MTH 3220 R Notes 1

1 Getting Started

1.1 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
%%                  # Modulo (i.e. remainder)
%/%                 # Integer divide
*  /                # 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 1.1 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`

f) `3^2^3`

Exercise 2 The operator `%%` returns the remainder when one number is divided by another. More precisely, for two numbers `x` and `y`,

```
x %% y
```

returns the remainder after `x` is divided by `y`.

The operator `%/%` performs "integer division" in which the remainder is discarded.

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

a) `4 %% 3`

b) `15 %% 6`

c) `4 %/% 3`

d) `15 %/% 6`

1.2 Special Characters, Special Values, Etc.

1.2.1 White Spaces

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

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

1.2.2 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, on subsequent lines and continue to read input until the command is complete:

```
3 + 2 * (8 -


6)

## [1] 7
```

1.2.3 Special Characters: ; and

- We can put more than one R statement on a line, separated by semicolons, ';'. R executes the statements sequentially, for example:

```
2 * 2;  2^3

## [1] 4
## [1] 8
```

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

1.2.4 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 1.2 Exercises

Exercise 3 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`

e) `Inf + Inf`

f) `Inf - Inf`

g) `0 * Inf`

1.3 Variables and the Assignment Operator

1.3.1 Introduction

- In R, ***variables*** (or ***scalars***) 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
```

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

```
x

## [1] 10
```

1.3.2 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.

1.3.3 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
```

1.3.4 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

## [1] 12
```

1.3.5 Other Types of Variables

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

```
"double"          "integer"
"character"       "logical"
"complex"         "raw"
```

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", "complex", "raw", or
             # "NULL".
```

- Here are a couple of examples:

```
x <- 3.14159
typeof(x)

## [1] "double"
```

```
y <- "a"
typeof(y)

## [1] "character"
```

```
z <- TRUE
typeof(z)

## [1] "logical"
```

- Most numeric variables are "`double`", which stands for *double-precision floating-point*. These variables can store any real (i.e. non-complex) number, i.e. they 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".
```

- The last two variable types, `"complex"` and `"raw"`, are rarely encountered in practice.

Section 1.3 Exercises

Exercise 4 Write a sequence of commands that do the following (in order):

1. Create a variable `y` containing the value 5.
2. Overwrite the value of `y` by the value of `3 * y` (by putting `3 * y` on the right side of the assignment operator).
3. Copy the current value of `y` into a new variable `z` (by putting `y` on the right side of the assignment operator).

Exercise 5 Guess the resulting value of `x` will be after the following sequence of commands is executed. Then check your answer.

```
x <- 2
x <- x * 2 + 1
x <- x * 3
```

Exercise 6 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 <- "a"
is.numeric(x)
```

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

e)
```
x <- FALSE
is.logical(x)
```

1.4 The R Workspace

1.4.1 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:

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

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

```
ls()

## [1] "x" "y" "z"
```

  Right now, the only object in the Workspace is the variables `x`, `y`, and `z` created earlier.

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

```
rm(x)
```

  Now we get:

```
ls()

## [1] "y" "z"
```

  which shows that `x` no longer exists in the Workspace.

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

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

1.4.2 "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 1.4 Exercises

Exercise 7 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())   # Only do this if you are OK with clearing the Workspace
ls()
```

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

1.5 Introduction to Functions

1.5.1 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
sign()              # Returns -1, 0, or +1 depending on whether its
```

```
                        # argument is negative, zero, or positive
round()                 # Round a value to a specified number of digits
signif()                # Express a value to a specified number of
                        # significant digits
floor()                 # Largest integer not greater than a value
ceiling()               # Smallest integer not less than a value
trunc()                 # Truncate a value toward 0
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
choose()                # Number of ways to choose x objects from n
                        # objects
sin(); cos(); tan()     # 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'. For example:

```
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
```

1.5.2 Viewing a Function's Named Arguments

- Most functions take multiple arguments, each of which has a name, and some of which are optional.
- One way to see what arguments a function takes and which ones are optional is to use the function:

```
args()                  # View a function's named arguments
```

- Another way to view a function's arguments is to look at its help page (see Subsection 1.6).
- For example, to see what arguments `round()` takes (using `args()`), we'd type:

```
args(round)

## function (x, digits = 0)
## NULL
```

We see that `round()` has two arguments, `x`, a 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(4.679, 2)

## [1] 4.68
```

1.5.3 Optional Arguments and Default Values

- The specification `digits=0` in the output from `args(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).

1.5.4 Positional Matching and Named Argument Matching

- When we type

```
round(4.679, 2)

## [1] 4.68
```

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.

