Introduction to R and RStudio

A Guide for Beginners

Statistical Computing and Empirical Methods
Unit EMATM0061, Data Science MSc

Rihuan Ke

rihuan.ke@bristol.ac.uk

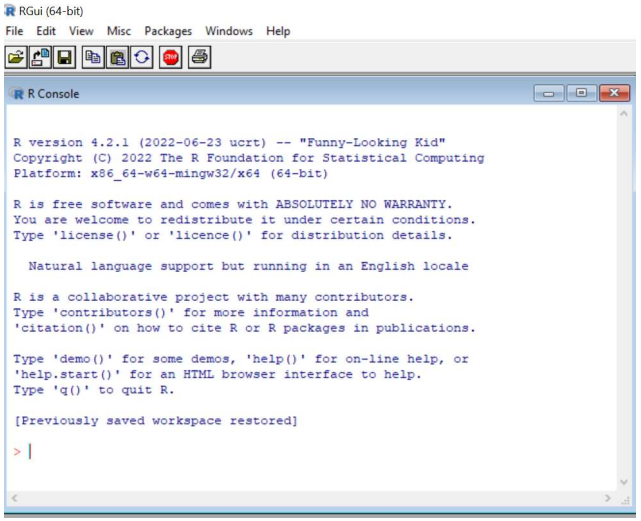
Teaching Block 1, 2024

What we will cover in this lecture

- We will introduce the software of R and RStudio
- We will learn about the basic objects and operations
- We will learn how to write a simple function in R with control flow statements
- We will see how R facilitates a functional paradigm with call-by-value semantics
- We have a brief look at lazy evaluation
- We will give a few signposts for where to learn more.

What is R?

R is a programming language designed for Statistical Computing



R provides a fantastic ecosystem for:

a) Graphics and data visualization
b) Efficient data wrangling
c) Statistical inference
d) Machine learning

It is free, open-source, and supported by a vast and active online community of contributors! (with around 20000 packages developed)

Other similar languages: Python, Julia

What is RStudio?

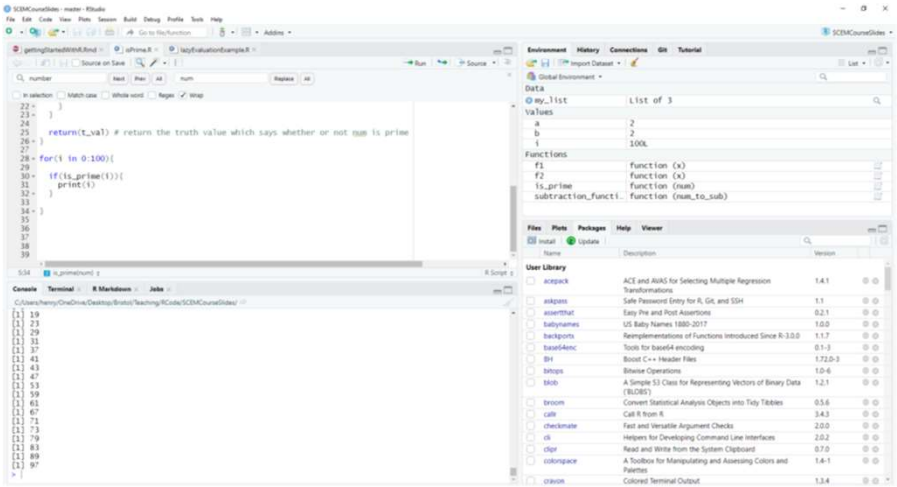
An integrated development environment for R that includes

- a console with a command line interface
- a code editor that supports syntax highlighting, code completion, smart indentation, direct code execution
- tools for debugging, plotting, workspace management, …

It is free and open-source

Reproducible analysis via knitr & R

Convenient interface for version control via Git



Installing R and RStudio

You can install both R and RStudio in Windows, Linux or Mac OS X.

