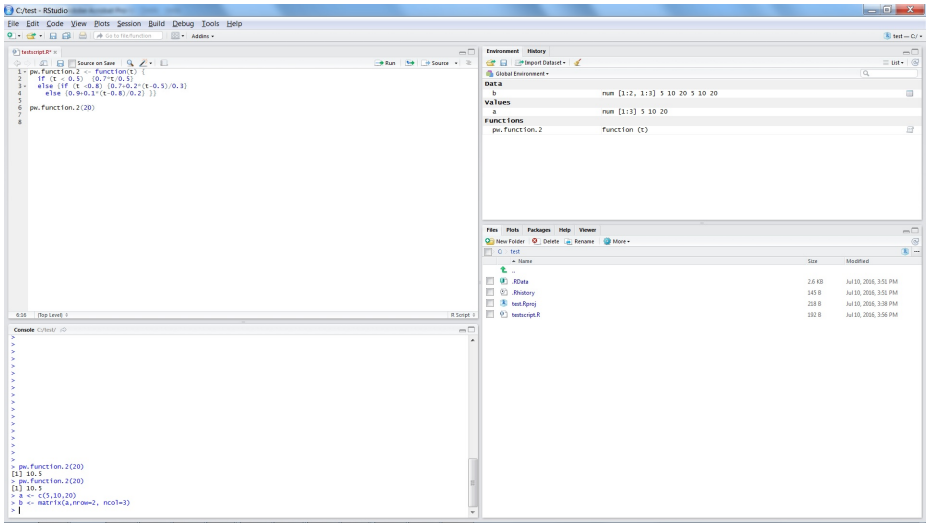
R Tutorial - Basic Concepts

Simon James, Sutharshan Rajasegarar

1 Introduction

There are many online tutorials available for becoming familiar with the programming language R. Once you have the basic commands down, you will find that an online search for how to do something specific can almost always find an existing solution either as part of standard packages, or as code that someone else has provided. We will start with some basic operations and then move toward being able to perform the machine learning and pattern recognition activities that we came across in this unit.

You can use RStudio, which provides a user friendly IDE for R Programming. It is an open source software and can be obtained from `www.rstudio.com`. Below is a screenshot of an RStudio window.



Entering commands in the console

For simple one-line commands, we can enter these straight into the console by typing the command and pressing return/enter.

If we want to retrieve a previous command, we can press the up arrow.

Basic mathematical operations

Elementary functions and basic arithmetic operations are all part of `R`. In most cases, these are the same as what you may have used before in programs like Microsoft Excel.

R Exercise 1 *Trying entering in the following commands (the expected results are shown for the numerical examples given).*

Operation	*Symbol*	*in R*	*Example*	*in R*	*Expected result*
Addition	$+$	$+$	$5 + 2$	*5+2*	*7*
Subtraction	$-$	-	*6 - 8*	*6 - 8*	*-2*
Multiplication	$\times$	*	3×4	*3*4*	*12*
Division	$\div$	/	$3 \div 4$	*3/4*	*0.75*
Powers	$(\cdot)^p$	^	3^4	*3^4*	*81*

Assignment of variables

We are often going to be working with general operations in terms of variables and in some cases we will require operations on vectors.

We can use almost any word or letter for a variable along with the full stop symbol `"."` and underscore `"_"`. We can't use the dash (because it treats it as a minus sign) and we can't start with a number and there are some special words we can't use *for functions* because they are already used by `R` for other things (we can check this by typing the function and an open bracket, e.g. `max(` and seeing whether the function parameters come up in the bottom status bar.

To assign a value to a variable we use the *less than* symbol '<' and the dash '–' together (it makes an arrow). So if we want to set the value of a variable a to 3, we write

```
a <- 3
```

Try doing this and see what happens when you enter the command $a + 4$.

