Workshop 1: Introduction to R

The goal of this first workshop is to get you started working in R, and to introduce the most commonly-used data types, operations, and functions.

First, a quick comment about using "Base R" vs RStudio. When running computationally intensive analyses in R, I prefer to separate the text editor from software that runs R. For this reason, I prefer to work in Base R and edit my text in a separate dedicated text editor. In addition, it is commonplace to run jobs on remote computers or to simultaneously run multiple instances of R on the same machine - this is much easier when working with a dedicated text editor and Base R. In addition, if R crashes, it is nice to not have it take down your text editor (and any unsaved changes) as well.

Try out the command line

The command line in the R console is where you interact with R. The command prompt is a > symbol.

Calculator

At its most basic, the command line is a calculator. The basic operations are

```
+
-
*
/
```

for addition, subtraction, multiplication, division. Familiar calculator functions also work on the command line as built in functions within R. For example, to take the natural log of 2, enter

```
log(2)
```

Note that R interprets the end of a line of input as the end of a command. This is in contrast to some other langues, such as C, which require a symbol such a semicolon to indicate the end of a command.

1. Try the calculator out to get a feel for this basic application and the style of the output. Try `log` and a few other functions (find some online).

2. In R you can store or assign numbers and character strings to named variables called vectors, which are a type of "object" in R. For example, to assign the number 3 to a variable `x`, use

```
x <- 3
```

Note that the = symbol also works for assignment, however, in other contexts, the two symbols are different and so it is more standard convention to use <-. Try assigning a single number to a named variable.

3. In R you can also assign character strings (enter using single or double quotes) to named variables. Try entering

```
z <- 'Hello' # single or double quotes needed
```

4. At any time, enter ls() to see the names of all the objects in the working R environment. You can save the environment for later use by entering save.image() or by saving when you exit R. Assign a single number to the variable x and another number to the variable y. Then watch what happens when you type an operation, such as

```
x * y
```

Finally, you can also store the result in a third variable.

```
z <- x * y
```

To print the contents of z, just enter the name on the command line, or enter print(z).

5. The calculator will also give a TRUE or FALSE response to a logical operation. Try one or more variations of the following examples on the command line to see the outcome.

```
2 + 2 == 4      # Note that == for logical 'is equal to'
3 <= 2          # less than or equal to
'A' > 'a'       # greater than
'Hi' != 'hi'    # not equal to (i.e., \R{} is case sensitive)
```

Vectors

Vectors in R are used to represent variables. R can assign sets of numbers or character strings to named variables using the c() command, for concatenate. R treats a single number or character string as a vector containing just one element.

```
x <- c(1,2,333,65,45,-88)
```

1. Assign a set of 10 numbers to a variable x. Make sure it includes some positive and some negative numbers. To see the contents afterward, enter x on the command line. Is it really a vector? Enter is.vector(x) to confirm.

2. Use integers in square brackets to access specific elements of vector x.

```
x[5] # fifth element
```

Try this out. See also what happens when you enter vectors of indices,

```
x[1:3] # 1:3 is a shortcut for c(1,2,3)
x[c(2,4,9)]
x[c(-1,-3)]
```

Print the 3rd and 6th elements of x with a single command.

3. Some functions of vectors yield integer results and so can be used as indices too. For example, enter the function

```
length(x)
```

Since the result is an integer, it is ok to use as follows,

```
x[length(x)]
```

4. Logical operations can also be used to generate indicators. First, enter the following command and compare with the contents of `x`,

```
x > 0
```

Now enter

```
x[x > 0]
```

Try this yourself: print all elements of `x` that are non-negative.
The `which` command will identify the elements corresponding to `TRUE`. For example, try the following and compare with your vector `x`.

```
which(x > 0)
```

5. Indicators can be used to change individual elements of the vector `x`. For example, to change the fifth element of `x` to `0`,

```
x[5] <- 0
```

Try this yourself. Change the last value of your `x` vector to a different number. Change the 2nd, 6th, and 10th values of `x` all to 3 new numbers with a single command.

