Chapter 3

Conditional Statements, Loops, and Functions in R

In this chapter, we discuss several basic programming tools that will allow us to write `R` scripts, a list of commands that are executed to produce output.

3.1 Conditional Statements

It is often useful to complete one of several tasks conditional on the value of certain objects or under certain conditions. There are a variety of ways to program conditional statements in `R` and we delineate the syntax of two of them below – the shorthand notation and the block notation.

Shorthand notation:

```
ifelse(condition, code to run if condition is true, code to run if condition is false)
```

Block notation:

```
if(condition1){
  #code to run if condition 1 is true
} else if(condition2){
  #code to run if condition 1 is false and condition 2 is true
} else{
  #code to run if condition 1 and condition 2 are false
}
```

In `R`, we have the following conditional statements for block notation:

- Use `if()` to run a block of code, if a condition is true
- Use `else if()` to specify a new condition to test, if the first condition is false
- Use `else` to run a block of code, if the all specified conditions are false

Remark: The `else` part of the block notation is optional. This creates an interesting aspect of the notation because when an `if()` statement is finished, `R` won't expect an `else` statement to follow; e.g., the following would yield an error.

Incorrect block notation:

```
if(condition1){
  #code to run if condition 1 is true
}
else if(condition2){
  #code to run if condition 1 is false and condition 2 is true
}
else{
  #code to run if condition 1 and condition 2 are false
}
```

The ability to program such conditional statements is important because they allow us to tell `R` how to make decisions while running the script. This is done by telling `R` what to run based on logical objects; as a reminder, a few examples of such logical conditions are seen below.

```
> x=5
> x<3 #"less than three"
[1] FALSE
> x<=3 #"at most three"
[1] FALSE
> x>3  #"greater than three"
[1] TRUE
> x>=3 #"at least three"
[1] TRUE
> x==3 #"equal to three" note the double equals sign"
[1] FALSE
> x%%2==0 #"is even" note this says the remainder is zero when divided by 2"
[1] FALSE
> is.numeric(x) #"is x of numeric type?"
[1] TRUE
> is.character(x) #"Is x of character type?"
[1] FALSE
```

Example 3.1. Fox et al. (2018) provide data on 30,351 responses to the U.S. General Social Surveys from 1972-2016. Each observation contains the year, sex, education (in years) and a vocabulary test score revealing the number correct out of ten. Suppose that the researchers wanted to create a object that is 1 when the participant passed the vocabulary test and 0 when they failed; i.e., 1 when they answered more than half of the questions successfully and 0 otherwise.

Below, we create this vector four ways.

```
> library("carData")
> data("Vocab")
> head(Vocab)
year    sex education vocabulary
```

```
19740001 1974   Male        14          9
19740002 1974   Male        16          9
19740003 1974 Female        10          9
19740004 1974 Female        10          5
19740005 1974 Female        12          8
19740006 1974   Male        16          8
#Using the which() function -- starting vector of 0s
> pass1<-rep(0,nrow(Vocab))
> pass1[which(Vocab$vocabulary>=6)]<-1
> pass2<-rep(0,nrow(Vocab))
> pass2[which(Vocab$vocabulary>5)]<-1
#Using the which() function -- starting vector of 1s
> pass3<-rep(1,nrow(Vocab))
> pass3[which(Vocab$vocabulary<6)]<-0
> pass4<-rep(1,nrow(Vocab))
> pass4[which(Vocab$vocabulary<=5)]<-0
#Using the ifelse() function
> pass5<-ifelse(Vocab$vocabulary>=6,1,0)
> pass6<-ifelse(Vocab$vocabulary>5,1,0)
```

Asking for a table of the data shows that roughly 62%, or 18,668 of 30,351 participants, were able to pass the vocabulary test.

```
#Tabulate frequencies
> table(pass1)
pass1
0     1
11683 18668
#Tabulate proportions
> prop.table(table(pass1))
pass1
0         1
0.3849297 0.6150703
```

Example 3.2. Venables and Ripley (2002) provide data on 332 women of Pima Indian heritage who were at least 21 years old and living near Phoenix, Arizona. Each woman was tested for diabetes according to World Health Organization criteria and their result was recorded along with data about their age, the number of pregnancies they've had, plasma glucose concentration, diastolic blood pressure (mm Hg), tricep skin fold thickness (mm), bmi (kg/m^2), and information about their family history. Suppose the researchers wanted a table of BMI categories, defined as follows, and the diabetes test results.

