Learning Goals

1. Familiarity with the concept of a function and a piecewise defined function.
2. Applying inbuilt functions in `R`.
3. Funcions in `R` with vector arguments.
4. Vectorized functions in `R`.
5. Defining a function in `R`.
6. Valid function names in `R`.
7. Assigning arguments when defining functions.
8. Syntax for use of logical operators.
9. Using `sapply()`.
10. Using `else if`

Functions in `R`

Most of the work done in R is through functions. We've already used several functions, for example; `sum(), mean()` and `plot()`. In this section we will pull out the general principles of functions and their use, and learn to create functions of our own.

Functions A function occurrs in mathematics when one variable depends on another (or several other variables). For example the level of muscle soreness a football player feels after a game is a function of how much time has elapsed since the game.

Functions of a single variable In mathematics we often work with functions that have a formula such as the line $L(x) = 2x + 1$ or the parabola $f(x) = x^2$. Both of these are examples of functions, the values of L and f depend on a single variable x. Here x is called the independent variable and L and f are called dependent variables.

We have already encountered such functions:
Example: The number of games necessary in a round robin tournament with N players is a function of N:

$$F(N) = \frac{N(N-1)}{2}.$$

Functions of more than one variable Functions may have more than one independent variable. We have already encountered example of these, when we created formulas with regression.

Examples: $f(x, y) = x^2+y^2$ describes a function of two independent variables x and y. Here $f(1, 2) = 1^2 + 2^2 = 5$.
Earlier in the course, we estimated overall win-loss percentage (OWL.P) in our basketball data by a linear function with the four factors and independent variables.

Piecewise Defined Functions Sometimes the formula for a function is complicated and requires more than one sentence or formula to describe it. These are called **piecewise defined functions**. For example the absolute value function is a piecewise defined function.

$$|x| = \begin{cases} -x & x \leq 0 \\ x & x > 0 \end{cases} .$$

Let us now turn our attention to defining functions in R.

Functions in R: R comes with a number of built in functions which we can use. When applying a function, we call it by its name followed by a pair of parentheses. $\underline{\text{The argument}}$ to which we apply the function is placed in the parentheses and if there is more than one argument, we separate them by commas. The following example applies the sqare root function to the number 9(the argument).

```
> sqrt(9)

[1] 3
```

Functions can be applied to lists and vectors. For example, the `sum` function in *R* sums whatever list of numbers we feed to it.

```
> sum(9,3)

[1] 12

> sum(93,10,54)

[1] 157
```

We can also combine our list of numbers to which we wish to apply the sum into a vector and apply the sum function to the vector, for example:

```
> x<-c(93,10,54)
> sum(x)

[1] 157
```

Similarly, we can apply the length function to a vector to find its length:

```
> length(x)
```

```
[1] 3
```

We could find the mean of the numbers in `x` either by dividing the sum of the numbers by 3 or by applying the `mean` function:

```
> sum(x)/length(x)
```

```
[1] 52.33333
```

```
> mean(x)
```

```
[1] 52.33333
```

The functions shown above `mean()`, `sum()` and `length()` are examples of **reductive functions**, which means that they take in a vector as an argument and give back a number.

Vectorized Functions: R has several functions which when applied to a vector, they give back a vector where the function has been applied to each number in the input vector. The `sqrt` function is an example of such a function:

```
> sqrt(x)
```

```
[1] 9.643651 3.162278 7.348469
```

Info on an inbuilt function When we want information about a function in R, we can search under the help menu. If we search for mean, we see that there are a number of extra arguments allowed. If we do not specify their values, R chooses a default value for us:
`mean(x, trim = 0, na.rm = FALSE, ...)`
the `trim` option allows us to trim a certain percentage of the data (equal amounts from both ends), before calculating the mean to calculated a trimmed mean. If we do not specify a value for trim, R uses the default `trim = 0`.

```
> v<-c(1,5,5,5,5,5,5,5,5,1)
> mean(v)
```

```
[1] 4.2
```

```
> mean(v, trim=0.2)
```

```
[1] 5
```

Defining a function in R: The basic template for defining a new function in R is

```
function name<- function(argument list){
function formula or description
}
```

Valid Function names: You can construct a name for a function consisting of a sequence of letters, digits, the period (".") and the underscore, with the convention that they must not start with a digit nor underscore, nor with a period followed by a digit. For example, myfunction, my.function and my_function are valid names, as is .myfunction but not _myfunction or 1myfunction.
Arguments: A function may have one argument or several(separated by commas). Functions may have may arguments of many types, numbers, vectors or even other functions. The order in which the arguments are presented is important since R will match your input to the argument in your definition. Here are some examples where we define and use the functions $f(x) = x^2$ and $g(x, y) = x^2 + y$:

```
> f <-function(x){x^2}
> f(2)

[1] 4

> g<-function(x,y){x^2 + y}
> g(1,2)

[1] 3

> g(2,1)

[1] 5
```

Notice that the value of $g(1, 2)$ is different from the value of $g(2, 1)$ because in the former $x = 1$ and $y = 2$ and in the latter $x = 2$ and $y = 1$.

