"Putting the R in ScHARR"

Robert Smith

05. October 2019

Contents

Background

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 are we:

All of the tutors on the course are PhD candidates in the Wellcome Trust Doctoral Training Centre for Public Health Economics and Decision Science at the School of Health and Related Research at the University of Sheffield.

Robert Smith joined ScHARR in 2016. His research focuses 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 and Parkrun.

Paul Schneider joined ScHARR in 2018. He is working on conceptual and methodological problems in valuing health outcomes in economic evaluations. A medical doctor and epidemiologist by training, he has used R in various research projects, ranging from the modeling costs of breast cancer, and value of information analyses, to the monitoring of influenza in real-time using online data. He is a keen advocate of open science practices.

Sarah Bates joined ScHARR in 2016. Sarah is examining the role of psychological factors in weight trajectories during and after a weight management intervention, and how these factors can be used to inform weight trajectories in a health economic model of obesity. Sarah is using microsimulation modelling in R throughout her PhD.

Thomas Bayley joined ScHARR in 2016. His research focuses on modelling the complex relationships that exist between Obesity, Depression and Socioeconomic status. He is interested in how data analysis and simulation modelling can be used to understand causal mechanisms in complex systems.

Naomi Gibbs joined ScHARR in 2017. Naomi is working on a health economic model to inform alcohol pricing policy options in South Africa. She is working closely with local stakeholders to conceptualise and validate the model. Naomi will be using R to create and communicate her modelling work and hopes to enable greater engagement from policy makers through interactive dashboards.

Our 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 - Intro to R

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.
- Know where to find further information (StatsExchange/Google).

Course 2 - Intermediate R

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

- Create a custom function.
- Use if functions
- Use loops.
- Install and load packages.
- Use the apply, mapply and lapply functions.
- Use Order, aggregate and other functions to more effectively manipulate datasets.
- Know where to find further information (StatsExchange/Google).

Course 3 - Beautiful Visualisations

By the end of the half 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 4 - R for Health Economics

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 5 - R Shiny

By the end of the half 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 6 - Collaboration in R

By the end of the half 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 2 - Intermediate R

The aim of this course is to develop the skills taught in the previous course: Intro to R. The course may be useful for those who have previously used R but would like to improve the efficiency of their code or speed up their data cleaning and analysis. This course focuses on custom functions, if functions, loops and downloading packages. These skills form the foundations of modelling in R and so we take time to ensure a strong knowledge base.

Custom functions

Functions are the building blocks of R. They can be called by the script to allow users to repeat processes. This is similar to calling a do file in STATA, but far more powerful (as you will see).

R has some base functions (e.g. mean(), sum(), round(), etc) and there are numerous packages that provide many more functions.

However, we often want to do things for which no function exists. We therefore need to create our own custom functions.

Custom functions are objects which appear in the global environment. They can be created in console, but are usually created and stored in scripts.

There are four main steps to creating a useful function: 1. decide what needs to go into the function. 2. decide what will happen to the input. 3. decide what will be output from the function. 4. decide the name of the function.

The function below satisfies these criteria: 1. the object x, is input into the function. 2. an object *temp* is created **inside the function, not in the global environment**. 3. temp is output (returned) from the function. 4 the function is called add_five - it is in the global environment. If we click on it we can see it.

```
add_five <-  function(x){
  temp <- x + 5
  return(temp)
}
```

If we create add_five, and run it within the console with x = 3 we get ... 8. If there is only one input we don't need to specify x. If we have no input we will get an error (in this case) because the function requires an *x*.

```
add_five <-  function(x){
  temp <- x + 5
  return(temp)
}

# lets try...
add_five(x = 3)

# and in short form...
add_five(20)

# NOTE: temp_res only existed temporarily within the function, only the bits within return() are given.
#temp

# Also note: if we do not supply an argument, we get an error
add_five()
```

To avoid the problem of getting an error when there is no input, we can specify a default input. For example here we specify a default x = 0. This will over-write our current function in the global environment.

Note: we dont have to call it x, but can use any name!

```
add_five = function(x = 0){
  temp_res <- x + 5
  return(temp_res)
}

# when no argument is provided, the function uses the default
add_five()
```

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

```
# when an argument is provided, it overwrites the default
add_five(x=50)
```

```
## [1] 55
```

A function can have multiple inputs and/or outputs. The first example has two inputs, x and y, which it adds together, gets a total and returns the total. The second example takes the two numbers, calculates the total and the difference between the two, then returns (in a list), *x*, *y*, the *difference*, and the *total*.

```
#two inputs one output.
add_numbers <- function(x,y){
  total <- x+y
  return(total)
}

add_numbers(3.2,5.1)
```

```
## [1] 8.3
```

```
# two inputs three outputs.
three_out <- function(x,y){
  total <- x + y
  dif <- y - x
  return(list(x,y,dif,total))
}

three_out(1,5)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 4
##
## [[4]]
## [1] 6
```