- BMI less than 18.5 falls in the underweight range.
- BMI of at least 18.5 but less than 25 falls in the normal range.
- BMI of at least 25 but less than 30 falls in the overweight range.
- BMI of at least 30 falls in within the obese range.

We create the required object for an observation of 23.4.

```
> bmi<-23.4
> if(bmi<0){
+   category<-NA
+ }else if(bmi<18.5){
+   category<-"Underweight"
+ }else if(bmi<25){
+   category<-"Normal"
+ }else if(bmi<30){
+   category<-"Overweight"
+ }else if(bmi>=30){
+   category<-"Obese"
+ }
> category
[1] "Normal"
```

To create the vector of BMI categories we're looking for we have to apply this conditional statement to the entire vector of BMI measurements. We will delineate three ways of doing this, one in each of the following two subsections.

3.2 Loops

In this section, we will explore methods for repeating a sequence of commands. This allows us to automate the repeated running of code chunks, which is quite helpful. There are a variety of ways to program a loop in R to do something repeatedly and we delineate the syntax of three of them below.

The for loop is useful for a pre-determined number of iterations and it will only loop that specific number of times; e.g., to iterate through a vector of data from start to finish. A repeat or while loop, however, will iterate repeatedly until a pre-determined condition occurs. This condition may happen immediately, or it may require an undetermined of iterations. The important distinction here is that in a for loop the number of iterations is governed by a number and in a repeat or while loop it is determined by a condition.

for loop

```
for(i in set){
   #do these things
}
```

repeat

```
repeat{
   #do these things
if(condition){
  break
}
}
```

The break command ends iteration. Although the breaking condition is required for the repeat loop, the break command can be used in any of the looping syntax.

while

```
while(condition){
  #do these things
}
```

Remark: The repeat and while loops differ in their iteration structure. The while loop will only enter the first iteration if the controlling condition is TRUE and the repeat loop will enter the first iteration even if the controlling condition is false to start. Of course, this depends on where we place the breaking condition. If the breaking condition of a repeat loop is put at the beginning of the code to be iterated, the repeat loop will behave just as a while loop does.

Continuing Example 3.2: Recall, we wanted to create a vector of BMI categories based on the BMI measurements in the data. We've created a conditional statement but we want to perform his conditional statement repeatedly, once for every observation. This task calls for the for loop as we want to complete something a fixed amount of times, $n = 332$ times here.

```
# load data
> library("MASS")
> data("Pima.te")
> length(Pima.te$bmi)#332 sample size
> category<-rep(NA,length(Pima.te$bmi)) #a blank vector for filling
> for(i in 1:length(Pima.te$bmi)){#for each BMI observation
+   bmi=Pima.te$bmi[i] #set bmi to observation i
+   if(bmi<0){
+     cat<-NA #negative bmi values are not possible
+   }else if(bmi<18.5){
+     cat<-"Underweight"
+   }else if(bmi<25){
+     cat<-"Normal"
+   }else if(bmi<30){
+     cat<-"Overweight"
+   }else{  #otherwise bmi>=30
+     cat<-"Obese"
+   }  #end else if block
+   category[i]=cat #save the BMI category for observation i
+ }  #end for loop
> table(category,Pima.te$type)

category      No Yes
Normal       36   0
Obese       124  90
Overweight   63  19
#calculates proportions conditional on bmi category
> prop.table(x=table(category,Pima.te$type),margin=1)

category            No        Yes
```

```
Normal     1.0000000 0.0000000
Obese      0.5794393 0.4205607
Overweight 0.7682927 0.2317073
```

This table shows that roughly 42% of those in the sample categorized as "obese" were diagnosed with diabetes compared to 0% and roughly 23% of the "normal" and "overweight" categories, respectively.