R Exercise 2 *Assign the following values to each variable.*

```
a <- 3
b <- 7
the.value <- 12
the.index <- 2
the.index_2 <- 3
```

Now evaluate:

```
a*b
the.value * a
a^the.index + b^the.index_2
```

Assigning a vector to a variable

In order to assign a vector to a value, we use a 'c' along with the entries of the vector inside normal brackets, separated by commas. For example,

```
a <- c(3,2,1,0)
```

We have some other special commands that we can also use. If we want a vector of a set of sequential numbers, e.g. $\langle 4, 5, 6, 7, 8 \rangle$ we can write the starting and finishing numbers separated by a colon.

```
a <- 4:8
```

Sometimes we will need to first define an empty vector, or a vector with default starting entries. This is more easily achieved using the `array()` command in R. This command has two arguments. The first one is the default entry of the array, e.g. 0, while the second argument gives the length. For example,

```
a <- array(0,3)
b <- array(1,10)
long.array <- array(2,200)
```

We can also assign arrays that have a small sub-sequence. For example, suppose we want the vector $\langle 1,2,3,1,2,3,1,2,3\rangle$. We can assign this using a combination of the `array()` and `c()` commands.

```
my.seq <- array(c(1,2,3),200)
```

Notice that in this case, the `c(1,2,3)` acts as just one argument in the array command, even though it's a set of 3 numbers.

Basic operations on vectors

All of the basic operations we discussed, i.e. `+, -, *, /, ^`, can be used on vectors. If we are operating with a vector and a normal number value (e.g. 3) these are calculated component-wise.

R Exercise 3 *Enter the following commands from the input column and check the results.*

Input	*Expected output*
`c(1,2,3) + 3`	*4 5 6*
`c(1,2,3)*4`	*4 8 12*
`c(1,2,3)^2`	*1 4 9*

We can also have operations between vectors. In most cases, these operations should be between vectors of the same length. The operation is applied between each of the corresponding components, for example

$$\langle 2,3\rangle + \langle 5,-1\rangle = \langle 2+5, 3+(-1)\rangle = \langle 7,2\rangle.$$

R Exercise 4 *Assign the following two vectors,*

```
a <- c(1,6,7,9)
b <- c(-1, 2, 1, -2)
```

Now check the following commands against the expected output.

Input	*Expected output*
`a+b`	*0 8 8 7*
`a - b`	*2 4 6 11*
`a*b`	*-1 12 7 -18*
`a/b`	*-1 3 7 -4.5*
`a^b`	*1 36 7 0.01234568*

Existing functions

The mean and the median are already pre-programmed into `R`. We can calculate the arithmetic mean of a set of numbers using `mean()` where the argument is a vector. For example,

```
mean(c(2,3,6,7))
mean(2:7)
mean(my.seq)
mean(c(4,a,1:3,the.value,long.array))
```

Notice that in this last example, we could combine numbers, pre-assigned values and vectors to use as the input. We just need to remember to use `c()` and separate the values by a comma. The `mean()` function has some other optional parameters but we will not use these. If we entered something like `mean(3,2,51,2)`, the value returned would only be based on the first number, i.e. 3, so we need to be careful.

The median function is also pre-programmed into R.

```
median(1:6)
median(c(1,7,82,2))
median(my.seq)
```

Some other pre-programmed functions that will be useful for us are `sum()`, `prod()` and `length()` which return the sum of the entries of the vector, the product of the values in the vector, and the length of the vector respectively.

We also have `min()` and `max()` functions which can take multiple arguments including a combination of single values or vectors and return the minimum and maximum arguments.

These can also be combined with the previous operations we looked at, so `sum(a^2)` would first square every value in a, and then add all these values together.

