Chapter 3 Programming Fundamentals

3.1 Logicals

A logical is a binary representation of True and False:

```
  a <- TRUE

  b <- FALSE

  typeof(b)
```

Logicals are used for evaluating comparisons, such as:

```
  #equality
  # note the double equals == operator is a logical comparison, opposed to a
single equal = being an assignment

  2==2
  typeof(2==2)

  'cat' == 'dog'

  'cat' == 'cat'

  #Not equal operator
  2!=2

  # greater than/less than
  2>2

  2>=2

  2<1

  2<=1

  # Null Values
  V1 <- 1
  V1[10] <- 10

  is.na(V1)
  !is.na(V1)

  # Contained within set
  V1

  1 %in% V1
  10 %in% V1
  2 %in% V1
```

```r
c(1,2) %in% V1

# If we want not in, then use the not ! operator around the entire statement
!(1 %in% V1)

# Note, logicals can be used with other data types as well
a <- c(1,2,3,4,5)
a <= 1

b <- c(1,2,7,9,5)
a == b
a != b

A <- matrix(1:10,2,5)
B <- matrix(seq(1,20,2),2,5)

A == B
A != B

identical(a,b)
identical(A,B)

all.equal(A,B)
```

3.2 Conditionals

Conditionals are expressions that perform different computations or actions depending on whether a predefined boolean condition is TRUE or FALSE. Conditionals allow us to control the flow of execution. We use logicals to evaluate conditional statements. Conditionals include if, else, ifelse.

The syntax of if statement is:

```r
if (test_expression) {
statement
}
```

If the test_expression is `TRUE`, the statement gets executed. But if it is `FALSE`, nothing happens. For example:

```r
x <- 5
if(x > 0){
print("Positive number")
}
```

If you will be executing a single statement after the conditional, you can withold the { }

```r
x <- 5
if(x > 0) print("Positive number")
```

If we want code to execute when the test_expression is `FALSE`, then we add an else statement:

```
  if (test_expression) {
  statement
  } else {
  statement2
  }
```

For example:

```
  x <- -5
  if(x > 0){
  print("Non-negative number")
  } else {
  print("Negative number")
  }
```

Note that you can use multiple logicals in the test_expression, such as:

- Or operator |
- And operator &

For example:

```
  mean_values <- 3.1
  std_dev_values <- 1.95

  if (mean_values>=0 & std_dev_values>=1) {
      print(paste("The mean", mean_values, "is above 0, and the standard
deviation", std_dev_values, "is above 1"))
  }
```

Additionally, you can use the ifelse function to quickly assign a value based on a condition. The syntax for the ifelse function is `ifelse(test_expression, yes, no)`. For example:

```
  std_dev_vector <- c(1.2,0.8,0.3,2.4)
  ifelse(std_dev_vector>=1, "above one", "below one")
```

3.3 Loops

Occassionally, we need perform a task in an iterative cycle, also known as a loop. According to the R base manual, among the control flow commands, the loop constructs are `for`, `while` and `repeat`, with the additional clauses `break` and `next`.
A for loop is used for iterating over a sequence (they iterate a defined number of times):

```
sequence <- seq(0,50,5)

#Note this will iterate 11 times, the number of elements in "sequence"
for(i in 1:length(sequence)){
  print(paste("Now at iteration", i, ", value of sequence is", sequence[i]))
  Sys.sleep(0.50)
}
```

By default, R will iterate over all the elements in a vector without needing to designate a number

```
for(value in sequence){
  print(paste("Value of sequence is", value))
  Sys.sleep(0.50)
}
```

With the break statement, we can stop the loop before it has looped through all the items:

```
for(i in 1:length(sequence)){
  if(i>5) break
  print(paste("Now at iteration", i, ", value of sequence is", sequence[i]))
  Sys.sleep(0.50)
}
```

With the next statement, we can skip an iteration without terminating the loop:

```
for(i in 1:length(sequence)){
  if((i %% 2)!=0) next
  print(paste("Now at iteration", i, ", value of sequence is", sequence[i]))
  Sys.sleep(0.50)
}
```

The while loops are used to loop until a specific condition is met:

```
while (test_expression)
{
statement
}
```

Here, test_expression is evaluated and the body of the loop is entered if the result is TRUE. The statements inside the loop are executed and the flow returns to evaluate the test_expression again. This is repeated each time until test_expression evaluates to FALSE, in which case, the loop exits. For example:

```
#While loops - iterate so long as the condition is true
#With while loops, it's important to set your starting conditions
continue <- TRUE
i <- 0
while(continue==TRUE){
  i = i + 1
  print(paste("At iteration", i, "continue still set to", continue))
  if(i>=5){
    continue <- FALSE
    print(paste("Iteration", i, "reached. Continue set to", continue))
    print(paste("While loop terminated upon reaching iteration", i))
  }
  Sys.sleep(0.50)
}
```

A repeat loop is used to iterate over a block of code multiple number of times. There is no condition check in repeat loop to exit the loop. We must ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. Failing to do so will result into an infinite loop.

```
repeat {
statement
}
```

For example:

```
x <- 1
repeat {
print(x)
x = x+1
if (x == 6){
break
}
}
```

Finally, we can nest loops of the same type or different types - for example, for every value of i it will do a loop of n:

```
continue <- TRUE
i <- 0

while(continue==TRUE){
  i = i + 1
  print(i)
  if(i>=5) continue <- FALSE

  for(n in 1:3){
    print(paste("The value of i is", i, ",and the value of n is", n))
    Sys.sleep(0.25)
  }

}
```

3.4 Functions

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

An R function is created by using the keyword function. The basic syntax of an R function definition is as follows:

```
function_name <- function(arg_1, arg_2, arg_n) {
   Function_Body
}
```

An example of a custom function is the following:

```
numbers <- seq(1:10)

calculator <- function(data, type){
  if(type=="mean"){
    value <- mean(data)
  } else if (type=="sum"){
    value <- sum(data)
  } else if (type=="min"){
    value <- min(data)
  } else {
    value <- "function not in calculator"
  }
  return(value)
}

calculator(numbers, "mean")
calculator(numbers, "sum")
calculator(numbers, "min")
calculator(numbers, "max")
```

To provide a default in the parameters, use the equal sign:

```
calculator <- function(data, type='mean'){
  if(type=="mean"){
    value <- mean(data)
  } else if (type=="sum"){
    value <- sum(data)
  } else if (type=="min"){
    value <- min(data)
  } else {
    value <- "function not in calculator"
  }
  return(value)
}

calculator(numbers)
```

If we want to return multiple values, we need store them in a vector or list and return that object:

```
univariate_summary <- function(data){
  ntot <- length(data)
  mean <- mean(data)
  std <- sd(data)
  percentiles  <- quantile(data, probs=c(0, 0.25, 0.50, 0.75, 1))

  l1 <- list(n_obs=ntot, mean=mean, std=std, percentiles=percentiles)
  return(l1)
}

results <- univariate_summary(numbers)
results$percentiles
results$std
```