Example 3.3. Outliers are observations that deviate from the general pattern. Data drawn from the standard Gaussian or normal distribution – the bell-shaped distribution – is known to have outlier probability of approximately 0.27%. An outlier in this scenario is any observation with a magnitude greater than 3; e.g., any value less than -3 or greater than 3.

To convince ourselves this is true, we will generate standard normal data until we have 100 outliers. Note that our iteration is governed by the condition that we've found 100 outliers which makes it a candidate for the repeat or while loop.

The repeat loop for this activity is as follows.

```
> nTrial<-0 #A object for counting trials
> nOutlier<-0 #A object for counting outliers
> repeat{
+   nTrial<-nTrial+1 #count the trial at the beginning of each iteration
+   x<-rnorm(1,0,1) #generate random normal data
+   if(x < -3){ #first outlier condition
+     nOutlier<-nOutlier+1
+   }else if(x>3){ #second outlier condition
+     nOutlier<-nOutlier+1
+   }else{ #otherwise -3<=x<=3 (not an outlier)
+     #do nothing, the observation is not outlying
+   }
+   if(nOutlier==100){ #stopping condition
+     break
+   }
+ }
> nOutlier
[1] 100
> nTrial
[1] 39275
> nOutlier/nTrial #True proportion is about 0.0027
[1] 0.002546149
```

The equivalent while loop only requires a small syntax change.

```
> nTrial<-0 #A object for counting trials
> nOutlier<-0 #A object for counting outliers
> while(nOutlier<100){ #complete while the count of outliers is less than 100
+   nTrial<-nTrial+1 #count the trial at the beginning of each iteration
+   x<-rnorm(1,0,1) #generate random normal data
+   if(x < -3){ #first outlier condition
+      nOutlier<-nOutlier+1
+    }else if(x>3){ #second outlier condition
+      nOutlier<-nOutlier+1
+    }else{ #otherwise -3<=x<=3 (not an outlier)
+      #do nothing, the observation is not outlying
+    }
+  }
>  nOutlier
[1] 100
>  nTrial
[1] 31642
>  nOutlier/nTrial #True proportion is about 0.0027
[1] 0.003160356
```

Here, we can see motivation for using a repeat or while loop as both loops required a different number of trials. In fact, every time either loop is executed we will get slightly different results due to the randomly generated data. These looping functions allow us to get away with solving the problem without running each iteration manually.

There may be cases where we want to skip an iteration of the loop. This is useful if there are some restrictions under which we would not want to run the rest of the code in the loop.

Example 3.4. Consider the power digit sum of powers of two. For example,

$$2^8 = 256 \longrightarrow 2+5+6 = 13.$$

Let's find the largest digit sum of powers of two, 2^k for $k = 1, 2, ..., 20$, under the restriction that the digit sum is less than less than 25.

```
> library("stringr") #This package has a tool for separating digits into a vector
> max.digit.sum<-0
> max.k<-0
> for(k in 1:20){
+   current.power<-2^k
+   #Split the power into a vector of digits
+   #simplify=TRUE ensures a vector, not a list, is returned
+   digits<-as.numeric(str_split(string=current.power,pattern="",simplify=TRUE))
+   current.digit.sum<-sum(digits)
+   if(current.digit.sum>=25){
+     next  #skips to next loop because restriction is transgressed
+   }
+   if(current.digit.sum>max.digit.sum){
+     max.digit.sum<-current.digit.sum
+     max.k<-k
+   }
+ }
> max.k
[1] 14
> max.digit.sum
[1] 22
```

Remark: Again, in R there are many paths to the same place. Below, we've rewritten the loop without the next command.

```
> max.digit.sum<-0
> max.k<-0
> for(k in 1:20){
+   current.power<-2^k
    #Split the power into a vector of digits
    #simplify=TRUE ensures a vector, not a list, is returned
+   digits<-as.numeric(str_split(string=current.power,pattern="",simplify=TRUE))
+   current.digit.sum<-sum(digits)
+   if(current.digit.sum>max.digit.sum&current.digit.sum<25){
+     max.digit.sum<-current.digit.sum
+     max.k<-k
+   }
+ }
> max.k
[1] 14
> max.digit.sum
[1] 22
```

3.2.1 Summary of R Commands

Table 3.2.1 summarizes the operators used in this subsection.