R Exercise 5 *For the following vectors,*

```
a <- c(1,8,3,9)
b <- c(2,2,1,1)
d <- c(3,4,6,81,9)
```

Evaluate:

Input	*Expected output*
`sum(a)`	*21*
`prod(c)`	*52488*
`length(b)`	*4*
`sum(a*b)`	*30*
`sum(a^b)`	*77*
`prod(a)^(1/length(b))`	*3.833659*
`min(d)`	*3*
`max(a,d)`	*81*
`min(max(a),max(b))`	*2*

Side Note 1.1 *Make sure you are careful with your brackets! The commands follow order of operations, which means without brackets in something like* `prod(a)^(1/length(b))`*, i.e. if you just wrote* `prod(a)^1/length(b)` *then we would have the product to the power of 1 (which is just the product), which is then divided by its length, so it would be* $(1 \times 8 \times 3 \times 9)/4 = 54$.

Creating new functions

In many situations it is convenient for us to be able to program our own functions. This is actually relatively easy to do in R and can become a very powerful tool!

Creation of a basic function essentially consists of 3 components:

- Pre-defining the function inputs
- A sequence of calculations
- Return of an output

We can start with an example that calculates the arithmetic mean (even though we don't need it because it is already defined in R with `mean()`).

```
our.mean <- function(x) {
sum(x)/length(x)
}
```

The value returned (i.e. the output) is always the calculation on the last line of our function before the } bracket. In this case, our whole calculation fits on one line, so we don't need to do any temporary assignment of values throughout. However we could if we wanted to. Here is another example that will return the same output.

```
our.mean.2 <- function(x) {
n <- length(x)
s <- sum(x)
output <- s/n
output
}
```

In both of these examples, the `x` input is expected to be a vector. We could now calculate the mean using these `R` functions, e.g. by entering `our.mean(a)` or `our.mean(c(38,27,1))`. You will notice that the variables `n` and `s` are not saved in the R workspace (i.e. if you type 'n' and press enter, R will say that 'n' is not found.

> **Side Note 1.2** *Usually in programming we might be concerned about the efficiency of our steps. We won't focus on this at the moment, although it definitely can be important when we are dealing with large datasets.*

If we do an assignment on the last line of the function, no output will be returned, e.g.

```
nothing.happens <- function(x) {
sum(x) +2
a <- prod(x)
}
```

R will still make these calculations when you use the function, but it won't return any values.

> **Side Note 1.3** *So far we have only considered one input into the function (although it is a multi-argument vector). In future topics we may need some other parameters.*

Let's now program functions for computing geometric and harmonic means, given by the following formulae.

Geometric mean

$$\mathrm{GM}(\mathbf{x}) = \left(\prod_{i=1}^{n} x_i\right)^{1/n} = (x_1 x_2 \cdots x_n)^{1/n} \tag{1}$$

Harmonic mean

$$\mathrm{HM}(\mathbf{x}) = n\left(\sum_{i=1}^{n} \frac{1}{x_i}\right)^{-1} = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}} \tag{2}$$

```
GM <- function(x) {
prod(x)^(1/length(x))
}

HM <- function(x) {
length(x)/sum(1/x)
}
```

> **Side Note 1.4** *When functions have just one line of calculation, there is no need for us to separate them like this. We use this format for clarity, however if we wanted to, we could enter the harmonic mean using*
>
> `HM <- function(x) { length(x)/sum(1/x)}`
>
> *and the function would be the same. In some future functions,*

however, we will need several lines of calculation. When entering multi-line functions in R, you can either do this directly in the command console (a '+' symbol will appear to let you know that you haven't closed off the necessary brackets), or you can hold the function in a library file and then copy and paste the whole sequence to the command console, or alternatively in support applications like RStudio there are other ways to execute the code.

Matrices and Transformations

Here we will start working with whole matrices of data and learn how to implement transformations.