Body of function: The body of the function may have a formula for the function if it exists. It may have a number of steps that R must work through to evaluate the function. The value returned by the function is the result of the last expression evaluated.

`If` **Statements** For piecewise defined functions, we must translate our piecewise description of the formula to its analogue in R; an `if` statement. Recall that the syntax in R for an `if` statement looks like this:

```
if (condition){
expression1
} else{expression2
}
```

If the condition is satisfied, expression1 is calculated, otherwise, expression2 is calculated. The condition should be expressed as a logical statement using logical operators.

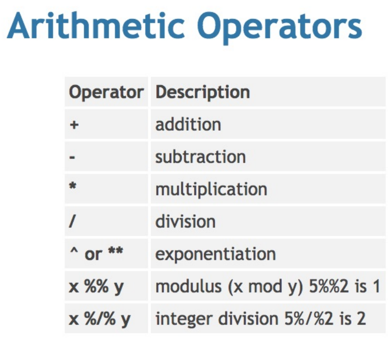
Example Consider the absolute value function with the piecewise definition:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0\ . \end{cases}$$

we can create this function in R as follows:

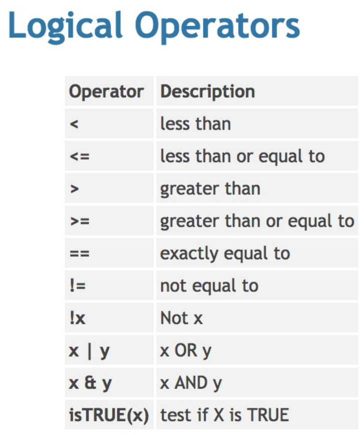
```
> abs.val <- function(x){
+   if(x>=0){x
+   }else{
+     -x
+   }
+ }
> abs.val(-2)

[1] 2
```

As you can see the syntax for the condition $x \geq 0$ in R is $x >= 0$ and that for $x < 0$ is the same $(x < 0)$ in R. The following table gives a list of the syntax in R for common arithmetic operators and logical operators in R that can be used to create logical expressions matching our conditions:



Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2



Logical Operators

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

One arithmetic operator you may not be familiar with is x mod y which translates to $x\%\%y$ in R. This arithmetic operator applies to integers x and y and the statement

$$x \bmod y = r$$

translates to "r is the reminder on division of x by y". For example :
$10 \bmod 4 = 2$
$100 \bmod 5 = 0$
$42 \bmod 2 = 0$
$42 \bmod 5 = 2$

Example Suppose we want to define the following function in R, which can be applied to any integer:

$$h(N) = \begin{cases} N & \text{if } N \text{ is odd} \\ N-1 & \text{if } N \text{ is even.} \end{cases}$$

Note that the statement "N is odd" is the same as saying "$N \bmod 2 = 1$" and the statement "N is even" is the same as saying "$N \bmod 2 = 0$". Thus the statement “N is odd” translates to “`N%%2 == 1`” in R and the statement “N is even” translates to “`N%%2 == 0`” in R. We can now define the function $h(N)$ above in R as follows

```
> h <- function(n){
+   if(n%%2==1){n
+   }else{
+     n-1
+   }
+ }
> h(15)

[1] 15

> h(20)

[1] 19

> z<-c(22,41,54)
> h(z)

[1] 21 40 53
```

(WRONG!!!)
Vectorizing your Function: Note that our function is not vectorized, when we apply it to a vector we get the wrong answer. We can create a new vectorized form of our function with the command `Vectorize(h)` or use the command `sapply(z, h)` to apply it to each element of the vector `z`.

```
> sapply(z, h)

[1] 21 41 53

> vh<-Vectorize(h)
> vh(z)

[1] 21 41 53
```

Example: (a) Create a function in R called my.function which is defined by the formula:

$$\text{my.function}(N) = \begin{cases} N & \text{if } N \text{ is divisible by 5} \\ 5N & \text{if } N \text{ is not divisible by 5.} \end{cases}$$

(b) Apply your function to the vector $(2, 3, 10, 4)$ using the sapply command.