Operator	Functionality
for(...)	completes a set number of iterations
repeat{...}	repeats code – must break this loop
while(...)	repeats code while condition is true
break	stops the iterations of a loop
as.numeric()	converts x to its numeric value if possible
str_split(...)	splits number into vector of digits
next	skips to the next iteration of a loop

Table 3.2.1: A list of basic operators for loops in R.

3.3 Creating Functions

A function in R, just as in math, can take input and produce output. Of course, we've already used functions, like when asking for a square root.

```
> sqrt(16) #sqrt() is a function, x=16 is input
[1] 4
```

The square root function, `sqrt()`, takes at least one numeric or complex value as input and returns as much output; note that this function only returns the positive square root.

In the following example we will create a function called `nsqrt`, that takes one number as input and returns one number as output. The general structure of a function in `R` is as follows.

```
new.function<-function(arguments){       #tell R we're creating a new function
                                    # code that processes arguments to output
                                    # return the output of the function based on arguments called
                                    }      #end function call
```

Example 3.5. Now, suppose we wanted to write a function that returns the negative square root for non-negative real input, and an error to the user when they provide negative real input.

```
> nsqrt<-function(x){ #nsqrt will give the negative square root of positive real x
+  if(x>=0){ #the square root does not exist for negative values
+    return(-sqrt(x)) #return the negative root to the user
+  }else{ #the user has asked for the square root of an inappropriate in
+    stop("The square root of a negative number produces non-real solutions.")
+  }
+}
> nsqrt(16) #nsqrt() is a function, x=16 is input
[1] -4
> nsqrt(-1)
Error in nsqrt(-1) :
The square root of a negative number produces non-real solutions.
```

When we ask for the negative square root of a negative value, we receive the error message we specified using the `stop()` function.

Example 3.6. Now, suppose we wanted to write a function that returns the negative and positive square root for non-negative real input, and an error to the user when they provide negative real input.

```
#bothsqrst will give both the negative and positive square root of positive real x
> bothsqrst<-function(x){
+   if(x>0){
+     c(-sqrt(x),sqrt(x))
+   }else{
+     stop("The square root of a negative number produces non-real solutions.")
+   }
+ }
> bothsqrts(16) #bothsqrts() is a function, x=16 is input
[1] -4  4
> bothsqrst(-1)
Error in bothsqrst(-1) :
The square root of a negative number produces non-real solutions.
```

Example 3.7. Now, suppose we wanted to write a function that returns the both the negative and positive square roots for non-negative real input, both the negative and positive complex roots for complex input (with a warning), and a error to the user when they've provided problematic input that is neither numeric or complex.

```
#gensqrt will give the negative and positive square root of any input x
> gensqrt<-function(x){
+   if(is.complex(x)){
+     return(c(-sqrt(x),sqrt(x)))#sqrt() works for complex input
+   }
+   if(is.numeric(x)){
+     if(x>0){
+       return(c(-sqrt(x),sqrt(x)))#sqrt() works for positive real input
+     } else{
+       warning("The square root of a negative number yields a complex solution.")
#we must convert numeric to complex for negative real input
+       return(c(-sqrt(as.complex(x)),sqrt(as.complex(x))))
+     }
+   }
+   if(!is.numeric(x)&!is.complex(x)){
+     stop("The gensqrt function requires complex or real input.")
+   }
+ }
> gensqrt(16)
[1] -4  4
> gensqrt(-1)
[1] 0-1i 0+1i
Warning message:
In gensqrt(-1) :
The square root of a negative number yields a complex solution.
> gensqrt("hello")
Error in gensqrt("hello") :
The gensqrt function requires complex or real input.
```

Remark: This function goes beyond those in Examples 3.5 and 3.6 to catch and provide helpful error messages for even nonsensical input from a user; e.g., "hello."