Replacing values

We learnt how to assign vectors and values in the previous topic. Sometimes we just want to change one entry. For changing the value of one entry in a vector, we can use the assign command and indicate the index in square brackets, e.g. to change the 5th entry in a vector a we would enter

```
a[5] <- 10
```

We can also replace multiple entries at once

```
a[c(5,7)] <- c(10,3)
a[3:5] <- c(1,0,1)
```

R Exercise 6 *Create the vector and change the entries.*

```
a <- array(0,20)
a[5] <- 1
a[c(3,7,11)] <- c(2,6,1)
a[17:20] <- c(1,2,1,4)
```

Your expected output when you type a *and press enter should now be*

```
 0 0 2 0 1 0 6 0 0 0 1 0 0 0 0 0 1 2 1 4
```

Arrays and matrices

Sometimes we will need to consider whole datasets at a time. For this, rather than just vectors, we will need rows of vectors, or matrices and arrays.

We can build them step by step with the cbind (column bind) and rbind (row bind) functions.

```
a <- cbind(c(2,3,5,1,0), c(6,1,8,2,9))
b <- rbind(c(4,1), c(2,-2),c(5,6))
```

R Exercise 7 *Assign the vectors and perform the* `cbind()` *and* `rbind` *operations.*

```
a <- c(1,2,3,7,9)
a <- cbind(a, c(21,2,1,5,6))
a <- cbind(a, c(2,-1,5,0,-1))
a <- cbind(a,c(1,9,7,2,1),array(6,5))

b <-c(3,6,1,92)
b <- rbind(b, c(3,2,1,8,9))
b <- rbind(b, c(4,1,12,1,2))
```

Note that for vectors like `c(3,2,1,8,9)`, R doesn't treat it as either a row or a column when it's by itself and so it is flexible when it comes to combining vectors either as rows or as columns. However once we have a matrix, the `rbind` and `cbind` operations will come up with a warning if the row length is not the same - however it will still merge them (it just fills the remaining space by repeating the sequence of the row).

We can also create large $m \times n$ arrays and then input values later. We will use the array function for this.

Side Note 1.5 *Arrays in* R *differ to matrices in that matrices must have numerical inputs. This can sometimes cause problems when we import our data from somewhere else (see read.table below), however for us the array function will be the most straightforward to use at this time.*

As with the array function previously, the first entry is the default value that will populate the array, however this time we will use a vector `c(rows,columns)` to indicate how many rows and how many columns there needs to be. A matrix/array with 3 rows and 4 columns (prepopulated with 0s) would be

```
A <- array(0,c(3,4))
```

Side Note 1.6 *We also can have higher dimension arrays, however for the moment we will stick to two.*

Once we've created our array with default values, we can then proceed to fill it. We refer to the cells using `A[row,column]` so that the entry in the first row and the second column would be `A[1,2]`. We can also consider entire columns, e.g. the second using `A[,2]` and entire rows using `A[1,]`. Note that this corresponds with our $\mathbf{x}_j$ and $x_{i,j}$ notation.

R Exercise 8 *Create a 3×4 array and then assign values to different entries using the following.*

```
A[3,1] <- 4
A[1,] <- c(1,2,3,4)
A[,2] <- c(6,5,4)
A[3:4,2:3] <- array(-1,c(2,2))
```

Your final matrix should appear as

```
     [,1] [,2] [,3] [,4]
[1,]    1    6    3    4
[2,]    0    5   -1   -1
[3,]    4    4   -1   -1
```

Reading a table

Often we will have data available from some other source, e.g. as an excel spreadsheet. We can import this data using the `read.table()` function. The easiest way is to have the data in a txt or csv file. We need to know whether the entries are separated by commas or spaces and whether they have labels. In the

simplest case, we can save the table to an array. If the file is in the same folder as our R working directory[1], we can use the command

```
A <- read.table("thedata.csv")
```

If the data has headers (i.e. the first row is not entries but data labels), or is separated by commas, then we can add extra options.

```
A <- read.table("thedata.csv",header=TRUE,sep=",")
```

It is easiest if our data is already numerical data. In some cases, the data can be coded as text even if it is numbers. If columns in the data are numeric and R has trouble interpreting them as numbers (e.g. it says NA when you try to add 2), they can be extracted from the table using the `as.numeric()` command. The following exercise uses the `write.table()` command as well.

