MTH 3240 R Notes 1

1 Introduction to R

1.1 About R

- **R** is a programming language for statistical and scientific computing, data analysis, graphics, and simulations.
- R is free, open-source software that's based on the (proprietary) S and S-Plus languages.
- S was created by John Chambers at Bell Labs in 1977. R was created by Ross Ihaka and Robert Gentleman at the University of Aukland, New Zealand in 1995.
- R is distributed and maintained by the R-Project, whose financial stability is provided by the nonprofit R Foundation.
- **R packages** (add-ons) are available for free download and installation. Anyone (e.g. you or me) can contribute an R package.

1.2 R Download

- To download R, go to the R website

  `http://www.r-project.org`

  and then to the Download page (CRAN).

1.3 About R Studio

- **R Studio** is front-end (i.e. user interface) for R that helps you organize your work and makes using R more pleasant.
- R Studio is free software, and is available for download from `https://www.rstudio.com/`

2 Getting Started

2.1 Arithmetic Operators

- R can be used as a calculator. Arithmetic expressions are typed on the command line in the R Console window, and are evaluated upon hitting 'Enter'.
- The syntax for several mathematical operators is shown below, ordered from highest to lowest precedence:

```
^                    # Exponentiation (right to left in the case of
                     # "stacked" exponents)
-                    # Unary minus sign
*  /                 # Multiplication, Division
+  -                 # Addition, Subtraction
```

- Within an expression, higher precedence operations are carried out first. If two or more operators have equal precedence, they're evaluated from left to right.
- Parentheses, `( )`, can be used to change the order of operations. Operations in parentheses are carried out first.
- For more information on these and other operators, type:

```
? Syntax
```

Section 2.1 Exercises

Exercise 1 Guess what the result of each of the following will be, then check your answers:

a) `4 + 2 * 8`

b) `4 + 2 * 8 + 3`

c) `-2^2`

d) `1 + 2^2 * 4`

e) `(2 + 4) / 3 / 2`

2.2 Special Characters, Special Values, Etc.

2.2.1 White Spaces

- Extra spaces are ignored by R. For example, the following produce the same result:

```
2+2
2 + 2
2    +     2
```

2.2.2 Continuing a Command on the Next Line

- If a command isn't complete at the end of a line (and you hit 'Enter' anyway), R will give a different prompt, the '+' sign (not shown below), on subsequent lines and continue to read input until the command is complete:

```
3 + 2 * (8 -
6)

## [1] 7
```

2.2.3 Special Character:

- The # symbol is used for comments. Anything after # (and in the same line) is not evaluated in R.

2.2.4 Special Values: Inf and NaN

- Occasionally a computation will result in one of the following special values:

```
Inf       # Infinity
NaN       # "Not a number"
```

- Any positive number divided by 0 will result in Inf, whereas 0 divided by 0 results in NaN.
- Inf can be used in calculations and it behaves just like ∞.

Section 2.2 Exercises

Exercise 2 Guess what the result of each of the following will be, then check your answers:

a) `5 / 0`

b) `1 / Inf`

c) `0 / 0`

d)
```
Inf + 1
```

2.3 Variables and the Assignment Operator

2.3.1 Introduction

- In R, (single-valued) ***variables*** are used to store numerical values. We assign values to variables using the ***assignment operator***:

  ```
    <-                    # Assigns a value to a variable
  ```

  For example below the value `10` is assigned to the variable `x`:

  ```
  x <- 10
  ```

  This stores the value 10 at a location in the computer's memory and then associates the name `x` with that memory location.

- To view the contents of a variable, type its name on the command line and hit 'Enter':

  ```
  x

  ## [1] 10
  ```

2.3.2 Variable Naming Conventions

- Variable names can be any length and can contain letters, numbers, and '`.`' and '`_`' characters, but they must begin with a letter or a '`.`'.

- R is *case sensitive*, so `x` and `X` are different symbols and would refer to different variables.

2.3.3 Using Variables in Computations

- Once a value has been assigned to a variable, we can perform computations involving that variable. For example, using the variable `x` from above:

  ```
  (x^2 + 5)/5

  ## [1] 21
  ```

2.3.4 Overwriting the Value of a Variable

- We can use the assignment operator `<-` to overwrite the value of a variable:

```
x <- 11
x

## [1] 11
```

Above, the value 10 previously stored in `x` was overwritten by the new value 11.

