# "Putting the R in ScHARR"

*Robert Smith*

*08. September 2019*

# Contents


Introduction to R . . . 3
Who we are: . . . 3

**Series of Short Courses in R. 4**

Course 1 Aims: . . . 4
Course 2 Aims: . . . 4
Course 3 Aims . . . 4
Course 4 Aims . . . 4
Course 5 Aims . . . 5

**Course 1 - Introduction to R. 6**

Install and navigate R Studio. . . . 7
Basics LED BY ROB. . . . 7
Basic operations . . . 7
Objects Basics . . . 8
Manipulations . . . 8
More complex objects . . . 8
Evaluations . . . 9
Subsetting . . . 10
Exercises . . . 11
Understanding R. #SARAH . . . 13
Basic functions . . . 13
Attributes . . . 14
Classes . . . 15
Environment . . . 16
Working with R. #Rob . . . 18
If Statements . . . 18
Loops . . . 18
Custom Function . . . 20
Packages . . . 21
Doing Research in R # Paul . . . 22
Importing Data . . . 22
Summarising Data . . . 23
Plotting Data . . . 23
Basic Regression . . . 23
Basic Simulation . . . 23

## Introduction to R

This series of short courses are designed to equip the participant with a basic set of tools to undertake research using R. The aim is to create a strong foundation on which participants can build skills and knowledge specific to their research and consultancy objectives. The course makes use of the authors' experiences (many of which were frustrating) of working with R for data-science and statistical analysis. However there are many other resources available, and we would particularly recommend the freely available content at *R for Data Science* as a good place to recap the materials taught in this course. The hard copy of Hadley Wickham and Garrett Grolemund's book of the same name (and content) is available at *Amazon.com*. Alternatively, a user guide is available on the CRAN R-Project website here, although the author finds this less easy to follow than Hadley Wickham's book described above. Further details of where to go to answer more specific questions are provided throughout the course.

Requirements: It is assumed that all participants on the course have their own laptop, and have previously used software such as Excel or SPSS. Some basic understanding of statistics and mathematics is required (e.g. mean, median, minimum, maximum).

## Who we are:

*Robert Smith* is a PhD Candidate at ScHARR, funded by the Wellcome Trust Doctoral Training Centre in Public Health Economics and Decision Science. His focus is on the methods used to estimate the costs and benefits of public health interventions, with a specific interest in microsimulation modelling (done in R). He has become increasingly intersted in the use of R-Markdown and R-Shiny to make research more transparent and to aid decision makers. While doing his PhD, Robert has been involved in projects with the WHO, Parkrun and Western Park Sports Medicine.

Paul Schneider ...

Sarah Bates ...

## Series of Short Courses in R.

We have put together content for a series of short courses in R. These courses are designed to provide the user with the necessary skills to utilise R to answer questions using data. By the end of the series a user should be able to manage data efficiently, analyse relationships between variables and display this in aesthetically pleasing graphs, run simulations, feed back on methods and create apps to facilitate the decision making process. The objective is to give the user an understanding of what is possible, and the foundation on which to achieve it.

**Course 1 Aims:**

By the end of the 1 day short course, the attendee should be able to:

- Install and navigate R Studio. Set working directory.
- Understand the types of objects and basic operations in R.
- Read in data from csv and excel files.
- Summarise data.
- Perform basic regression analysis on data.
- Create basic plots and graphs.
- Use an 'if' function
- Create a loop
- Know where to find further information (StatsExchange/Google).

**Course 2 Aims:**

By the end of the 1 day short course, the attendee should be able to:

- Create beautiful graphs using ggplot.
- Use the 'apply()' function to improve run-time.
- Create a custom function.
- Know where to find further information (StatsExchange/Google).

**Course 3 Aims**

By the end of 1 day short course, the attendee should also be able to:

- Understand the strengths and limitations of R for HTA.
- Create a markov model from scratch given known parameters.
- Create a microsimulation model to incorporate hetrogeneity between groups.
- Understand the importance of tranparency of coding. In particular commenting.

**Course 4 Aims**

By the end of the 1 day short course, the attendee should be able to:

- Understand the benefits and limitations of R-Shiny.
- Have a basic understanding of the principles behind R-Shiny.
- Create an R-Shiny application from scratch.
- Integrate beautiful plots into R-Shiny.
- Develop a user interface for an existing complex model in R-Shiny.

