coders is that they can identify their mistakes quickly. This warning says

```
Error: object 'eight' not found
```

R is saying it can't find the object `eight`. R rarely makes mistakes. People are very error prone. Since you know you just made it, a first check should be to see whether you made an error. Indeed, we made an error by not capitalizing `eight`; `eight` is not the same as the `Eight`, which is what we told R to identify the object as.

Built-in Functions

R has a mind-blowing collection of built-in functions that are accessed like so

```
funct.name(argument1 = value1, argument2 = value2, and so on)
```

Lets demonstrate a few

```
help(rnorm)
X <- rnorm(100, mean = 5, sd = 1.5)
mean(X)
```

```
## [1] 4.868429
```

```
sd(X)
```

```
## [1] 1.458159
```

Checkpoint 2: What did we just do? To help, look at the object X.

Let's try using seq() which helps make regular sequences of numbers and, while we're at it, demo more helpful features of RStudio. Type `se` and hit TAB. A pop up shows you possible completions. Specify `seq()` by typing more to disambiguate or using the up/down arrows to select. Notice the floating tool-tip-type help that pops up, reminding you of a function's arguments. If you want even more help, press F1 as directed to get the full documentation in the help tab of the lower right pane. Now open the parentheses and notice the automatic addition of the closing parenthesis and the placement of cursor in the middle. Type the arguments 1,10 and hit return. RStudio also exits the parenthetical expression for you. IDEs (integrated development environments) are great.

```
seq(1,10)
```

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

The above also demonstrates something about how R resolves function arguments. You can always specify in name = value form. But if you do not, R attempts to resolve by position. In the case of `seq()`, the help documentation tells us that we can specify the number the sequence starts *from* in the first argument and the number the sequence goes *to* in the second argument (as well as other arguments). So above, it is assumed that we want a sequence "from = 1" that goes "to = 10". The function also has an argument for the step size, but since we didn't specify step size, the default value of "by" in the function definition is used, which ends up being 1 in this case. For functions I call often, I might use this resolve by position for the first argument or maybe the first two. After that, I always use name = value.
Examples:

```
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```
seq(10,1)
```

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

```
seq(to = 10, from = 1)
```

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

```
seq(from = 2, to = 3, length = 10)
```

```
##  [1] 2.000000 2.111111 2.222222 2.333333 2.444444 2.555556 2.666667 2.777778
##  [9] 2.888889 3.000000
```

If you just make an assignment, you don't get to see the value, so then you're tempted to immediately inspect.

```
y <- seq(1,10)
```

In order to see the value when you make the assignment, you can either type the object name (`y` in this case) or by surrounding the assignment with parentheses, which causes the assignment and "print to screen" to happen simultaneously.

```
(y <- seq(1,10))
```

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

Not all functions require arguments:

```
date()
```

```
## [1] "Fri Jan 17 11:56:55 2025"
```

Now look at your workspace - in the lower left pane. The workspace is where user-defined objects accumulate. You can also get a listing of these objects with commands:

```
objects()
```

```
## [1] "Eight"                "MyCalc"               "thisIsAReallyLongName"
## [4] "X"                    "y"
```

```
ls()
```

```
## [1] "Eight"                "MyCalc"               "thisIsAReallyLongName"
## [4] "X"                    "y"
```

If you want to remove something you can use `rm()`. Let's remove `y` like this

```
rm(y)
```

Let's list the objects now

```
ls()
```

```
## [1] "Eight"                "MyCalc"               "thisIsAReallyLongName"
## [4] "X"
```

As expected, `y` is gone. If you ask for it, R will throw a warning that it doesn't exist.

To remove everything:

```
rm(list = ls())
```

or click the broom in the workspace pane.

Checkpoint 3: Any questions about functions in R?

Working with scripts

So far, we have been typing things directly into the command line. Any action you want R to take must occur there. However, this has many downsides. Say you want to run multiple lines of code in a row and you make a mistake on the first line. To fix the mistake, you will need to re-type the multiple lines of code. This isn't the best way to do things. Writing code is a bit like writing a play with actions, and writing in the command line is a bit like specifying the actions in the play in real time.

An alternative is to use a script. These are like text files that can be saved and edited as needed. Think of scripts as complete, well, scripts of a play, but for actions in R.

Start by opening a new script in R by using the pane in the top left with the "+" sign over a white sheet of paper. Once the empty script opens, write the following on the first two lines

```
# BIO 380 Lab; Week 1
# <Insert Your Name Here>
rm(list = ls())
```

The number symbols preceding the text indicate that what follows is not a command to read and evaluated by R. These are simply **comments**. Comments are very helpful as they provide extra information about your goals, intentions, and thoughts when programming.