We can use functions within functions. For example the function below takes y, uses the *add_five* function to add five to y, then squares the number, returning the result.

```
add_five_squared <- function(y){
  temp_1 <- add_five(x=y)
  temp_2 <- temp_1^2
  return(temp_2)
}

add_five_squared(5)
```

```
## [1] 100
```

Exercise

Questions

1. Create a function 'miles_to_km' that translates miles in km (1km = 0.621 miles).
2. How many kms are 5, 13, and 60 miles?
3. Why would you ever want to write a custom function? Give 3 examples

Solutions

1&2

```
miles_to_km <- function(km){
  miles <- km * 0.621
  return(miles)
}

miles_to_km(5)
```

```
## [1] 3.105
```

```
miles_to_km(13)
```

```
## [1] 8.073
```

```
miles_to_km(60)
```

```
## [1] 37.26
```

3.

Coding principle: never write code more than once! This is the main principle of building packages in R. People use existing functions, combine and adjust them, and thereby create new functions. There are functions to do all kinds of things including calculating mortality risk from a vector of characteristics, plotting data in a template plot to ensure consistency, running simulations for a baseline, then treatment group.

If statements —-

If statements check whether a condition is met, and if true executes some command, if not true then the command is not executed. For example, if the statement within the brackets (1+1 == 2) is TRUE, then the if-function will print "This statement is true". If the statement is not satisfied, then nothing happens, as shown in the second example below:

```
if(1+1 == 2){
  print("This statement is TRUE")
}
```

```
## [1] "This statement is TRUE"
```

```
if(100 < 3){
  print("This statement is TRUE")
}
```

NOTE: if() takes a logical as argument (TRUE or FALSE)

The if function is followed by two kinds of brackets. The code within the first set of brackets (the normal brackets) is evaluated evalauted to determine whether it is TRUE. If TRUE, all code within the curly brackets is run. If false then the code is not run.

The code can do multiple things... so below foo = foo so the code prints two lines of code. The second example bar does not equal foo so the two lines are not run.

```
if("foo" == "foo"){
  print("line 1")
  print("line 2")
}
```

```
## [1] "line 1"
## [1] "line 2"
```

```
# here, the opposite case
if("foo" == "bar"){
  print("line 1")
  print("line 2")
}
```

We can add an *else* statement, to specify what should happen if the condition is not satisfied. The *else* sits after the code to be run if the statement was true. This makes it simple to see what will be run in either case

```
if("foo" == "bar"){
  print("This statement is TRUE!")
} else {      # NOTE: else command must be in the same line as the } !!!
  print("This statement is FALSE!")
}
```

```
## [1] "This statement is FALSE!"
```

The use of if statements is common, is used regularly for all kinds of purposes. If statements can be used in functions as below:

```
abs_value = function(num){
  if(num<0){
    res = -1 * num
  } else {
    res = num
  }
  return(res)
}

abs_value(-2)
```

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

```
abs_value(5)
```

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

Exercise

Question

1. create a function 'careful_division' that takes two numbers num and den and divides them (num / den)
2. adapt the function to only divide the numbers if den is not equal to zero.
3. if den is zero, display an error message (use the print() function)

Solution

```
careful_division = function(nom,den){
  if(den != 0){
    res = nom/den
    return(res)
  } else {
    print("error")
  }
}

careful_division(1,2)
```

```
## [1] 0.5
```

```
careful_division(1,0)
```

```
## [1] "error"
```

Loops

There are many ways to program a loop in R. There is a 'family' of functions that can be used in different circumstances (apply, lapply, sapply, eapply, mapply). However the most intuitive way is the *for* loop. For loops perform commands multiple times.

The for statement consists of two brackets, the first normal brackets generally contain three terms:

1. A letter or name of the object passed into the function, in the example below we use *i*.
2. A term, in this case *in* which tells us we will loop through all values in the set provided.
3. A set of values, in the order which i will take in the loop.

So in the example below i takes the value 1, then 2 then 3. The code therefore runs three times. Each time the command *print(i)* is run. i <- 10

```
for(i in 1:3){ # 1:3 is short for c(1,2,3)
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

Note: it is not necessary to always use i. Any name/leter will work.