- The same variable can appear on both sides of an assignment operator. The right side is always evaluated first:

```
x <- x + 1              # x is 11 on the right side of <-.
x

## [1] 12
```

2.3.5 Other Types of Variables

- Variables can store not just numerical values, but any of the so-called ***atomic*** types of values:

  ”`double`”
  ”`character`”
  ”`integer`”
  ”`logical`”

  or they can be `NULL`, in which case the variable is interpreted as being empty:

```
NULL                  # Represents an "empty" variable
```

- We can check the type of a variable using the `typeof()` function:

```
typeof()     # Check the type of a variable.  Returns either "double",
             # "integer", "character", "logical", or "NULL".
```

- Here are a couple of examples:

```
num.var <- 3.14159
typeof(num.var)

## [1] "double"
```

```
char.var <- "a"
typeof(char.var)

## [1] "character"
```

```
logic.var <- TRUE
typeof(logic.var)

## [1] "logical"
```

- Most numeric variables are ”`double`”, which stands for *double-precision floating-point*. These variables can store both integer values and non-integer decimal values.

  Occasionally, a numeric variable is ”`integer`”. These can only store integer values.

  In either case, we can check that a variable is numeric using the `is.numeric()` function:

  ```
  is.numeric()      # Checks to see if a variable is numeric.  Returns
                    # TRUE if the variable is either "double" or "integer"
                    # and FALSE otherwise.
  ```

- Similarly, we can check that a variable is `"character"` or `"logical"` using the functions:

  ```
  is.character()    # Checks to see if a variable is "character".
  is.logical()      # Checks to see if a variable is "logical".
  ```

Section 2.3 Exercises

Exercise 3 What type of variable is created in each of the following commands? Check your answers by typing `typeof(x)`:

a) `x <- 45.3`

b) `x <- "foo"`

c) `x <- FALSE`

d) `x <- NULL`

Exercise 4 Guess the output from each of the following pairs of commands. Then check your answers:

a)
```
x <- 45.3
is.numeric(x)
```

b)
```
x <- 45.3
is.character(x)
```

c)
```
x <- "foo"
is.numeric(x)
```

d)
```
x <- "foo"
is.character(x)
```

2.4 The R Workspace

2.4.1 Viewing and Removing Objects from the Workspace

- R calls the directory (folder) in which it stores user-created ***objects*** such as variables and data sets the ***Workspace***. To view or remove objects from the Workspace, we use:

```
ls()                    # List the objects in the Workspace
objects()               # List the objects in the Workspace (same as
                        # ls())
rm()                    # Remove objects from the Workspace
```

- For example, type `ls()` (or `objects()`) to see what's currently stored in the Workspace:

```
ls()

## [1] "char.var"  "logic.var" "num.var"   "x"
```

  Right now, the objects in the Workspace are the four variables we created earlier.

- To remove `x` from the Workspace, use `rm()`:

```
rm(x)
```

  Now we get:

```
ls()

## [1] "char.var"  "logic.var" "num.var"
```

indicating that `x` no longer exists.

- To remove *all* objects from the Workspace, use:

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

2.4.2 ”Save the Workspace Image?”

- When you end an R session (for example by typing `q()`) you’ll be asked if you want to ”Save the Workspace Image?”. If you choose to do so, the objects you created in the current R session will be available for re-use in future sessions. Otherwise, they won’t.

Section 2.4 Exercises

Exercise 5 Create a few variables named `x`, `y`, and `z` (using any values). Then type the following sequence of commands, paying attention to the output from `ls()` each time:

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

What are the outputs from the three calls to `ls()`?

2.5 Introduction to Functions

2.5.1 Using Built-In Functions

- R has an extensive set of built-in ***functions***, a few of which are listed below:

```
sqrt()                # Square root
abs()                 # Absolute value
round()               # Round a value to a specified number of digits
signif()              # Express a value to a specified number of
                      # significant digits
log(); log10()        # Natural logarithm, base 10 logarithm
exp()                 # Exponential function (exp(1) is the exponential
                      # constant e, exp(2) is the square of e, etc.)
factorial()           # Factorial
```

```
sin(); cos(); tan()   # Trigonometric functions sine, cosine, tangent
```

- Each function accepts one or more values passed to it as ***arguments***, performs computations or operations on those values, and returns a result.
- To perform a ***function call***, type the name of the function with the values of its argument(s) in parentheses, then hit 'Enter':

```
sqrt(2)

## [1] 1.414214
```

Values passed as arguments can be in the form of variables, such as `x` below:

```
x <- 2
sqrt(x)

## [1] 1.414214
```

or they can be entire expressions, such as `x^2 + 5` below:

```
sqrt(x^2 + 5)

## [1] 3
```

2.5.2 Getting Help

- Here are some ways to get help for an R function or operator:

```
help()          # Open the built-in html help file for a function (or
                # an operator in quotes, e.g. help("*"))
?               # Open the built-in html help file for a function (or
                # an operator in quotes, e.g. ? "*")
```

- Passing a function name to `help()` opens the html help file for that function. For example, typing:

```
help(sqrt)
```

opens the help file for the function `sqrt()`.