(c) Create a vectorized version of your function `vmy.function` using the `Vectorize()` function in R and apply it to the above vector.

<u>`else if`</u> If our piecewise defined function requires more than two equations to describe it, we can use `else if` to describe it.

Example Consider the following function which does not change a number unless it is divisible by 5, multiplies multiples of 10 by 3 and multiplies all other multiples of 5 by 2:

$$f(n) = \begin{cases} n & \text{if } n \text{ is not divisible by 5} \\ 2n & \text{if } n \text{ is divisible by 5 but not divisible by 10} \\ 3n & \text{if } n \text{ is divisible by 10 .} \end{cases}$$

Using our logical operators we can describe the conditions above as follows in R:

- (if n is not divisible by 5) $n\%\%5 \; != \; 0$
- (if n is divisible by 5 but not divisible by 10) $(n\%\%10 \; != \; 0)\&(n\%\%5 \; == \; 0)$
- (if n is divisible by 10) $n\%\%10 \; == \; 0$

Using `else if` we can define the function in R as:

```
> f <- function(n){
+   if(n%%5!=0){
+     n
+     }else if((n%%10!=0)&(n%%5==0)){
+     2*n
+     }else{
+       3*n
+     }
+   }
> f(15)

[1] 30

> f(100)

[1] 300

> f(2)
```

```
[1] 2
```

Applying Functions in data sets to create new variables using the `apply()` family

We saw above that we could apply a function to a vector(by name) if it is vectorized. We can use the `apply()` family of functions `https://www.datacamp.com/community/tutorials/r-tutorial-apply-family` to simplify the application of functions in a data set if the function is not vectorized. Let's say we want to create a new variable, which is the result of applying the function $f(x) = x^2$ to the variable `ht` in the data frame below:

```
> ht<-c(3,4,5,6)
> wt<-c(1,2,3,4)
> age<-c(7,8,9,10)
> df<-data.frame(ht,wt,age)
> df

  ht wt age
1  3  1   7
2  4  2   8
3  5  3   9
4  6  4  10
```

Since this is just a a function of a single variable(not vectorized), I can apply it to the column `ht` using the `sapply` command, with arguments `(variable,function)`. I can add it to the data frame in one command;

```
> f<-function(x){x^2}
> df$new<-sapply(df$ht,f)
> df

  ht wt age new
1  3  1   7   9
2  4  2   8  16
3  5  3   9  25
4  6  4  10  36
```

If I wish to add ane variable of the form $(wt)^2 + 2*(age)$, then I can use `mapply` with arguments `(function, variables(in correct order))`

```
> g<-function(x,y){x^2+2*y}
> df$new2<-mapply(g,df$wt,df$age)
> df
```

```
  ht wt age new new2
1  3  1   7   9   15
2  4  2   8  16   20
3  5  3   9  25   27
4  6  4  10  36   36
```

If I have a vectorized function, such as `mean()` that I wish to apply to the columns, I can do this with the `apply()` function by specifying the margin (2 for columns, 1 for rows).

```
> apply(df,2,mean)#the margin specified is 2,

  ht   wt  age  new new2
 4.5  2.5  8.5 21.5 24.5

>                  #which indicates that we are applying the function to
>                 #the columns
>
> #To apply to the rows, we set margin equal to 1
> apply(df,1,mean)

[1]  7.0 10.0 13.8 18.4
```

Loops and if statements revisited: We used loops to fill up the Colley and Massey matrices in previous lectures. We can also use loops to create a new variable in a data set. The syntax above helps us create the `if` statements when the description of the variable is complicated.

Example (Weighting Games): In the `Games.csv` file, downloaded from Massey's site, we would like to create a new variable `W`, which assigns a weight of 2 to a game that resulted in an away win. There are a number of ways to do this.

Method 1. We could use our already familiar pattern of creating the variable with every value equal to 0 first. Then, we will use a for loop to run through the lines of the file and replace the values of the variable one-by-one with the correct value.

```
> g<-"https://www.masseyratings.com/scores.php?s=305972&sub=10423&all=1&mode=2&sch=
> download.file(g,"Games.csv","curl")
> A<-read.csv("Games.csv", header=FALSE)
> names<-c("days", "YYYYMMDD","team1","homefield1", "score1", "team2", "homefield2"
> colnames(A)<-names
> head(A)
```

```
    days YYYYMMDD team1 homefield1 score1 team2 homefield2 score2
1 737425 20190101    14          1     81    10         -1     66
2 737427 20190103     8         -1     87     7          1     82
3 737429 20190105    14          1     77     1         -1     66
4 737429 20190105     9         -1     85    11          1     60
5 737429 20190105    12         -1     72    10          1     62
6 737429 20190105     3          1     87     2         -1     68
```