6. Missing values in R are indicated by `NA`. Try changing the 2nd value of `x` to a missing value. Print `x` to see the result. You can use the `is.na(x)` command to identify which values are `NA`. See what the following gives you

```
x[!is.na(x)]
```

7. `R` can be used as a calculator for arrays of numbers too. To see this, create a second numerical vector `y` of the same length as `x`. Now try out a few ordinary mathematical operations on the whole vectors of numbers,

```
z <- x * y
z <- y - 2 * x
```

Examine the results to see how R behaves. It executes the operation on the first elements of `x` and `y`, then on the corresponding second elements, and so on. Each result is stored in the corresponding element of `z`. Logical operations are the same,

```
z <- x >= y               # greater than or equal to
z <- x[abs(x) < abs(y)]   # absolute values
```

What does R do if the two vectors are not the same length? The answer is that the elements in the shorter vector are "recycled", starting from the beginning. This is basically what R does when you multiply a vector by a single number. The single number is recycled, and so is applied to each of the elements of `x` in turn.

```
z <- 2 * x
```

You can also create vectors where the elements of vectors are named:

```
x <- c(dog=1, cat=2, mouse=333)
```

and then you can access these variables using their names. E.g., try

```
x['mouse']
```

This can often be very useful, because you will access the correct entry, even if the order of the items in the vector changes.

Lists

Lists are like vectors except that, unlike a vector, the elements of a list do not need to be of the same type. For example, we could create a list of assorted types as such:

```
x <- list(5, 'hello', matrix(1:4))
```

The primary difference between lists and vectors is that, with lists, you access entries using double brackets, rather than single brackets. E.g., try:

```
x[[2]]
```

While using single braces for a list may appear to work, it doesn't do quite what you expect. Best practice is to always uses double brackets, unless you have good reason not to.

As was true for vectors, we can also create lists with named entries:

```
x <- list(a=5, b='hello', c=matrix(1:4))
```

Try

```
x[['a']]
```

You can also add elements to lists simply by indexing an extra element. E.g., try

```
x[['d']] <- 17
```

You can create an empty list, to then fill in later, using the following (this one is length 5)

```
empty.list <- vector(mode = "list", length=5)
```

Functions

One of the beautiful features of R is how easy it is to write your own functions. This is a great way to stream-line your code and minimize repetition (which is also a good way to prevent copy-paste errors). **This will also be an essential skill that you will need to master early for this class.**

1. Syntax for function construction is as follows

```
myfunction1 <- function(x) {
  out <- 2*x^2 + 3
  return(out)
}
```

Here the function I have created is called `myfunction1`. The `return` command tells R what value to return from the function call. The following will call this function with `x=5`, and assign the output to a value `z`

```
z <- myfunction1(5)
z # to look at the output
```

Try this function with some different values of `x`.

2. Now try typing the variable name `'out'` into the R console. This should cause an error:

   ```
   Error: object 'out' not found
   ```

   The variable `out` is only defined *locally* within `myfunction1`, so if you try to access it from "outside" the function, it won't exist.

   Note that functions, by default, return the last line of code, so the above function is equivalent to

   ```
   myfunction1 <- function(x) {
     out <- 2*x^2 + 3
     out
   }
   ```

   which is also equivalent to

   ```
   myfunction1 <- function(x) {
     2*x^2 + 3
   }
   ```

   If the entire function is only a single line of code, you can drop the curly braces, yielding

   ```
   myfunction1 <- function(x)
     2*x^2 + 3
   ```

3. Construct your own function with *two* arguments (call them `x` and `y`). Name this function `myfunction2`. Try calling this function with some different combinations of `x` and `y`.

For loops

A "for" loop is a way to automate a task (and is a common command in most programming languages). For example, try running:

```
for(i in 1:50) {
  print(i)
}
```

1. Write a function that squares a number, and then use a for loop to apply this function to the numbers $1, \ldots, 2$.

2. Write a for loop to fill in a vector with the above values.

3. Write a for loop to fill in a list with the above values.

Apply statements

While for loops are common and easy to use, to really utilize the power of R, you should familiarize yourself with some of the various apply statements. These are essentially wrapped up for loops with some extra features.

There are a number of apply statement: `apply`, `lapply`, `mapply`, `sapply`, `tapply`, `rapply`. The most straightforward is `lapply`. To use this, we could apply `myfunction1` to the vector we created earlier using `lapply`:

```
x <- c(1,2,333,65,45,-88)
lapply(x, myfunction1)
```

1. Try `sapply` instead of `lapply` for the above. `lapply` returns the data in a *list* format, whereas `sapply` returns it as a *vector*.

2. Try using `mapply` to apply `myfunction2` to two vectors such that the first vector is used for the `x` argument and the second vector is used for the `y` argument.

3. Try the following command

   ```
   sapply(1:10, myfunction2, y=5)
   ```

   What happened?