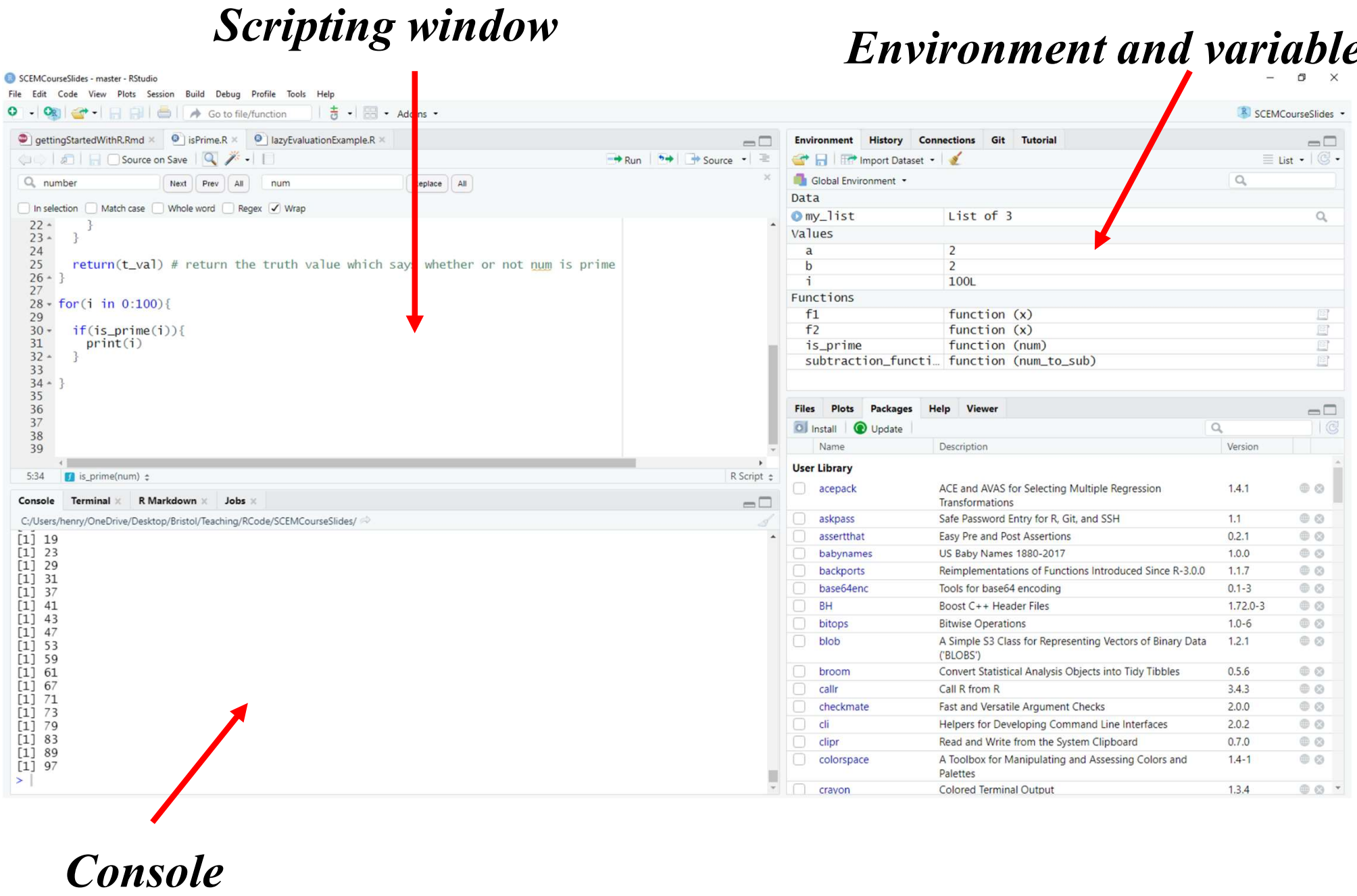
First download and install R from the Comprehensive R Archive Network:

https://www.r-project.org/

Then download and install RStudio

https://www.rstudio.com/download

Start your first R session in RStudio



Vectors

```
x <- c(3,7,4,2,1,2,-4,-5) # vector of numbers (use "<-" for assignment )
x
```

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

```
y <- seq(5) # A vector of numbers generated as a sequence
y
```

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

```
x[3] # You can access an element of a vector like this
```

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

```
x[c(2,3)] # Or several elements like this
```

```
## [1] 7 4
```

```
x[1:4] # Or the first four elements like this
```

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

Vectors

```
z <- c("Bristol", "Bath", "London") # You can have a vector of strings
z
```

```
## [1] "Bristol" "Bath"    "London"
```

```
w <- c(TRUE,FALSE,TRUE,FALSE) # Or a vector of Booleans
w
```

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

```
a <- c(TRUE, 3, "Bristol") # You can't have a vector of mixed type!
a
```

```
## [1] "TRUE"    "3"       "Bristol"
```

```
mode(a) # You can test the type like this
```

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

Matrices

```
M <- matrix(seq(10), 2, 5) # You can generate a 2 by 5 matrix
M
```

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

```
M[2,3] # The third element of the second row can be accessed directly
```

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

```
M[,4] # Or we can inspect the entire four coloumn
```

```
## [1] 7 8
```

```
is.vector(M[2,]) # We can check that a selected row or coloumn is a vector
```

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

Lists

```
first_list <- list(TRUE, 3, "Bristol")  # lists can be of mixed type
first_list
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] "Bristol"
```

```
second_list <- list( t_value=TRUE, num_value=3, city = "Bristol") # lists members can be named like a dictionary

second_list$t_value
```

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

```
second_list$num_value
```

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

Data frames

Data frames are powerful objects for representing and manipulating tabular data.

```
city_name <- c( "Bristol", "Manchester", "Birmingham", "London") # vector of city names
population <- c(0.5,0.5,1,9) # vector of populations

first_data_frame <-data.frame(city_name,population) # we can generate a data frame like this
first_data_frame
```

```
##    city_name population
## 1    Bristol        0.5
## 2 Manchester        0.5
## 3 Birmingham        1.0
## 4     London        9.0
```

Unlike matrices, in data frames,

- columns are named
- different columns may be of different type

However, the cells within a column must be of the same type.