One of the real values of coding is **transparency and reproducibility.** Code provides documentation of all the operations you do on data to come to your conclusions. One of the basic principles of science is that your experiment and analysis is documented. That used to mean keeping notes of your activities in a lab notebook. For modern analysis, the data are so large and the analyses are so complex that it is hard to write it all down. However, code makes this all available for others to look over. When writing code, it is helpful to write the code in a way that is as readable as possible and provides enough information to know what the code is doing.

You may think this is just for the benefit of other people. It's not! Speaking from personal experience, there have been too many times when I can't decipher my own intentions when looking at old code. If I can't figure out my own intentions, then how will anyone else! Comments can be very helpful in putting enough extra information to make code readable and interpretable for you and others.

Now that we have a script, we can make a little program. This program simulates tossing multiple coins and calculates properties of the coin tosses. (You may have done coin tossing in a high school stats class; real boring stuff; R makes it easy to do this in a much less boring way).

Let's begin by making a coin. In you script write

```
coin <- c("H", "T")
```

where H represents a head and T represents tails. This is just like a real coin. Coin have two features: a head and a tail. This object has two features: a head and a tail.

Now we want to simulate what it would be like to toss the coin. To do so, we will use the function `sample(x, size, replace)`. This function samples from an object x, samples 'size' number of items from x, and 'replace' is an argument that determines whether we should put the item that was sampled back before we sample again. This is perfect for our needs of throwing coins. We can set 'size' to the number of coins, and 'replace' to TRUE. Doing so mimics the exact process of tossing a coin, marking its outcome, and then tossing the coin again 'size' times.

Here is an example where we toss the coin 4 times.

```
sample(coin, size = 4, replace = TRUE)
```

```
## [1] "H" "T" "T" "T"
```

Let's make this a bit more general. BEFORE using the sample function, let's make another object that marks the number of coin tosses.

```
NumTosses <- 4
CoinToss <- sample(coin, size = NumTosses, replace = TRUE)
CoinToss
```

```
## [1] "H" "T" "T" "H"
```

Now if we want to toss more coins, all we need to do is change `NumTosses`, and rerun the script. To run the script efficiently, you can run a single line by typing

- Cmd+return on a Mac
- Cntrl+enter on a Windows based machine

If you want to run the entire script all at once (this is my preference), then highlight everything by typing

- Cmd+A on a Mac
- Cntrl+A in Windows

and then running it in the console as if it were a single line.

Try it with 100 coin tosses, or 1000, or 10,000 if you want!

Now that we have a simple experiment, let's do some simple statistics. First, let's count the fraction that are heads. You'll have to trust me know that these work, or you can intuit what the functions are doing to get it figured out. The point here is not to understand each line of code, but to understand how scripts can be helpful for running simulations and doing analyses.

```
FracHeads <- sum(CoinToss == "H")/length(CoinToss)
FracHeads
```

```
## [1] 0.5
```

Another thing we can do is ask when the first head shows up in a sequence and when the last head shows up.

```
FirstHead <- min(which(CoinToss == "H"))
LastHead <- max(which(CoinToss == "H"))

FirstHead
```

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

```
LastHead
```

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

Checkpoint 4: Run a simulation to find the fraction of heads in a toss of 147 coins and when the first and last heads are in the sequence. Record them as a comment in your script. Save your script and submit it as lab assignment 1.

Before you leave - Load the tutorial 'swirl' in R

Before you leave, download the package called 'swirl' in R. OK, but first, what is a package? A package is a collection of operations and functions that act together for a special purpose. In this case, there are special operations in this package that take you through the basics of how to program in R and how to use the R environment.

Packages can be installed by using the function `install.packages()`. To use this, type the following into the command line:

```
install.packages('swirl')
```

After the package is installed, you will need to load it. After you start working with R and come to use it frequently, you can have hundreds of packages installed. However, you don't need them all at once. When you need one, you can call it up to action. To do so, you write

```
library(swirl)
```

When asked whether you want to install a course, choose option 1 "R Programming: The basics of programming in R". As homework, go through lesson 1: Basic Building Blocks, 3: Sequences of Numbers, 4: Vectors, 5: Missing Values, 6: Subsetting Vectors, and 7: Matrices and Data Frames. After each lesson, the course will ask if you want credit with Coursera.org. Simply answer "No".

Checkpoint 5: Call Dr. Kortessis over to ensure you have the 'swirl' package installed and loaded and verify that you understand the homework assignment.