**Course 5 Aims**

By the end of the 1 day short course, the attendee should be able to:

- Understand the strengths and limitations of R-Markdown.
- Create replicatable HTML, Word and PDF documents using R-Markdown.
- Include chunks of code, graphs, references and biblographies, links to websites and pictures.
- Change replicate analysis for new or updated datasets, or create templates for routine data.
- Create markdown presentations.

# Course 1 - Introduction to R.

## Install and navigate R Studio.

R is a free software environment for statistical analysis and data science. It is a collaborative effort started by Robert Gentleman and Ross Ihaka in New Zealand, but now made up of hundreds of people. R is made freely available by the Comprehensive R archive Network (CRAN), in the UK it can be downloaded through the University of Bristol here.

Downloading R gives you the basic software environment, but an incredibly popular add-on called 'RStudio' is required to follow this course. You should download the free 'RStudio Desktop Open Source Licence' version for the laptop you will be attending the course with from RStudio.com.

If you have time before the course, it would be hugely beneficial to get familiar with RStudio.

**Objectives:**

Download R from https://www.stats.bris.ac.uk/R/.
Download RStudio from https://www.rstudio.com/products/rstudio/#Desktop.

## Basics LED BY ROB.

Our course starts in the R console, which those of you who are familiar with R but not RStudio will recognise. We will enter commands as input into the console, and receive output from the console. We will start with some simple basic operations, for which R is clearly very excessive.

### Basic operations

Entering 1+1, we get the output [1] 2. The output is 2, but the [1] lets us know that the number 2 is the first number of output. If we had an output that was particularly large (e.g. 100 seperate numbers) then r may let us know that the first row displayed starts with the first value [1] and the second row starts with the [x]th value.

```
# add 1 to 1.
1 + 1
```

```
## [1] 2
```

```
# divide 12 by 4
12/4
```

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

```
# Showing that the numbers in brackets show the number in the vector for the first number of the row.
runif(n = 30,min = 0,max = 1)
```

```
##  [1] 0.298922024 0.729350199 0.179309649 0.015150229 0.666157874
##  [6] 0.467735816 0.025391224 0.707295227 0.942377994 0.028251878
## [11] 0.699103247 0.182631316 0.007215845 0.226107838 0.500328743
## [16] 0.426208870 0.729363145 0.322356349 0.824036027 0.222459023
## [21] 0.122596158 0.505017173 0.398684589 0.124502667 0.862916592
## [26] 0.020472930 0.854383973 0.796377585 0.967495205 0.863049060
```

### Objects Basics

It is possible to create objects, and assign these objects some value (e.g. a number, a letter, a word, a time). We can then use these objects going forward rather than the values directly. Operations can be applied to these objects, and objects can be over-written.

```
# objects basics
x = 3
y = 5
x + y
```

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

```
x = 4
x + y
```

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

```
z = x + y
z
```

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

### Manipulations

```
x.2 = x
x.2
```

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

```
x.2 = x.2 + 1
x.2
```

```
## [1] 5
```

```
x
```

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

Objects don't have to take a single value. For example a single object may be the heights of each child in a class of children (in the example below a small class of 4). We can store objects together in a data-frame. In our example data-frame each column is a variable (height, weight, first_name), and each row is an individual. We can create a new variable in the dataframe by assigning a vector of values to that 'column'.

### More complex objects

```r
height = c(1.72,1.78,1.65,1.90) # 1:4
height
```

```
## [1] 1.72 1.78 1.65 1.90
```

```r
weight = c(68,75,55,79)
weight
```

```
## [1] 68 75 55 79
```

```r
first_name = c("Alice","Bob","Harry","Jane")
first_name
```

```
## [1] "Alice" "Bob"   "Harry" "Jane"
```

```r
#first_name + 1 # error

df = data.frame(height,weight,first_name)
df
```

```
##   height weight first_name
## 1   1.72     68      Alice
## 2   1.78     75        Bob
## 3   1.65     55      Harry
## 4   1.90     79       Jane
```

```r
df$bmi = df$weight / df$height^2
df
```

```
##   height weight first_name      bmi
## 1   1.72     68      Alice 22.98540
## 2   1.78     75        Bob 23.67125
## 3   1.65     55      Harry 20.20202
## 4   1.90     79       Jane 21.88366
```

### Evaluations

We can perform evaluations, which provide a true or false answer. For example the input 4>2 returns "FALSE". These evaluations can be done on a single value, or multiple values in all kinds of structures (a scalar, a vector, a matrix, an array ...) It can be very useful in cases where an outcome is binary (e.g. an individual dies or remains alive). Or where we want to change a continous variable to a binary.

```r
# simple evaluations
# 4 is greater than 2
4 > 2
```