Example 3.8. Now, suppose we wanted the function in Example 3.7 to return the positive square root for real positive input by default, but the capability of returning other types when asked. In this example, we will be more succinct with our code since the logic is seen in the previous example; this is necessary for keeping the code reasonably short.

```
> gensqrt2<-function(x,type="positive"){
+   if(!is.numeric(x)&!is.complex(x)){
+     stop("The gensqrt function requires complex or real input.")
+   }else{
+     if(is.numeric(x) && x<0){Checks if numeric first
+       x<-as.complex(x)
+       warning("The square root of a negative number yields a complex solution.")
+     }
+     if(type=="positive"){
+       return(sqrt(x))
+     }else if(type=="negative"){
+       return(-sqrt(x))
+     }else if(type=="both"){
+       return(c(-sqrt(x),sqrt(x)))
+     }else{
+       stop("The type argument must be ‘positive’, ’negative’, or ’both’.")
+     }
+   }
+ }
> gensqrt2(16,type="both")
[1] -4  4
> gensqrt2(25,type="negative")
[1] -5
> gensqrt2(-1,type="positive")
[1] 0+1i
Warning message:
In gensqrt2(-1) :
The square root of a negative number yields a complex solution.
> gensqrt2("hello")
Error in gensqrt2("hello") :
The gensqrt function requires complex or real input.
```

In this example, we see a couple of new things. First, `type` is “positive” by default; this is done by specifying the argument as positive in the `function` command. Thus, if we don’t specify `type` we will receive the positive square root by default.

```
> gensqrt2(-1)
[1] 0+1i
Warning message:
In gensqrt2(-1) :
The square root of a negative number yields a complex solution.
```

We also used && for the first time. This is very important, and very different from &. Using the && command allows us to condense two conditional blocks into one and is necessary because the inequality with 0 yields an error for complex objects. If the input is complex, then we don’t check if it’s less than zero because `is.numeric(x)` is FALSE. If the input is numeric, then we check if $x < 0$ we convert x to be complex, otherwise it stays as numeric.

```
> x<-3+1i
> is.numeric(x)
[1] FALSE
> x<0
Error in x < 0 : invalid comparison with complex values
> is.numeric(x) & x<0
Error in x < 0 : invalid comparison with complex values
> is.numeric(x) && x<0
[1] FALSE
```

Example 3.9. Below, we attempt to apply the square root functions created in Examples 3.5, 3.6, 3.7, and 3.8. Since all functions were designed an written for a singular numeric value, we need to vectorize them; i.e., write the functions so that they work for vectors.

```
> sqrt(c(1,4,9,16,25))
[1] 1 2 3 4 5
> nsqrt(c(1,4,9,16,25))
[1] -1 -2 -3 -4 -5
Warning message:
In if (x >= 0) { :
  the condition has length > 1 and only the first element will be used
> bothsqrst(c(1,4,9,16,25))
[1] -1 -2 -3 -4 -5  1  2  3  4  5
Warning message:
In if (x > 0) { :
  the condition has length > 1 and only the first element will be used
> gensqrt(c(1,4,9,16,25))
[1] -1 -2 -3 -4 -5  1  2  3  4  5
Warning message:
In if (x > 0) { :
  the condition has length > 1 and only the first element will be used
```

Remark: Many functions in R are vectorized, meaning that the operations can be applied to vectors; e.g., the `sqrt()` function works for the vector above.

For brevity, we will vectorize the `nsqrt()` and `gensqrt()` functions. First, we vectorize the `nsqrt` function using a loop that applies the commands of the original function to each element of the vector input.

```
> nsqrt.vectorized<-function(x){
+   output<-rep(NA,length(x))     #a place to save output
+   for(i in 1:length(x)){        #loop through each observation
+     if(x[i]>=0){                #change all x to x[i] so that each x is run
+       output[i]<-(-sqrt(x[i])) #save the output for the current x value
+     }else{
+       stop("The square root of a negative number produces non-real solutions.")
+     }
+   }
+   return(output)                #return the output for all x
+ }
> nsqrt.vectorized(16)
[1] -4
> nsqrt.vectorized(c(1,4,9,16,25))
[1] -1 -2 -3 -4 -5
```