Typing `?` followed by a function name does the same thing as `help()`:

```
? sqrt
```

- Use quotations for help on operators represented by symbols, for example:

```
? "*"
```

- If you're not sure how to do something in R, Google usually returns suggestions.

Section 2.5 Exercises

Exercise 6 Look at the help file for `sqrt()` by typing:

```
? sqrt
```

a) Besides `sqrt()`, what other R function is described in the help file?

b) From the help file, how many arguments does `sqrt()` have?

2.5.3 Viewing a Function's Arguments

- Most functions take multiple arguments, each of which has a name, and some of which are optional.
- We can see what arguments a function takes and which ones are optional in its help file.

  For example, to see what arguments `round()` takes, we'd type:

```
? round
```

We see from the help file that `round()` has two arguments, `x`, the numeric value to be rounded, and `digits`, an integer specifying the number of decimal places to round to. Thus to round 4.679 to 2 decimal places, we type:

```
round(x = 4.679, digits = 2)

## [1] 4.68
```

2.5.4 Optional Arguments and Default Values

- The specification `digits = 0` in the help file for `round()` tells us that `digits` has a ***default value*** of 0. This means that it's an ***optional argument*** and if no value is passed for that argument, rounding is done to 0 decimal places (i.e. to the nearest integer).

2.5.5 Named Argument Matching and Positional Matching

- When we type:

  ```
  round(x = 4.679, digits = 2)

  ## [1] 4.68
  ```

  we specify values for the arguments by their names (`x` and `digits`).

- When ***named argument matching*** is used, as above, the order of the arguments is irrelevant. For example, we get the same result by typing:

  ```
  round(digits = 2, x = 4.679)

  ## [1] 4.68
  ```

- Another way to specify values for the arguments by their positions, for example:

  ```
  round(4.679, 2)
  ```

  R knows, by ***positional matching***, that the first value, 4.679, is the value to be rounded and the second one, 2, is the number of decimal places to round to.

  For example, R would do something completely different if we typed:

  ```
  round(2, 4.679)
  ```

- The two types of argument specification (positional and named argument matching) can be be mixed in the same function call.

Section 2.5 Exercises

Exercise 7 Look at the arguments for the function `signif()` by typing:

```
? signif
```

The function `signif()` prints the value passed for `x` to the number of significant digits specified by `digits`.

a) From the help file for `signif()`, what is the default value for `digits`?

b) To how many significant digits will the value 342.88937 be printed by the following command?

   ```
   signif(x = 342.88937)
   ```

c) Write a command using *named argument matching* that prints the value 342.88937 to 5 significant digits.

d) Write a command using *positional matching* that prints the value 342.88937 to 5 significant digits.

2.6 Editing Commands

- Some keystrokes for editing commands in the R Console window are shown below.

```
→             # Move the cursor forward one space.
←             # Move the cursor backward one space.
'Home'        # Move the cursor to the start of the line.
'End'         # Move the cursor to the end of the line.
↑             # Move up to the previous line.
↓             # Move down to the next line.
'Delete'      # Delete the current character.
'Backspace'   # Delete the preceding character.
Ctrl+c        # Copy the selected text to the clipboard (use the left
              # mouse button held down to select text).
Ctrl+v        # Paste the contents of the clipboard to the Console
              # window.
Ctrl+l        # Clear the Console window (that's a lower case "L", not
              # the number 1).
Ctrl+o        # Toggle the "overwrite" mode (initially off).  You can
              # also use the 'Insert' key.
Esc           # Interrupt the current computation.
```

Section 2.6 Exercises

Exercise 8 Typing ↑ (the up arrow key) moves the cursor up to the previous line, and typing ↓ (the up arrow key) moves it down to the next line.

Type the following sequence of commands, hitting 'Enter' after each one:

```
x <- 4
y <- 5
z <- 6
x + y
```

Then hit the up arrow key ↑ four times followed by the down arrow key ↓ twice. What command do you end up on?