R Exercise 9 *Use the following to create and write a table to the working directory*

```
write.table(cbind(c("a",1,2),c(3,2,5)),"wontwork.txt")
```

Now read the table from the file and assign it to `my.table` *using the* `read.table()` *command.*

```
my.table <- read.table("wontwork.txt")
```

See what happens when you input the following.

[1] In R studio, the working directory can be set using the Session→Set Working Directory option from the menu. In the standard R application on a Mac, the option to change the working directory is under Misc.

Input	*Expected Output*

`my.table`	`  V1 V2`
	`1  a  3`
	`2  1  2`
	`3  2  5`

`my.table+2`	`  V1 V2`
	`1  NA  5`
	`2  NA  4`
	`3  NA  7`

`my.table[2,]+2`	`  V1 V2` `2  NA  4`
`as.numeric(my.table[2,])+2`	`3  4`

Transforming variables

Simple transformations of variables can be achieved using operations such as the following.

Linear feature scaling

$$f(t) = \frac{t-a}{b}, \tag{3}$$
$$a = \min(\mathbf{x}_j), \qquad b = \max(\mathbf{x}_j) - \min(\mathbf{x}_j)$$

Standardisation

$$f(t) = \frac{t-\mu}{\sigma}, \tag{4}$$
$$\mu = \text{mean}(\mathbf{x}_j), \qquad \sigma = \text{std. dev of } \mathbf{x}_j$$

We can either choose to simply replace our values with the transformed ones or we can create a new matrix.

Let's first load the volleball data and then make a copy which we will store as `original`.

The volleyball data has the following information. Note that the file "'volley.txt"' does not have the headers and the first column of names in it.

Student	Sprint 100m (seconds)	Height (cm)	Serving (out of 100)	Endurance (out of 30)
Mizuho	15.78	148	94	17
Yukie	21.15	147	94	20
Megumi	14.30	134	91	17
Sakura	19.59	174	88	16
Izumi	10.96	145	93	16
Yukiko	19.17	158	83	12
Yumiko	18.35	157	99	20
Kayoko	14.09	177	82	23
Yuko	27.98	155	93	19
Hirono	16.51	165	85	7
Mitsuko	15.57	137	100	14
Haruka	14.16	162	93	16
Takako	22.40	176	95	15
Mayumi	21.34	153	97	9
Noriko	15.67	140	94	8
Yuka	19.12	155	81	3
Satomi	21.50	147	88	5
Fumiyo	40.29	161	95	19
Chisato	12.34	160	89	26
Kaori	13.38	134	81	16

```
V <- read.table("volley.txt")
original <- V
```

The second of these lines simply copies the table "V" and assigns the copy to "original".

The following command replaces the *Sprint* variable with transformed values according to a negation function.

```
V[,1] <- 51.24 - V[,1]
```

We've used `V[,1]` to access the whole first column and when we replace it with `51.24 - V[,1]`, each entry in the column is subtracted from 51.24. We can then transform it to the unit interval using the Linear feature scaling formula as follows:

```
V[,1] <- (V[,1]-min(V[,1]))/(max(V[,1])-min(V[,1]))
```

Be careful with brackets and make sure your operation is working correctly. Sometimes it's good to manually check the first few values that it is working.

Now let's transform the height variable using standardisation. For this, we will use the `sd()` function to calculate the standard deviation.

```
V[,2] <- (V[,2]-mean(V[,2]))/sd(V[,2])
```

> **R Exercise 10** *Use the linear feature scaling technique to get this column and the remaining columns, 3 and 4, to range between 0 and 1.*

Rank-based scores

There are three functions in R that can help us with situations where we are interested in the order of arguments in a vector. These are `sort()`, `order()` and `rank()`.