Next, we vectorize the `gensqrt` function. Vectorizing this function is a bit more complicated because of the output; looking at the output from the original function, we see the five negative square roots followed by the positive square roots. While we can visually separate the two quite easily, the output would be difficult to use in later steps of programming. We can display the output in many ways – below, we focus on readability.

```
> gensqrt.vectorized<-function(x){
+   output<-data.frame(input=rep(NA,length(x)),negative.root=rep(NA,length(x)),
+                      positive.root=rep(NA,length(x)))
+   for(i in 1:length(x)){    #loop through each observation
+     if(is.complex(x[i])){   #change all x to x[i] so that each x is run
+       #save the output for the current x value
+       output[i,]<-c(x[i],-sqrt(x[i]),sqrt(x[i]))
+     }
+     if(is.numeric(x[i])){
+       if(x[i]>0){
+         #save the output for the current x value
+         output[i,]<-c(x[i],-sqrt(x[i]),sqrt(x[i]))
+       } else{
+         warning("The square root of a negative number yields a complex solution.")
+         #save the output for the current x value
+         output[i,]<-c(x[i],-sqrt(as.complex(x[i])),sqrt(as.complex(x[i])))
+       }
+     }
+     if(!is.numeric(x[i])&!is.complex(x[i])){
+       stop("The gensqrt function requires complex or real input.")
+     }
+   }
+   return(output)  #return the output for all x
+ }
> gensqrt.vectorized(16)
input negative.root positive.root
1    16            -4             4
> gensqrt.vectorized(c(1,4,9,16,25))
input negative.root positive.root
1     1            -1             1
2     4            -2             2
3     9            -3             3
4    16            -4             4
5    25            -5             5
```

Here, we use the same technique of vectorizing using a loop and we can see that storing the output in a dataframe greatly improves the readability of the output.

3.3.1 Summary of R Commands

Table 3.3.2 summarizes the operators used in this subsection.

Operator	Functionality
function(...)...	creates a function
return(x)	exits the function by returning x
stop(...)	exits the function in error with the provided error message
warning(...)	continues the completion of the function and returns output with the provided warning message

Table 3.3.2: A list of basic operators for writing functions in R.

3.4 The Apply Family of Functions

The apply family of functions – `apply()`, `lapply()`, `sapply()`, and `tapply()` – can be viewed as a substitution for vectorizing using a loop as in Example 3.9. These functions are described in Table 3.4.3.

Function	Description
apply()	applies a function to the rows or columns of a matrix
lapply()	applies a function each element of a vector returning a list
sapply()	applies a function each element of a vector returning a vector or matrix
tapply()	applies a function to elements of a vector grouped by the factor values provided

Table 3.4.3: A list of basic operators for objects in a R session.

Example 3.10. We demonstrate the `apply` function on the matrix `A` from Example 2.15 by asking for the average of the columns and the average of the rows.

```
> A
[,1] [,2]
[1,]    2    1
[2,]    0    8
> apply(X=A,MARGIN=1,FUN=mean) #margin=1 indicates to apply summary() to the rows
[1] 1.5 4.0
> apply(X=A,MARGIN=2,FUN=mean) #margin=2 indicates to apply summary() to the columns
[1] 1.0 4.5
```

Example 3.11. Spotting an Error in R. Below, we see an error that doesn't look like an error. When we run the `class()` function individually, we get the expected result. However, when we run the `class()` function using `apply` both columns are reported to be of character type.

```
> head(ship1dat)
iNKT Ship1
1 1.34   +/+
2 1.19   +/+
3 0.89   +/+
4 0.88   +/+
5 0.98   +/+
6 0.85   +/+
> class(ship1dat$iNKT)
[1] "numeric"
> class(ship1dat$Ship1)
[1] "factor"
> apply(X=ship1dat,MARGIN=2,FUN=class) #margin=2 indicates to apply summary() to the columns
iNKT        Ship1
"character" "character"
```

This error occurs because `ship1dat` is converted to a matrix which is atomic – all data must be of one type – so it converts all columns to type character.

```
> head(as.matrix(ship1dat))
iNKT   Ship1
[1,] "1.34" "+/+"
[2,] "1.19" "+/+"
[3,] "0.89" "+/+"
[4,] "0.88" "+/+"
[5,] "0.98" "+/+"
[6,] "0.85" "+/+"
```

This makes clear, again, that using a programming language requires us to ask precise questions. In this example, `R` is doing exactly what we asked it to do – unfortunately, that's not always what we want.