```
for(j in 5:10){  # 5:10 is short for c(5,6,7,8,9,10)
  print(j)
}
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```
for(name in c("yoda","rey","luke")){
  print(name)
}
```

```
## [1] "yoda"
## [1] "rey"
## [1] "luke"
```

Loops are useful because we often need to do the same or similar operations multiple times. For example if we want to add all numbers from 1 to 100:

```
sum_1_100 = 0 # initialise object

for(i in 1:100){
  sum_1_100 = sum_1_100 + i
}
sum_1_100
```

```
## [1] 5050
```

```
sum_1_100 <- 0 # initialise object
for(i in 1:100){
  # use print() to see what is going on within the loop
  print(paste(sum_1_100,"+",i))     # note: paste() is a function to combine strings
  sum_1_100 = sum_1_100 + i         # add whatever i is to sum_1_100
}
```

```
## [1] "0 + 1"
## [1] "1 + 2"
## [1] "3 + 3"
## [1] "6 + 4"
## [1] "10 + 5"
## [1] "15 + 6"
## [1] "21 + 7"
## [1] "28 + 8"
## [1] "36 + 9"
## [1] "45 + 10"
## [1] "55 + 11"
## [1] "66 + 12"
## [1] "78 + 13"
## [1] "91 + 14"
## [1] "105 + 15"
## [1] "120 + 16"
## [1] "136 + 17"
## [1] "153 + 18"
## [1] "171 + 19"
## [1] "190 + 20"
## [1] "210 + 21"
## [1] "231 + 22"
## [1] "253 + 23"
## [1] "276 + 24"
## [1] "300 + 25"
## [1] "325 + 26"
## [1] "351 + 27"
## [1] "378 + 28"
## [1] "406 + 29"
## [1] "435 + 30"
## [1] "465 + 31"
## [1] "496 + 32"
## [1] "528 + 33"
## [1] "561 + 34"
## [1] "595 + 35"
## [1] "630 + 36"
## [1] "666 + 37"
## [1] "703 + 38"
## [1] "741 + 39"
## [1] "780 + 40"
## [1] "820 + 41"
## [1] "861 + 42"
## [1] "903 + 43"
## [1] "946 + 44"
## [1] "990 + 45"
```

```
## [1] "1035 + 46"
## [1] "1081 + 47"
## [1] "1128 + 48"
## [1] "1176 + 49"
## [1] "1225 + 50"
## [1] "1275 + 51"
## [1] "1326 + 52"
## [1] "1378 + 53"
## [1] "1431 + 54"
## [1] "1485 + 55"
## [1] "1540 + 56"
## [1] "1596 + 57"
## [1] "1653 + 58"
## [1] "1711 + 59"
## [1] "1770 + 60"
## [1] "1830 + 61"
## [1] "1891 + 62"
## [1] "1953 + 63"
## [1] "2016 + 64"
## [1] "2080 + 65"
## [1] "2145 + 66"
## [1] "2211 + 67"
## [1] "2278 + 68"
## [1] "2346 + 69"
## [1] "2415 + 70"
## [1] "2485 + 71"
## [1] "2556 + 72"
## [1] "2628 + 73"
## [1] "2701 + 74"
## [1] "2775 + 75"
## [1] "2850 + 76"
## [1] "2926 + 77"
## [1] "3003 + 78"
## [1] "3081 + 79"
## [1] "3160 + 80"
## [1] "3240 + 81"
## [1] "3321 + 82"
## [1] "3403 + 83"
## [1] "3486 + 84"
## [1] "3570 + 85"
## [1] "3655 + 86"
## [1] "3741 + 87"
## [1] "3828 + 88"
## [1] "3916 + 89"
## [1] "4005 + 90"
## [1] "4095 + 91"
## [1] "4186 + 92"
## [1] "4278 + 93"
## [1] "4371 + 94"
## [1] "4465 + 95"
## [1] "4560 + 96"
## [1] "4656 + 97"
## [1] "4753 + 98"
## [1] "4851 + 99"
```

```
## [1] "4950 + 100"
```

```
sum_1_100
```

```
## [1] 5050
```

Note: of course, this code is still very inefficient, to add all numbers between a and 100 you can use:

```
    sum(1:100)
```

```
## [1] 5050
```

Exercise

Questions

1. create a vector of jedis, named Luke, rey, yoda.
2. use a loop to say 'hi' to each of them, use paste("hi",name) for that!

Solutions

```
jedis <- c("Luke","Rey","Yoda")

for(name in jedis){
  print(  paste("Hi",name)  )
}
```

```
## [1] "Hi Luke"
## [1] "Hi Rey"
## [1] "Hi Yoda"
```

We can also store the results within a for loop. For example say we want to create a multiplication table for 6:

```
count_numbers <- c() # We create an empty object
count_numbers       # it exists, but has no content
```

```
## NULL
```

```
for(i in 1:10){
  temp <- 6 * i
  count_numbers = c(count_numbers, temp)
}

count_numbers
```

```
##  [1]  6 12 18 24 30 36 42 48 54 60
```

For loops are certainly the most common type of loop. However *while* loops can also be very useful. They repetitively loop through the code within the curly brackets while their parameter is TRUE. Note: be careful that this isn't forever!!

In the example below we initialise start_value to 1. Then we have a while loop. The code in the curly bracket will loop until start_value is not lower than 8. (i.e. WHILE start_value <8).

Therfore we should get printed values from 1-7, and start_value will be 8 but this value was not printed.