The `sort()` function re-orders a vector into increasing (or non-decreasing) order. So with the vector $\langle 1, 6, 2, 3 \rangle$,

```
sort(c(1,6,2,3))
```

would have an output of $1\ 2\ 3\ 6$.

Order, on the other hand, tells us the indices from highest to lowest.

```
order(c(1,6,2,3))
```

would have the output $1\ 3\ 4\ 2$ because the ordering is $x_1 < x_3 < x_4 < x_2$. If there are ties then the `order()` function will output then sort them according to the index, i.e. $\langle 3, 2, 3 \rangle$ would be sorted $2\ 1\ 3$ and not $2\ 3\ 1$. With both of these functions, we can change to descending order by adding the additional argument `decreasing = TRUE`, i.e.

```
order(c(1,6,2,3), decreasing = TRUE)
```

will produce the output $2\ 4\ 3\ 1$.
Rank tells us the relative ranking of the variables. So

```
rank(c(1,6,2,3))
```

will have an output of $1\ 4\ 2\ 3$ because the 6 is ranked fourth, the 2 is ranked second etc. The decreasing option is not available for `rank()`, however by using a negative in front of the input vector the opposite ranking will be obtained, i.e.

```
rank(-c(1,6,2,3))
```

would be $4\ 1\ 3\ 2$.
The most useful function for us in order to convert our *Sprint* times to rank-scores would hence be the `rank()` function. The following ranks the times and then scales them to the unit interval (we get our original times back first).

```
V[,1] <- original[,1]
V[,1] <- (length(V[,1]) - rank(V[,1]))/(length(V[,1])-1)
```

In this case we used `length(V[,1]) - rank(V[,1])` because the lowest value is given the rank 1 but we want it to have the highest score.

Using `if()` for cases

Using the `if()` function requires a careful consideration of the sequence and logic of our function. Let's first consider an example of a piecewise function for values between 0 and 1 that has its join at $(0.5, 0.7)$. So if the input is 0.5, then the output is 0.7. If the input is below 0.5, then it gets increased at the same ratio, and if it is above 0.5, then the ratio of increase drops off so that it still has the output of 1 if the input is 1.

We express the function as an equation in the following way

$$f(t) = \begin{cases} 0.7\frac{t}{0.5}, & 0 \leq t < 0.5 \\ 0.7 + 0.3\frac{t-0.5}{0.5}, & 0.5 \leq t \leq 1. \end{cases}$$

As a function in `R`, we need to create a clause that transforms it using the first equation if the value is less then 0.5, and using the second equation if it is above 0.5. There are a few different ways to do this. The easiest is to use `if(...) {...} else {...}`. Inside the `if()` brackets, we have something

like `t < 0.5` or `t >= 0.5` (the latter means greater than or equal to 0.5). Then in the first case brackets $\{...\}$, we tell the function what to do if the `if()` statement is true, and the second case brackets tells the function what to do otherwise. The following programs our piecewise function above.

```
pw.function <- function(t) {
if(t < 0.5) {0.7*t/0.5} else {0.7+0.3*(t-0.5)/0.5}
}
```

R Exercise 11 *Enter in the function and try out a few entries to see that it makes sense and is working correctly.*

We can repeat this process in nested form to define more cases. In the following we interpolate the points $(0.5, 0.7)$ and $(0.8, 0.9)$. In this case our intervals in terms of the input cases are $[0, 0.5]$, $[0.5, 0.8]$ and $[0.8, 1]$ respectively.

```
pw.function.2 <- function(t) {
if(t < 0.5) {0.7*t/0.5}
else {if(t <0.8) {0.7+0.2*(t-0.5)/0.3}
else {0.9+0.1*(t-0.8)/0.2} }
}
```

Another way to have a look at our variables and make sure our functions are working correctly is to plot them.