Example 3.12. Below, we revisit the task in Example 3.9 where we wanted to vectorize the functions created in Examples 3.5, 3.6, 3.7, and 3.8. Instead of using a loop, we can use the apply family of functions. Here, we want to apply a function to each element in a vector; i.e., we want to use `lapply()` or `sapply()`

The l in `lapply()` stands for list, which this function returns as output. The `sapply()` function takes the same input as `lapply()` but returns a vector. Unlike `apply()`, these functions can be used for other objects like data frames and lists, and do not require the MARGIN argument because they are applied elementwise.

```
# Recall the output of our vectorized function
> gensqrt.vectorized(c(1,4,9,16,25))
input negative.root positive.root
1     1            -1             1
2     4            -2             2
3     9            -3             3
4    16            -4             4
5    25            -5             5
> lapply(X=c(1,4,9,16,25),FUN=gensqrt)
[[1]]
[1] -1  1
```

```
[[2]]
[1] -2  2

[[3]]
[1] -3  3

[[4]]
[1] -4  4

[[5]]
[1] -5  5
> sapply(X=c(1,4,9,16,25),FUN=gensqrt)
[,1] [,2] [,3] [,4] [,5]
[1,]   -1   -2   -3   -4   -5
[2,]    1    2    3    4    5
```

We see that the output from all three approaches is the same, but look different. We have customized output in our own vectorized function, `lapply()` returns a list, and `sapply()` returns a matrix. Like many tasks in `R`, there are many paths to the same place.

Example 3.13. Below, we revist the data from Example 2.3. Before, if we wanted to summarize the data by Ship1 deletion status we used the `which()` function to subset the data and ask questions about the data separately.

```
> mean(ship1dat$iNKT[which(ship1dat$Ship1=="-/-")])
[1] 0.56375
> mean(ship1dat$iNKT[which(ship1dat$Ship1=="+/-")])
[1] 0.9677778
> mean(ship1dat$iNKT[which(ship1dat$Ship1=="+/+")])
[1] 1.041429
> tapply(X=ship1dat$iNKT,INDEX=ship1dat$Ship1,FUN=mean)
-/-       +/-       +/+
0.5637500 0.9677778 1.0414286
```

The `tapply()` function provides a succinct way of summarizing data. The benefit of this function is clear above, but it is practically necessary were we to have data with more than a handful of groups.

Continuing Example 3.2 Recall, we wanted to create a vector of BMI categories based on the BMI measurements in the data. We've seen how to do this for one observation, and then for all observations using a loop. The final and third way we'll do this is using a function and applying the function to all BMI observations.

First, we create a function that includes the conditional statement we had constructed earlier. Note, we reproduced our categorization of an observed BMI of 23.4 as normal.

```
> bmi<-function(bmi){
+   category<-NULL
+   if(bmi<0){
+     category<-NA
+   }else if(bmi<18.5){
+     category<-"Underweight"
+   }else if(bmi<25){
+     category<-"Normal"
+   }else if(bmi<30){
+     category<-"Overweight"
+   }else{
+     category<-"Obese"
+   }
+   return(category)
+ }
> bmi(23.4)
[1] "Normal"
```

We can now apply this to all BMI observations and recreate the table.

```
#apply bmi() function to all BMI observations
> category<-sapply(X=Pima.te$bmi,FUN=bmi)
> table(category,Pima.te$type) #create the table

category      No Yes
Normal       36   0
Obese       124  90
Overweight   63  19
```

We use `sapply()` to apply the `bmi()` function to each BMI observation.