- We can also specify values for the arguments by name, for example:

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

## [1] 4.68
```

- 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
```

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

```
round(4.679, digits = 2)

## [1] 4.68
```

Section 1.5 Exercises

Exercise 8 Recall that the real roots of a quadratic equation

$$ax^2 + bx + c = 0$$

are given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The function `sqrt()` will compute the square root of a value. Write R commands that compute the roots of the quadratic equation

$$x^2 + 7x + 3 = 0$$

Exercise 9 Look at the arguments for the function `signif()`:

```
args(signif)

## function (x, digits = 6)
## NULL
```

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

a) What's the *default* value for `digits`? To how many significant digits will the value 342.88935224 be printed by the following command?

```
signif(x = 342.88935224)
```

b) The following commands both use *named argument matching* to pass values as arguments to `signif()`. Will they produce the same result?

```
signif(x = 342.88935224, digits = 4)
```

```
signif(digits = 4, x = 342.88935224)
```

c) The following both commands use *positional matching* to pass values as arguments to `signif()`. Will they produce the same result?

```
signif(342.88935224, 4)
```

```
signif(4, 342.88935224)
```

1.6 Getting Help

- Here's one way to get help for an R function or operator:

```
?               # Open the built-in html help page for a function or
                # operator
```

- Typing `?` followed by a function name opens the html help page for that function. For example, typing:

```
? sqrt
```

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

- Use quotations for help on operators represented by symbols. Here's an example:

```
? "/"
```

- If you don't know the name of the R function, a quick Google search will almost always lead to some results.

Section 1.6 Exercises

Exercise 10 Look at the help page for `sqrt()` by typing:

```
? sqrt
```

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

b) From the help page, how many arguments does `sqrt()` have? What is it named?

1.7 A Preview of R Data Structures

- There are five ways to store data sets in R. These differ according to their dimensionality (1D, 2D, or nD) and whether they're ***homogeneous*** (all contents must be of the same type) or ***heterogeneous*** (contents can be of different types):
  - **Vectors** (1D, homog.)
  - **Lists** (1D, heterog.)
  - **Matrices** (2D, homog.)
  - **Data Frames** (2D, heterog.)
  - **Arrays** (nD, homog.)

1.7.1 A Preview of Vectors

- ***Vectors*** (sometimes called ***atomic vectors*** because each element of a vector is a single value, or "atom") are created using the "combine" function:

```
c()                    # Combine values to form a vector
```

- Here's an example:

```
x <- c(7, 4, 5)
x

## [1] 7 4 5
```

- Vectors can store any of the six so-called ***atomic*** types (`"double"`, `"integer"`, `"character"`, `"complex"`, and `"raw"`). Here's one that stores `"character"` values:

```
y <- c("a", "b", "c")
y

## [1] "a" "b" "c"
```

and here's one that stores `"logical"`:

```
z <- c(TRUE, TRUE, FALSE)
z

## [1]  TRUE  TRUE FALSE
```

- The functions `typeof()`, `is.numeric()`, `is.character()`, and `is.logical()` work on vectors too:

```
typeof(z)

## [1] "logical"
```

1.7.2 A Preview of Matrices

- ***Matrices*** are like two-dimensional vectors (i.e. they have rows and columns). One way to create a matrix is using:

```
matrix()              # Create a matrix, from a vector, with a speci-
                      # fied number of rows and columns
```

- Here's an example:

```
my.mat <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3)
my.mat

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

Note that by default, R fills the matrix by columns, left to right. The values in square brackets `[ ]` indicate the row or column number.

1.7.3 A Preview of Lists

- The elements of an atomic vector all have to be the same type. ***Lists*** are vectors whose elements can be of mixed types. One way to create a list is using:

```
list()              # Create a list from a set of R objects
```

- Here's an example:

```
my.list <- list("d", 12, TRUE)
my.list

## [[1]]
## [1] "d"
##
## [[2]]
## [1] 12
##
## [[3]]
## [1] TRUE
```

The value in double brackets `[[ ]]` indicates the position of the element in the list (i.e. its *index*).

- In fact, elements of a list can be *any* R objects. For example, here's a list of two vectors:

```
my.list <- list(c("a", "b", "c"), c(7, 4, 5))
my.list

## [[1]]
## [1] "a" "b" "c"
##
## [[2]]
## [1] 7 4 5
```

1.7.4 A Preview of Data Frames

- ***Data frames*** are like matrices, but the types of values they contain can differ from one column to the next (e.g. they can contain some `"numeric"` columns and some `"character"` ones).
- One way to create a data frame is with the function:

```
data.frame()          # Create a data frame from a set of vectors
                      # (which will form the columns of the data frame)
```

- We can choose names for the columns of a data frame in the call to `data.frame()`. In the example below, we use `Var1` and `Var2` for the column names:

```
my.df <- data.frame(Var1 = c("a", "b", "c"), Var2 = c(7, 4, 5))
my.df

##   Var1 Var2
## 1    a    7
## 2    b    4
## 3    c    5
```

1.7.5 A Preview of Arrays

- An ***array*** is like a matrix, but it can have more than two dimensions (e.g. it can have three dimensions: rows, columns, and layers). We can create an array using:

```
array()               # Create an array, from a vector, with number of
                      # dimensions specified by the dim argument
```

- For an example, here's a three-dimensional array with two rows, two columns, and three layers:

```
my.array <- array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), dim = c(2, 2, 3))
my.array

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

```
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 3
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
```

Section 1.7 Exercises

Exercise 11 Write a command using `c()` that creates a vector containing the values:

$$3, 7, 2, 8$$

Exercise 12 Write a command using `matrix()` that creates the following matrix:

$$\begin{pmatrix} 2 & 8 & 14 \\ 4 & 10 & 16 \\ 6 & 12 & 18 \end{pmatrix}$$

Exercise 13 Write a command using `list()` that creates a list containing the following elements:

$$\text{"e"}, 9, \text{TRUE}$$

Exercise 14 Write a command using `data.frame()` that creates a data frame containing the following data set:

TrtGroup	Response
A	5
A	4
B	6
B	6
C	9
C	8