Arithmetic operations

addition (+), subtraction (-), multiplication (x), division (÷), exponentiation, logarithmic functions...

```
(((4+2-1)*4)/2)^2 # Arithmetic operations - addition, subtraction, multiplication, division, exponentiation etc..
```

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

```
a<-matrix(sample(1:10, 6, replace=T),2,3) # a random 2 by 3 matrix
b<-matrix(sample(1:10, 6, replace=T),2,3) # a second random 2 by 3 matrix
a*b # this performs element wise multiplication
```

```
##      [,1] [,2] [,3]
## [1,]   15   49   15
## [2,]    6   20    9
```

```
a%*%t(b) # t(b) computes the transpose of b and %*% performs matrix multiplication
```

```
##      [,1] [,2]
## [1,]   79   49
## [2,]   65   35
```

Boolean operations

NOT (complement), AND (conjunction), OR (disjunction), XOR (exclusive disjunction), ...

```
a<-c(TRUE,TRUE,FALSE,FALSE) # a vector of Booleans
b<-c(TRUE,FALSE,TRUE,FALSE) # another vector of Booleans

!a # not a
```

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

```
a&b # a and b
```

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

```
a|b # the inclusive or between a and b
```

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

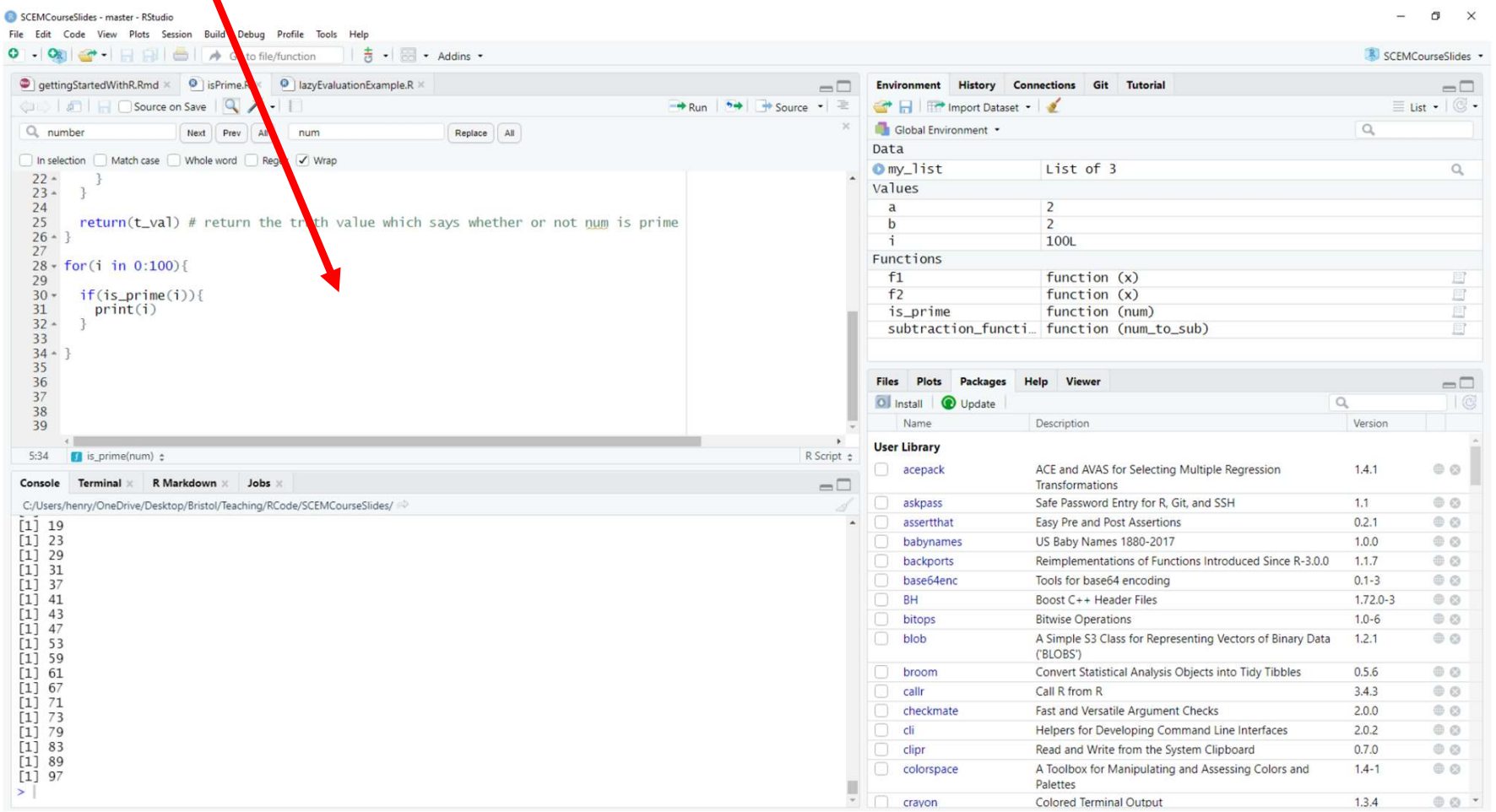
```
xor(a,b) # the exclusive or between a and b
```