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

```r
# 4 is greater than 5
4 > 5
```

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

```r
# 4 is equal to 3, note double == for an evaluation
4 == 3
```

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

```r
# 4 is not equal to 3, note != is not equal to.
4 != 3
```

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

```r
# the character x is equal to the character x.
"x" == "x"
```

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

```r
# show that evaluations can be undertaken on multple values, this returns the evaluation for each.
x <- c(1,2,3)
x == 1
```

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

```r
# the output from an evaluation can be stored as an object, x. This object can be subject to operations
x <- 4<2
# for example multiplication
x*2
```

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

### Subsetting

We can subset our data, to reduce it to those we are interested in. This is useful when cleaning our data, and when changing a continuous variable to a categorical.

```r
# Our data-frame contains the height, weight, first name and bmi of 4 individuals.
df
```

```
##   height weight first_name      bmi
## 1   1.72     68      Alice 22.98540
## 2   1.78     75        Bob 23.67125
## 3   1.65     55      Harry 20.20202
## 4   1.90     79       Jane 21.88366
```

```r
# create a variable called min_height which contains T/F for each individual being over 175cm.
min_height <- df$height >= 1.75
min_height
```

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

```r
# Subset the data to include only those observations (rows) for which height > 175cm (using min_height)
df.at_least_175 <- df[min_height,]
df.at_least_175
```

```
##   height weight first_name      bmi
## 2   1.78     75        Bob 23.67125
## 4   1.90     79       Jane 21.88366
```

```r
# Subset the data to include only those who are not above min-height of 175cm.
smaller = df$height < 1.75
df[smaller,]
```

```
##   height weight first_name      bmi
## 1   1.72     68      Alice 22.98540
## 3   1.65     55      Harry 20.20202
```

```r
df[!min_height,]
```

```
##   height weight first_name      bmi
## 1   1.72     68      Alice 22.98540
## 3   1.65     55      Harry 20.20202
```

```r
# we can also subset using %in% which limits the dataset to those meeting criteria, e.g. those whose fi
show_men = df$first_name %in% c("Harry","Bob")
df[show_men,]
```

```
##   height weight first_name      bmi
## 2   1.78     75        Bob 23.67125
## 3   1.65     55      Harry 20.20202
```

Not that there are other more advanced methods, which uses pipes and require less code (these are covered in more advanced courses).

### Exercises

#### Exercise 1

a) Calculate the following:

5*10 20/3

b) Calculate x where a = 20 b = 9, c = 5, d = 1.2 *you are only allowed to type each number once*

$x = 4b + 7c + 3d$

$x = \frac{8b+4c-12d}{a}$

$\frac{12\times(a+b)}{x}$

**Exercise 2**

a) Create a dataframe for five individuals (Andrew, Betty, Carl, Diane and Elisa) who are aged (62,80,24,40,56) and have gender (male, female, male,female, female).

b) Use evaluations and subsetting to find the characteristics of the individual who can claim their free bus pass (age 65+).

c) Create a variable in the dataframe called life expectancy, set this to 83 for females and 80 for males.

d) Create another variable called lyr (life years remaining) which is the number of years to life expectancy for each individual

## Understanding R. #SARAH

### Basic functions

Functions can be applied to make data manipulation easier. Many functions are included in baseR. For example mean. Mean returns the arithmetic mean of a vector of numbers. We can find out more about mean by entering ?mean.

```
# find out more about mean
?mean
```

```
## starting httpd help server ... done
```

```
# create a vector of numbers
some_numbers = c(10,12,13,13,14,18,20)

# find the arithmetic mean of the numbers
average_value = mean(some_numbers)

#return the numbers
average_value
```

```
## [1] 14.28571
```

Other functions are more complex, requiring multiple inputs. For example **round** requires two inputs: x) a value or multiple values digits) the number of digits to round to.

```
# find more information about round
?round

# Rounding 5.7 to 0 digits gives 6.
round(x = 5.7,digits = 0)
```

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

```
# Rounding our average value to 2 digits gives 14.29.
round(average_value,2)
```

```
## [1] 14.29
```

```
# We can do the same for multiple values
round(c(12.25,15.63,14.42),1)
```

```
## [1] 12.2 15.6 14.4
```

Another useful function is sequence:

This requires the number to start from (*from*), the number to finish at (*to*), and the interval between numbers (*by*) in the sequence. Alternatively, instead of *by* the length of the sequence can be set using *length.out*.

```r
# find out more about sequence
?seq

# create a sequence of odds from 1 to 21.
seq(from = 1, to = 21, by = 2)
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21
```

```r
# create a sequence of length 10 from 1 to 21
seq(from = 1, to = 21, length.out = 10)
```

```
##  [1]  1.000000  3.222222  5.444444  7.666667  9.888889 12.111111 14.333333
##  [8] 16.555556 18.777778 21.000000
```

### Attributes

```r
# attributes
df
```

```
##   height weight first_name      bmi
## 1   1.72     68      Alice 22.98540
## 2   1.78     75        Bob 23.67125
## 3   1.65     55      Harry 20.20202
## 4   1.90     79       Jane 21.88366
```

```r
names(df)
```

```
## [1] "height"     "weight"     "first_name" "bmi"
```

```r
#rownames(df) = c("row 1","row 2","row3") # error
rownames(df) = c("row 1","row 2","row3","row 4")
df
```

```
##       height weight first_name      bmi
## row 1   1.72     68      Alice 22.98540
## row 2   1.78     75        Bob 23.67125
## row3    1.65     55      Harry 20.20202
## row 4   1.90     79       Jane 21.88366
```

```r
x
```

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

```r
length(x)
```

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

```r
str(df)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ height    : num  1.72 1.78 1.65 1.9
##  $ weight    : num  68 75 55 79
##  $ first_name: Factor w/ 4 levels "Alice","Bob",..: 1 2 3 4
##  $ bmi       : num  23 23.7 20.2 21.9
```

```r
dim(df)
```

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

### Classes

```r
# classes
class(x)
```

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

```r
class("this is a sentence")
```

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

```r
class(df)
```

```
## [1] "data.frame"
```

```r
# matrix(1:9,ncol=3,nrow=3)

# the list
My_list = list(my_df = df,
               a_vector = x,
               another_one = y,
               some_text = c("foo","bar"))
My_list
```

```
## $my_df
##       height weight first_name      bmi
## row 1   1.72     68      Alice 22.98540
## row 2   1.78     75        Bob 23.67125
## row3    1.65     55      Harry 20.20202
## row 4   1.90     79       Jane 21.88366
##
## $a_vector
## [1] FALSE
##
## $another_one
## [1] 5
##
## $some_text
## [1] "foo" "bar"
```

```r
My_list$a_vector
```

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

```r
My_list$my_df
```

```
##       height weight first_name      bmi
## row 1   1.72     68      Alice 22.98540
## row 2   1.78     75        Bob 23.67125
## row3    1.65     55      Harry 20.20202
## row 4   1.90     79       Jane 21.88366
```

```r
My_list[[1]]
```

```
##       height weight first_name      bmi
## row 1   1.72     68      Alice 22.98540
## row 2   1.78     75        Bob 23.67125
## row3    1.65     55      Harry 20.20202
## row 4   1.90     79       Jane 21.88366
```

```r
My_list[[1]]$height
```

```
## [1] 1.72 1.78 1.65 1.90
```

```r
str(My_list)
```

```
## List of 4
##  $ my_df      :'data.frame': 4 obs. of  4 variables:
##   ..$ height    : num [1:4] 1.72 1.78 1.65 1.9
##   ..$ weight    : num [1:4] 68 75 55 79
##   ..$ first_name: Factor w/ 4 levels "Alice","Bob",..: 1 2 3 4
##   ..$ bmi       : num [1:4] 23 23.7 20.2 21.9
##  $ a_vector   : logi FALSE
##  $ another_one: num 5
##  $ some_text  : chr [1:2] "foo" "bar"
```

```r
# NA, Nan, Inf
```

### Environment

```r
# enviroment & wd
ls()
```

```
##  [1] "average_value"   "df"              "df.at_least_175"
##  [4] "first_name"      "height"          "min_height"
##  [7] "My_list"         "show_men"        "smaller"
## [10] "some_numbers"    "weight"          "x"
## [13] "x.2"             "y"               "z"
```

```
rm("y")
ls()
```

```
##  [1] "average_value"   "df"              "df.at_least_175"
##  [4] "first_name"      "height"          "min_height"
##  [7] "My_list"         "show_men"        "smaller"
## [10] "some_numbers"    "weight"          "x"
## [13] "x.2"             "z"
```

```
# attach(...)
rm(list=ls())
ls()
```

```
## character(0)
```

## Working with R. #Rob

### If Statements

If statements allow us to perform different actions depending on a condition. The normal brackets contain the evaluation, the curly brackets the command if that evaluation is true. For example here if$(1 = 1)$then print the number 2.

```
if(1==2){
  print(2)
}
```

We can use this to assign risks to individuals based on characteristics. For example if an individual has a health condition (e.g. Type2Diabetes) they may have a higher risk of Heart Disease. Therefore if we know that individuals with diabetes have a 10% risk of Heart Disease and individuals without Type 2 Diabetes have a 3% risk of Heart Disease we could assign risk to an individual as follows:“

```
t2d <- 0

if(t2d==1){
  risk <- 0.1
}else{
  risk <- 0.03
}
```

This method can be combined with the next tool, loops, to assign risks for lots of individuals.

### Loops

There are numerous different ways to create loops, we are going to look at the simplist, the **for** loop which executes the command for each number given.

```
 # Loop printing 1 to 10

for(i in 1:10) { # for each value of i from 1 to 10.
  print(i)       # print the value of i
}                # close the loop
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

We can also create an object, in this case a vector with the numbers we would like to loop through:

```r
# create an object of odds
odds <- c(1,3,5,7,9)

# Loop printing all the values in the odds vector.
for(i in odds) { # for each value of i in the odds vector.
  print(i)       # print the value of i
}                # close the loop
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
```

If we create a vector, *ourdata*, beforehand we can fill it with the output from each iteration of the loop. This is particularly useful if we want to record the output of a simulation.

```r
#create an vector of empty data
ourdata <- vector(mode = "numeric",length = 10)

# Fill our vector with values
for(i in 1:length(ourdata)) { # for each value of i in our data object.
  ourdata[i] <- i             # print the value of i
}                             # close the loop

print(ourdata)                # print the vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Having multiple loops can can sometimes make code very messy, an alternative is called apply. It *applies* (hence the name) the same function to multiple rows/columns. This is much tidier, especially where the function is complicated.

As an example, I create a dataset, a matrix of random numbers with 100 columns and 100 rows. I want to know the average of each row.

```
## Time difference of 0.03091884 secs
```

In this case using a function is much slower, but it does make the code tidier, which can be a bigger probem than model run-time. It is up for the programmer to decide which is more important.

```r
squarelength <- 1e4

randnums <- matrix(data = runif(squarelength*squarelength,min = 0,max = 1),
                   nrow = squarelength,
                   ncol = squarelength)

temp <- vector(mode = "numeric",
               length = squarelength)

# measure time to run
t <- Sys.time()
```

```r
temp <- apply(X = randnums,
              FUN = mean,
              MARGIN = 1)

print(Sys.time()-t)
```

```
## Time difference of 2.766603 secs
```

### Custom Function

Creating custom functions is a huge strength of R. Calling scripts in STATA is possible, but not as tidy as in R.

A function is a piece of script that is stored as an object. A function generally takes a user input, performs some actions based on the statements in the function, and returns some output. It is possible to write a script that uses no functions, but using them can reduce unncessary replication of long sections of code, thereby making work easier to follow.

There are many functions predefined in base R. *Mean* is an obvious example. Custom functions can also be defined within a script, or called from another R-Script using *source*.

An example function is shown below. The function simply takes a temperature, and a conversion (Celsius to Fahrenheit = True, Fahrenheit to Celsius = False), and returns the converted temperature. The function has the name *temperature_converter*, it uses two inputs: x : a scalar or vector of numbers c_to_f : if True then converts celsius to fahrenheight, if FALSE then converts fahrenheight to celsius.

```r
# my function
temperature_converter = function(x,c_to_f = T){
  if(c_to_f){
    temp = x * 9/5 + 32
  } else {
    temp = (x - 32) * 5/9
  }
  return(temp)
}

temperature_converter(c(40,30,20,10,0),c_to_f = T)
```

```
## [1] 104  86  68  50  32
```

When creating custom functions it is important to ensure that you don't overwrite an existing function. For example, we can create the function *mean* which does not return the arithmetic mean of a vector of numbers, but instead prints an unkind message.

```r
mean = function(x){
  print("You are a buffoon")
}
mean(c(15,10))
```

```
## [1] "You are a buffoon"
```

**Exercise 1**

Create a function which multiplies three input numbers together

**Exercise 2**

Create a function which divides a number by 37, then rounds the number to 2 decimal places. I find it easier to do things one step at at time. In this case I would create a temporary object which was the value of the number divided by 37, then round the temporary number to two digits, overwriting the original temporary object. I would then return that temporary number.

**Exercise 3**

Calculate, in real terms, the present value of stream of annual monetary benefits received at the start of each yearfor 10 years, given a user specified discount rate

### Packages

Creating all your functions from scratch is a lot of work, where others have previously created the same functions, why reinvent the wheel? Instead we can stand on the shoulders of others by downloading 'packages'. Packages usually contain functions, and sometimes example datasets or datasets which enable a user to do certain things (e.g. plot a world map). Importantly though, we must understand what the functions within these packages are doing if we are going to use them.

A package can be installed from CRAN using the function *install.packages()* with the name of the package in "". Packages can also be installed from Github, but this is covered later.

```
install.packages("stats")
devtools::install_github("hadley/babynames")
```

To find more information about the babynames package you can type

```
help(package = "babynames")
```

To load the downloaded package from the file it was downloaded to you can type:

```
library(babynames)
ls("package:babynames")
View(babynames)
```

In this package there is just a dataset of baby names. However other packages allow you to do certain things. In many cases there are lots of different options. For example the rworldmap package contains data on country boundaries and enables the user to create custom maps. For a really simple map see the example below:

```
#install.packages("rworldxtra")
#install.packages("mapdata")
library(rworldxtra)

newmap <- getMap(resolution = "high")

plot(newmap,
     xlim = c(-10, -2),
     ylim = c(50, 62),
     asp = 1)
```

```
points(x = -1.480840,
       y= 53.380720,
       col = "red",
       pch =  16,
       cex = 1)

text(x = -1.480840,
     y = 53.380720,
     labels = "Sheffield",
     pos = 1,
     col = "red")
```

## Doing Research in R # Paul

### Importing Data

In almost every case, you will want to import some data to analyse. This data will come in a variety of formats. R studio has a nice feature in Environment>Import_Dataset which enables you to select the file you wish to download (similar to STATA) from your computer. The data is then imported into R and the code available in the console.

It is possible to import files in the following formats:

| Type | Suffix |
|---|---|
| R | .R |
| CSV | .csv |
| Excel | .xls/.xlsx |
| SPSS | .spv |
| Stata | .dta |

Alternatively packages can be installed to import data in almost any format. For example the readr package can read in spreadsheets from text files or tab delimited files.

### CSV (Comma-seperated values)

We can import the file using the full path with the file name and suffix included such as below:

```
read.csv("C:/Users/Robert/Google Drive/MyProject/Data/rawdata.csv", header=FALSE)
```

### Setting Working Directory

If you know that you will be reading and writing multiple files from the same folder, you can set a working directory. The working directory is the defined folder in which R will then import and export files from and to. This allows users to send whole files to others who can replicate the work by simply changing the working directory to the new file location.

The current working directory can be found by typing:

```
getwd()
```

```
## [1] "C:/Users/Robert/Google Drive/Teaching/R Course/Intro_to_R"
```

A new working directory can be set by clicking on the tab (Session) then (Set_Working Directory), or by the command setwd:

```
# filename = "C:/Users/Robert/Documents"
setwd(filename)
```

### Downloading files from the internet

Sometimes it is more practical to download files directly from the internet. There are lots of different packages out there to do this. The one I use was developed by Hadley Wickham, called readr. Here we are going to download some data from the github page for the course.

```
# load the readr package, if this is not installed then install it.
library(readr)

#use the function read_csv
data <- read_csv("https://raw.githubusercontent.com/RobertASmith/Intro_to_R/master/Data/who_complete.cs
```

Downloading files directly to R within the same script as the analysis can be useful since it reduces the risk of you accidently changing the file. Just be careful that the data will always be available.

### Summarising Data

### Plotting Data

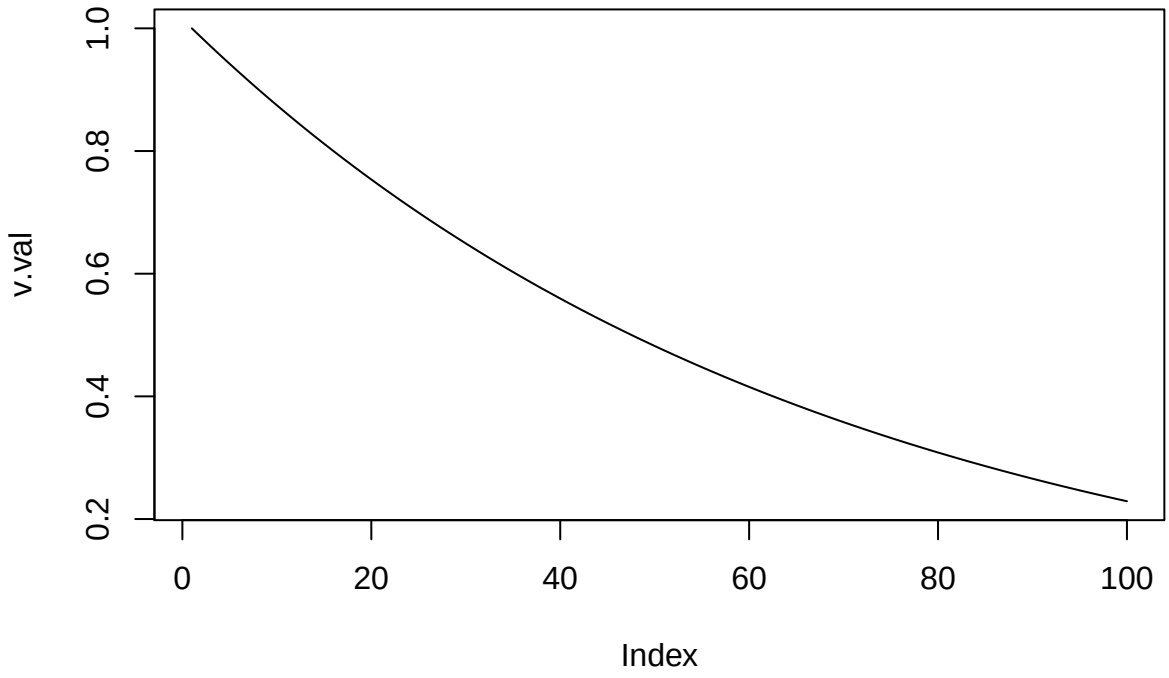
### Basic Regression

### Basic Simulation

We can do much more complex calculations within loops, the sky is the limit, but here are two examples.

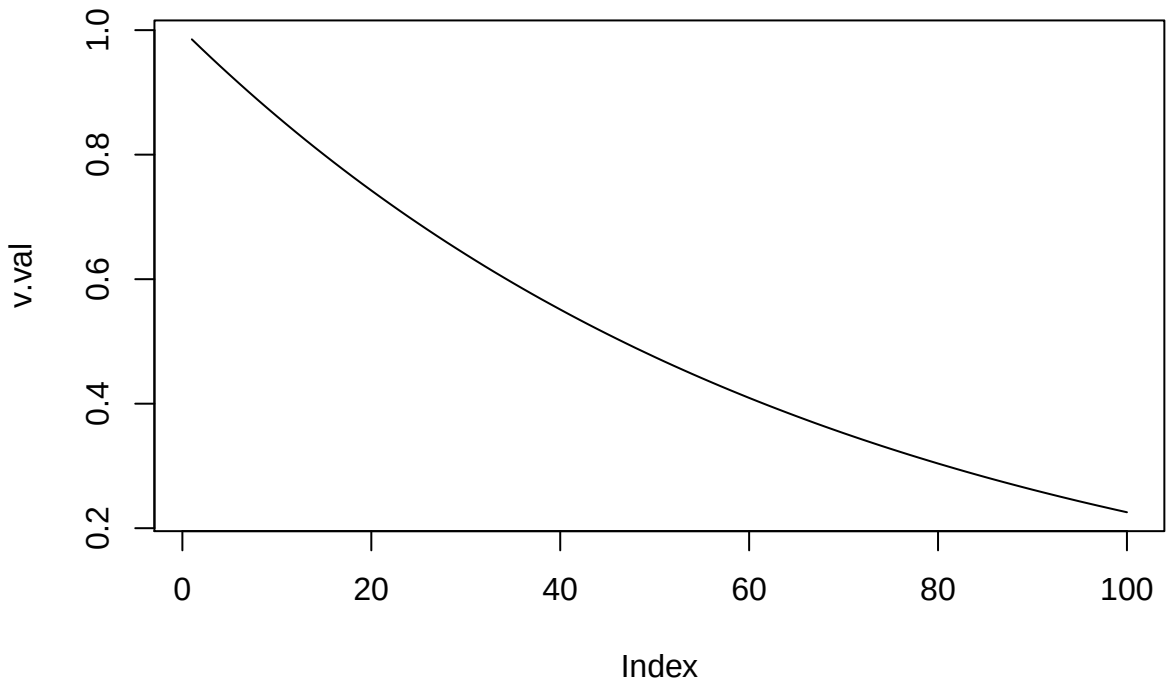
**Example 1** Example 1 is an example of discounting each year simply by multiplying the previous year value by (1/(1+d.r)).

```
#create an vector of empty data
v.val <- vector(mode = "numeric",length = 100)

# start with 1
v.val[1] <- 1

# Discount our value at 1.5% each year
for(i in 2:length(v.val)) {            # for each value of i in the odds object.

  v.val[i] <- v.val[i-1]* (1/1.015)    # it takes the value of the previous year value 1/(1.015)

  }                                      # close the loop

plot(v.val,type="l")                           # print the vector
```



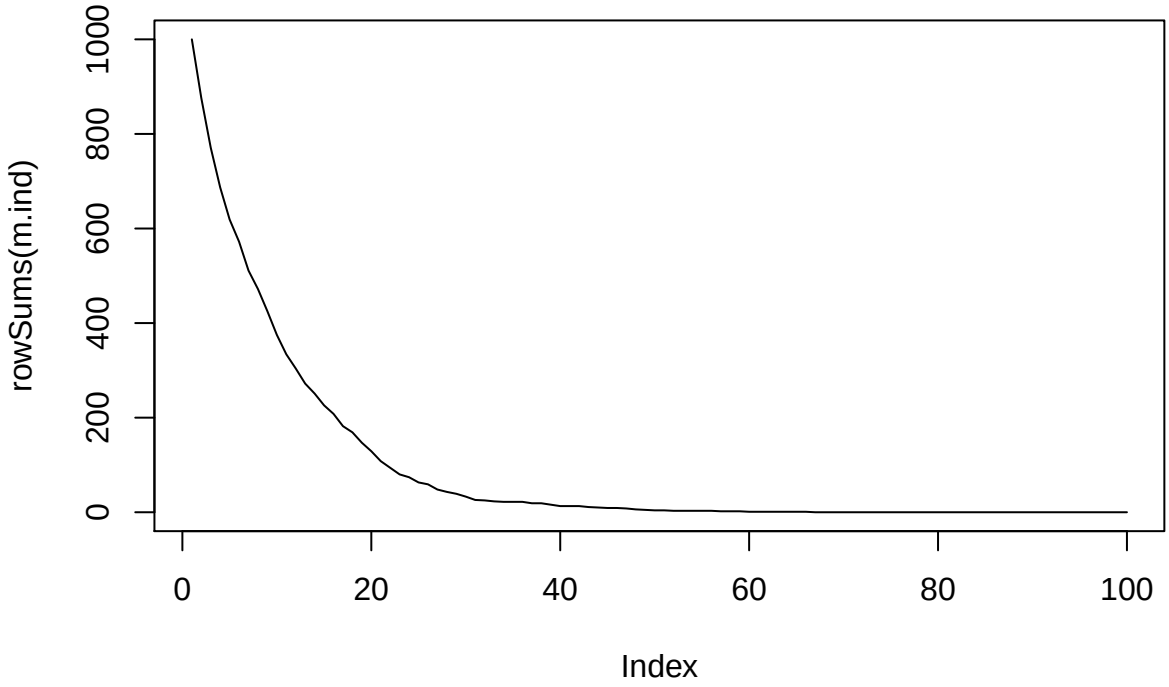

An important lesson though, it is often possible to achieve a goal without loops, which is much faster in R when running big simulations. In this case we could have achieved the same result with the following, much simpler code:

```
v.val <- (1/1.015)^(1:100)

plot(v.val,type = "l")
```




**Example 2** Example 2 records the survival of 1000 individuals who die with probability 0.1 in any given year.

```
#create an vector of empty data
m.ind <- matrix(data = NA,
                nrow = 100,
                ncol = 1000)

# everyone starts alive (1)
m.ind[1,] <- 1

# Each person has a probability of death of 0.1
for(i in 2:nrow(m.ind)) {

  # value is 1 if alive in previous period and random number > 0.1
  m.ind[i,] <- m.ind[i-1,] * runif( n = 1000, min = 0, max=1 ) > 0.1
}                                       # close the loop

plot(rowSums(m.ind),type="l")   # print the sums of the rows of the matrix (% alive)
```


rowSums(m.ind)
0
200
400
600
800
1000
0
20
40
60
80
100
Index