```
start_value <- 1

while(start_value < 8){
  print(start_value) # using print within loops is useful to see what is going on inside
  start_value = start_value + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

```
start_value # when start_value became 8, the while loop was terminated
```

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

We can do the same with data frames

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
dim(mtcars) # dim useful command when looping over a dataframe
```

```
## [1] 32 11
```

```
dim(mtcars)[1]   # gives us the number of rows
```

```
## [1] 32
```

```
dim(mtcars)[2]   # gives us the number of columns
```

```
## [1] 11
```

```r
1:dim(mtcars)[2] # i.e. gives us an index for each column in the dataframe
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11
```

```r
# To loop over the columns and compute the mean for each we can use:
col_means = c()
for(col in 1:dim(mtcars)[2]){
  temp = mean(mtcars[,col])
  col_means = c(col_means,temp)
}

col_means
```

```
##  [1]  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##  [7]  17.848750   0.437500   0.406250   3.687500   2.812500
```

Nested loops ——

As with functions it is entirely possible to nest loops. in the example below we nest the loops 1-3 for i and j. Don't worry too much about *cat*, the function simply prints and obeys the operations in the brackets. We get 3*3 = 9 rows.

Example 1

```r
for(i in 1:3){
  for(j in 1:3){
    cat("\n",i,"*",j,"=",i*j)
  }
}
```

```
##
##  1 * 1 = 1
##  1 * 2 = 2
##  1 * 3 = 3
##  2 * 1 = 2
##  2 * 2 = 4
##  2 * 3 = 6
##  3 * 1 = 3
##  3 * 2 = 6
##  3 * 3 = 9
```

Example 2 Here we can do something similar for different combinations of dress wear. Since there are two dimensions of 4 we have 4^2 = 16 combinations.

```r
hats = c("green hat","red cap","white hat","blue hat")
shoes = c("red shoes","blue shoes","pink shoes","no shoes")
possible_combinations = c()

for(hat in hats){
  for(shoe in shoes){
    possible_combinations = c(possible_combinations,paste(hat,"+",shoe))
```

```
  }
}
possible_combinations
```

```
##  [1] "green hat + red shoes"  "green hat + blue shoes"
##  [3] "green hat + pink shoes" "green hat + no shoes"
##  [5] "red cap + red shoes"    "red cap + blue shoes"
##  [7] "red cap + pink shoes"   "red cap + no shoes"
##  [9] "white hat + red shoes"  "white hat + blue shoes"
## [11] "white hat + pink shoes" "white hat + no shoes"
## [13] "blue hat + red shoes"   "blue hat + blue shoes"
## [15] "blue hat + pink shoes"  "blue hat + no shoes"
```

EXERCISE: putting it all together

Questions

1. create a function roll_dice that gives you ONE random number between 1 and 6. Use the function sample() for that (set the parameter n = 1).
2. create a loop to roll the dice 100 times and store the outcomes in a vector: 'eyes'. How many sixes have you rolled?.
3. create another loop to roll the dice until you roll a six record how many times you have to roll the dice.
4. take the loop from 3, and only count the throws that show a 1 (ignore 2,3,4 and 5), and stop when 6.

Solutions

Soluation 1

```
roll_dice = function(){
  spots = c(1,2,3,4,5,6)
  outcome = sample(x = spots,size = 1)
  return(outcome)
}

roll_dice()
```

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

Solution 2

```
eyes = c()
for(i in 1:100){
  eyes = c(eyes,roll_dice())
}

eyes
```

```
##   [1] 5 1 6 6 3 1 3 3 6 4 4 1 4 2 6 3 2 1 6 3 1 4 4 5 5 3 4 6 4 5 1 5 5 3 1
##  [36] 6 3 5 4 2 5 4 6 1 1 2 2 4 6 3 6 5 4 1 2 3 4 1 2 3 5 2 3 4 4 2 1 4 3 4
##  [71] 5 1 1 6 6 6 1 1 2 5 2 3 1 2 2 5 3 2 1 6 4 5 6 2 3 2 4 2 2 6
```

```
sum(eyes==6)
```

```
## [1] 16
```

Solution 3

```
isSix = F
rollCounter = 0
while(!isSix){
  isSix = roll_dice() == 6
  rollCounter = rollCounter +1
}
rollCounter
```

```
## [1] 10
```

Solution 4

```
isSix = F
rollCounter = 0
while(!isSix){
  res = roll_dice()
  if(res == 1){
    rollCounter = rollCounter+1
  }
  isSix = roll_dice() == 6
}
  rollCounter
```

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