First, we create a variable that equals 0 for all games:

```
> A$W=0
> head(A)
```

```
    days YYYYMMDD team1 homefield1 score1 team2 homefield2 score2 W
1 737425 20190101    14          1     81    10         -1     66 0
2 737427 20190103     8         -1     87     7          1     82 0
3 737429 20190105    14          1     77     1         -1     66 0
4 737429 20190105     9         -1     85    11          1     60 0
5 737429 20190105    12         -1     72    10          1     62 0
6 737429 20190105     3          1     87     2         -1     68 0
```

Now we tell R to run through the file, line-by-line and make the value of W equal to 2 if the game was an away win and 1 otherwise. Since Kenneth Massey always puts the winning team first, an away win can happen only if the value of homefield1 is -1. We need to translate this scenario to a logical description using the syntax in the table above.

```
> games<-nrow(A)
> for(i in 1:games){
+   if(A$homefield1[i]==-1){A$W[i]<-2}
+   else{A$W[i]<-1}
+ }
> head(A,10)
```

```
     days YYYYMMDD team1 homefield1 score1 team2 homefield2 score2 W
1  737425 20190101    14          1     81    10         -1     66 1
2  737427 20190103     8         -1     87     7          1     82 2
3  737429 20190105    14          1     77     1         -1     66 1
4  737429 20190105     9         -1     85    11          1     60 2
5  737429 20190105    12         -1     72    10          1     62 2
6  737429 20190105     3          1     87     2         -1     68 1
7  737429 20190105     5          1     92    15         -1     79 1
8  737429 20190105    13          1     65     4         -1     52 1
9  737430 20190106     6          1     90     7         -1     73 1
10 737432 20190108     3         -1     87    15          1     65 2
```

Method 2 We could use a function to define the weight and apply it to the data frame. Note that A[i,4] denotes the entry in the ith row and 4th column of our data frame A.

```
> w<-function(n){
+   if(n==-1)
+   {2}
+   else{1}
+ }
> A$W1<-mapply(w,A$homefield1)
> head(A)
```

```
    days YYYYMMDD team1 homefield1 score1 team2 homefield2 score2 W W1
1 737425 20190101    14          1     81    10         -1     66 1  1
2 737427 20190103     8         -1     87     7          1     82 2  2
3 737429 20190105    14          1     77     1         -1     66 1  1
4 737429 20190105     9         -1     85    11          1     60 2  2
5 737429 20190105    12         -1     72    10          1     62 2  2
6 737429 20190105     3          1     87     2         -1     68 1  1
```

Example The `print()` function prints the argument.

```
> print("I Love Math")
```

```
[1] "I Love Math"
```

Let's try to print all of the numbers between 1 and 100 which are divisible by 7, but not divisible by 3.

```
> for(i in 1:100){if((i%%7==0)&(i%%3!=0)){print(i)}}
```

```
[1] 7
[1] 14
[1] 28
[1] 35
[1] 49
[1] 56
[1] 70
[1] 77
[1] 91
[1] 98
```

Coming Up: In the next section, we will see how the syntax for logical statements/operations will help us to find subsets of our data.

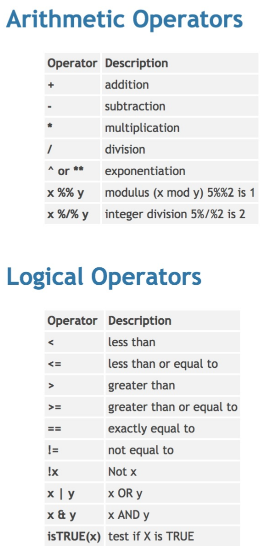
R commands

1. `sqrt()` : the square root function.
2. `sum()` : the summation function.
3. `length(x)` : gives the length of a vector `x`
4. `mean()` : gives the average.

5.
   ```
   function name<- function(argument list){
   function formula or description
   }
   ```
   : format of definition of a function.

6.
   ```
   if (condition){
   expression1
   } else{expression2
   }
   ```
   : format of an `if` statement.

7.
   ```
   if (condition 1){
   expression1
   } else if(condition 2){expression2
   } else{expression3
   }
   ```
   : format of an `if` statement with more than one condition.
8. `Vectorize(f)`: vectorize the function `f`.
9. `sapply(x,f)`: apply the function `f` to the vector `x`.
10. `mapply(f, var1,var2, ..., varn)`: apply the function `g` to the vector variables `var1,var2, ..., varn` in that order.
11. logical operators:



Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

Logical Operators

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE