Module One

1. Simple Operations

All of the typical mathematical operations can be directly entered into the console. Operations such as addition, subtraction, multiplication, division and exponentiation are referenced by their corresponding symbols +, -, /, * and ^.

Note that the R code is presented in the shaded blocks and the corresponding output(if any) is presented in the following white blocks.

Some basic math

```
1+2-3
```

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

```
3^2
```

```
## [1] 9
```

More advanced mathematical operations can be performed by invoking specific functions within R. The following R expressions represent, respectively, $\sqrt{9}, e^2, log_e 2, log_2 5, sin(5), 2^9$:

```
sqrt(9) # Squre root
exp(2) # Expoential
log(5) # Log with base e=2.71828...
log2(5) # Log with base 2
sin(5) # Sine
2^9 # power
```

For a more complex example, to calculate the expression $\sqrt{\log_2(10)} + e^{3\cos(3\pi)}$, we could type the following command:

```
sqrt(log2(10))+exp(3*cos(3*pi))
```

Logical math

Logical math includes statements such as "is equal to" and "not equal to." Some examples include the following:

```
1==2 # is equal to
```

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

```
1!=2 # not equal to
```

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

"AND", "OR", and "NOT" statements

```
TRUE & FALSE # AND
```

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

```
TRUE | FALSE # OR
```

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

```
!(2>1) # NOT
```

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

```
(3.5>3)&(2==2)
```

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

Variable Assignment and Types

Variable names in R are case-sensitive. The assignment of a value to a variable in R is accomplished via the <- symbol. Other languages use the = operator in place of the <- operator to denote assignment. R is capable of supporting both conventions. In this class we will stick with <- in order to minimize confusion. You are strongly encouraged to use <- instead of = when programming in R.

```
x<-3 # we pass the value 3 to the variable x
print(x) # view x using the print function
```

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

Other examples

```
str2<-str1<-"I love Biostatistics!" # we pass the sentence to the two variables str1 and str2
str3=str2 # we pass the value of str2 to a new variable str3. This is the same as str3 <- str2.
str1
```

```
## [1] "I love Biostatistics!"
```

```
str2
```

```
## [1] "I love Biostatistics!"
```

```
str3
```

```
## [1] "I love Biostatistics!"
```

The basic types of the variables include "numeric", "character", and "logical."

```
x # view the value of x
```

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

```
class(x) # view the class(or type) of x
```

```
## [1] "numeric"
```

```
str1 # view the value of str1
```

```
## [1] "I love Biostatistics!"
```

```
class(str1) # view the class(or type) of str1
```

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

```
A<-str1==str2
class(A)
```

```
## [1] "logical"
```

2. Vector and Matrix Operations

A vector can be thought of as a 1-dimensional array. It can hold data of similar type. Only numbers or only characters can be placed inside a vector. The concatenation operator `c()` is used to create a vector of numbers or strings.

Creating Vectors

```
v1<-c(1,2,3,4,5)
v1
```

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

```
v2<-rep(0,5)   # create a vector of (0,0,0,0,0)
v2
```

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

```
v3<-seq(from=0,to=5,length=6)
v3
```

```
## [1] 0 1 2 3 4 5
```

```r
v4<-seq(0,5,by=1)
v4
```

```
## [1] 0 1 2 3 4 5
```

```r
v5=1:5   # Another way of getting seq(0,5)
v5
```

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

```r
s1<-c("a", "b", "hello")
s1
```

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

```r
s2<-c("a", 1, 2)
s2
```

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

Subsetting Vectors

To get one element or some elements of a vector, we use the index inside the brackets.

```r
v1[1] # the first element of v1
```

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

```r
v1[2:4] # the second, third, fourth elements of v1, which is a vector with length 3
```

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

```r
v1[v1>2] # the elements in v1 that are greater than 2
```

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

Operations on Vectors

The next few examples illustrate mathematical operations that can be performed on vectors. The operations are often done element-wise.

```r
x1  <- c(2,4,6,8)
x2 <- 2*x1 #multiply each element by 2
x2
```

```
## [1]  4  8 12 16
```

```r
x3 <- x1^2 # element-wise square
x3
```

```
## [1]  4 16 36 64
```

```r
x4 <- x1/x2  # element-wise division
x4
```

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

```r
x5 <- sqrt(x1)  # element-wise square-root
x5
```

```
## [1] 1.414 2.000 2.449 2.828
```

```r
x6 <- sum(x1)  # sum up all elements
x6
```

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

```r
x7 <- max(x1)  # maximum element
x7
```

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

Creating a Matrix

A matrix can be thought of as a 2-dimensional vector. Matrices also hold data of similar type. The following code defines a matrix with 2-rows and 3-columns. In R, matrices are stored in columnar format by default.

```r
M1 <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = TRUE)
M1
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```r
M2 <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3) # same as if specify: byrow = F
M2
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Subsetting a Matrix

The elements are extracted from a matrix using 2 indices inside the brackets.

```r
M2[2,3]   # The element at the 2nd row and 3rd column
```

```
## [1] 6
```

```r
M2[,2:3]  # All elements in the 2nd and 3rd columns
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

Matrix Multiplication

The * indicates element-wise multiplication between two matrice

```r
M1*M2  # multiply element-wise
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   15
## [2,]    8   20   36
```

Matrix multiplication is done by %*%

```r
M1%*%t(M2) # t() is the transpose of a matrix
```

```
##      [,1] [,2]
## [1,]   22   28
## [2,]   49   64
```

3. Conditional Statements

The If-Else Statement

The statement within the parenthesis after the if keyword is a boolean(or logic) expression. It can either be TRUE or FALSE. If the value is TRUE, the code within the curly braces will be evaluated. If the statemenent is FALSE, the code within the curly braces will not be evaluated.

```r
x <- 0
y <- 20
 if(x == 0) {# if x is 0, then set y equal 10.
    y <- 10
 }
y
```

```
## [1] 10
```

If we want to provide an alternate evaluation branch, we can use the else keyword.

```
x <- 1
 if(x == 0) {# if x is 0, then set y equal 10.
    y <- 10
 } else {# if x is not 0, then set y equal 20.
    y <- 20
 }
y
```

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

Loops

The `for()` and `while()` structures are typically utilized when the user wants to perform a specific operation many times.

for()

```
 x <- c()
 for(i in 1:5) {
     x[i] <- i
 }
x
```

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

In the previous example the iterator i took values between 1 and 5. Note that any variable name can be used as an iterator.

while()

Another popular looping structure is the `while()` loop. The loop will perform a certain calculation until the boolean expression provided to it returns a FALSE value.

```
x <- 5
 while(x < 6) {
   x <- x + 1
 }
x
```

```
## [1] 6
```

4. Writing functions

In many cases, it is convenient to create our own custom functions to perform certain tasks. Here is an example of a simple function that accepts two inputs and produces one output with their sum:

```r
mySum  <- function(a, b) {
   return(a+b)
 }
mySum(1,3)
```

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

Note that you return something either by using the `return()` statement explicity, or implicitly: R will take the last line of the function to be the return value.

5 . Integration

Integration can be performed using R. In this example, we calculating the integral:

$$\int_0^\infty \frac{1}{(x+1)\sqrt{x}} dx = \pi.$$

```r
integrand <- function(x) 1 / ((x+1) * sqrt(x)) # writing the integrand function
integrate(integrand, lower = 0, upper = Inf)
```

```
## 3.142 with absolute error < 2.7e-05
```