Plotting in two variables

If we just want to view a variable to get an idea of the distribution, we can input the vector containing that variable's data. So to plot our original Sprint data.

```
plot(original[,1])
```

This just plots the value in sequence, so along the x-axis is the index of the datum and the y-axis contains its value. It can be easier to see the distribution by first sorting the data. So we can enter

```
plot(sort(original[,1]))
```

Of course, usually histograms are used to plot distributions. So we can also use the following to get a snapshot of our data.

```
hist(original[,1])
```

If we want to plot a single variate function, plotting is reasonably straightforward, however we usually need to create our 'x' values and 'y' values beforehand. To create equispaced x values, the easiest way is to create a vector with our colon option and then scale it to our interval.

So if we want to create a vector of 100 points over the unit interval, we can write `(1:100)/100`. If we want to start at 0 (which would give us 101 points), then we can write `(0:100)/100`.
To plot the function $f(t) = t^2$ for these values, we can either use

```
x <- (1:100)/100
plot(x^2)
```

or if we want the x labels to correspond with the data in that variable, we would include these as the first argument and the transformed values as the second argument.

```
x <- (1:100)/100
plot(x,x^2)
```

However, if we want to plot a function like our piecewise function, we need to create the y values separately first. This is because in our cases we used `if(t < 0.5)`, so when we try to input `pw.function(x)` it will ask whether the vector is less than 0.5, which is not a valid operation. To create the vector of y values, we will also now need to use a repeating operation `for()`.

The `for()` function can perform an operation for every entry of a set. This is very useful when we can index our numbers.

The following sequence of operations first creates a vector of y-values (pre-filled with zeros). It uses `length(x)` so that it will be the same length as our `x` vector, but we could also just write 100. It then says for all the numbers in 1 to 100 (using the `1:100` vector), each entry in the `y` vector will be replaced by the piecewise function of the *corresponding* entry in the `x` vector. The # can be used in R for notes. Anything on the line that comes after the # is "commented out".

```
y <- array(0,length(x))       # 1. create a vector of zeros
for(i in 1:length(x)) {       # 2. perform this operation
y[i] <- pw.function(xs[i])    #    changing 'i' for the numbers
}                             #    between 1 to 100.
```

We now should be able to plot this piecewise function.

```
plot(x,y)
```

Defining power means

Let us see how we can implement the following power mean function.

Definition 1.1 - The power mean
For an input vector $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$*, the power mean is*

$$PM_p(\mathbf{x}) = \left(\frac{1}{n} \sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}} = \left(\frac{x_1^p + x_2^p + \cdots + x_n^p}{n} \right)^{\frac{1}{p}}.$$

This time we will have two inputs to the function, 'x', which will be a vector of inputs and 'p', which will be the power used.

```
PM <- function(x,p) {      # 1. pre-defining the function inputs
(mean(x^p))^(1/p)          # 2. our calculation which will also
}                          #    be the output
```

However this function will not work if $p = 0$. So we need to create a special case. For this, we will use the `if()` function.

```
PM <- function(x,p) {      # 1. pre-defining the function inputs
if(p == 0) {               # 2. condition for 'if' statement
prod(x)^(1/length(x))      # 3. what to do when (p==0) is TRUE
}
else {
(mean(x^p))^(1/p)          # 4. what to do when (p==0) is FALSE
}
}
```

Note here that for '=' conditions, we use a double equals sign '=='. So the possible conditions we can use inside the `if()` brackets are '==', '<', '>', '<=' and '>='.

R Exercise 12 *Define the power mean as a function in* R *and check the following.*

Input	*Expected Output*
`PM(c(3,2,7),2)`	*4.546061*
`PM(c(1,0,7),0)`	*0*
`PM(c(0.28,0.4,0.47),-557)`	*0.2805528*
`PM(c(0.28,0.4,0.47),-558)`	*Inf*

Note that in the last case, 'Inf' should not actually be the result. It's just that at this point, the system cannot tell the difference between one of the transformed inputs and infinity, i.e. if you compare `0.28^(-557)` and `0.28^(-558)`, the latter will be 'Inf' and then all of the operations become absorbed by this value.