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

Your first R script

Creating a R script: File --> New File --> R Script

Scripting window



How to define and call a function

Example: define a function called *is_prime*

```
is_prime <- function(num){

# Function which takes as input a positive integer and outputs Boolean - TRUE if and only if the input is prime.

  stopifnot(is.numeric(num),num%%1==0,num>=0) # Stop if the input is not a positive integer

  t_val <- TRUE # Initialise truth value output with TRUE

  if(num<2){

    t_val<-FALSE # Output FALSE if input is either 0 or 1

    }else if(num>2){

      for(i in 2:sqrt(num)){ # Check possible divisors i no greater than sqrt(num)

        if(num%%i==0){
          t_val<-FALSE
          break     # if i divides num then num is not prime

      }
    }
  }

  return(t_val) # return the truth value which says whether or not num is prime
}

is_prime(39) #Now we can use our function to check if 39 is prime.
```

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

Call-by-value semantics

In R, arguments in functions are passed with call-by-value semantics.

The value of a variable, but not its address, is passed to the function

```
a<-seq(5,2) # Create a vector

demo_func_1 <- function(x){

  x[2]<- -10 # Set the second value of the input to -10
  print(x)

}

demo_func_1(a) # Apply demo_func_1 to a
```

```
## [1]   5 -10   3   2
```

```
a # Note that the value of a is unchanged.
```

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

This facilitates a functional programming style with limited side effects.

Lazy evaluation

In lazy evaluation, a symbol can be defined (for example, in a function), but it will only be evaluated when it is needed

```
> no_input <- function(x){
+     100
+ }
> print(no_input())
[1] 100
```

x is a symbol, it is not evaluated until a statement requires so

A more complicated example (num_to_sub):

```
subtraction_function <- function(num_to_sub){
  output_function <- function(x){
      return (x-num_to_sub)
  } # a function with input x and output x minus num_to_sub
  return(output_function) #output this function
}

a<-1 # initialise a
f1 <- subtraction_function(a) # construct a function which subtracts a
print(f1(2)) # evaluate function at 2
```

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

```
a<-2 # modify a
print(f1(2)) # doesn't change the function
```

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

Lazy evaluation

```
subtraction_function <- function(num_to_sub){
  output_function <- function(x){
      return (x-num_to_sub)
  } # a function with input x and output x minus num_to_sub
  return(output_function) #output this function
}

a<-1 # initialise a
f1 <- subtraction_function(a) # construct a function which subtracts a
print(f1(2)) # evaluate function at 2
```

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

```
a<-2 # modify a
print(f1(2)) # doesn't change the function
```

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

Lazy evaluation enables efficiency but has some surprising consequences.

```
b<-1 # now initialise a new variable b
f2 <- subtraction_function(b) # construct a function which outputs b
b<-2 # change the value of b
print(f2(2)) # evaluating the function reveals that the second choice of b was used.
```

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

How can we learn more?

Almost every R function has an associated help function which can be accessed via

```
> ?name_of_function
> help(name_of_function)
```

A fantastic resource to learn more about R is the Swirl package

```
> install.packages("swirl")
> library(swirl)
```

Another great resource is StackOverflow for R:

https://stackoverflow.com/questions/tagged/r

What we have covered

R and RStudio installation

Basic objects vectors, lists, matrices & data frames.

Basic arithmetic operations and Boolean operations

Defining and calling functions; call-by-value semantics;
lazy evaluation;

learning more with the help function, Swirl, StackOverflow

University of BRISTOL

Thanks for listening!

Dr. Rihuan Ke

rihuan.ke@bristol.ac.uk

Statistical Computing and Empirical Methods
Unit EMATM0061, MSc Data Science