1.3 R Programming Basics

R Programming Basics

Objectives:

- Understand basic R programming principles

Contents:

- Variables/objects
- Data types and data structures
- Indexing/slicing
- Arithmetic in R
- Conditional statements (ifelse)

Variables/objects

You will often hear R discussed as an 'object oriented' language. Almost everything in R is either an object or a function. In R, we can create objects using `=` or `<-`,

note that in R, the `c()` function is used for combining and allows us to concatenate everything in the brackets to create a single object:

```
numbers = c(10, 7, -4, 8) # concatenate these numbers to an input
numbers_1 <- numbers # create an object with <-
```

We can then use functions on the objects

```
> mean(numbers) # get the mean of 'numbers'
[1] 5.25
```

But what **is** an object?

Well, the short answer is almost anything! An object can be a number, a dataset, a table, a list, an image, or a statistical test. Objects come in many different shapes and sizes in R. There are simple objects which represent single numbers, **vectors** (like our `numbers` object above) which represent several numbers, more complex objects like **dataframes** which represent tables of data, and even more complex objects such as algorithms or statistical test outputs.

Any objects we create has specific characteristics, commonly known as **attributes**. For instance, the `numbers` vector has a length (4), while a linear regression object may have many attributes such as an R^2, β, a p-value and others.

Data types

Ok, this part can be a little bit dull but in order to understand R you need a strong understanding of data types and data structures and how to use/not use them.

Some common data types in R are:

- **Numeric**
  Numeric variables are simply numbers. The clue is in the name!

We can use the `class` function to check the class of an object

```
> x = 10

> class(x)
[1] "numeric"
```

- **Character**
  R can also store strings. A string/character is specified by using quotes.

```
> p = 'hello world'

> p
[1] "hello world"

> class(p)
[1] "character"
```

- **Logical**
  In a logical or boolean variable, there are two predefined possibilities, TRUE and FALSE

We can also create logical variables. Below we ask R if 10 is less than 5, which is obviously not true, therefore `x` is FALSE.

```
> x = 10<5 # check if 10 is less than 5

> x
[1] FALSE

> class(x)
[1] "logical"
```

- **Factors**
  Factors are the data objects which are used to categorise data and stored as distinct levels. They can store both strings and integers (whole numbers, such as 10). By definition, a factor is a column which have a limited number of unique values. Like participant ID, gender or any other group. They are extremely useful in data analysis.

- **Missing (NA)**
  The value `NA` is used to represent missing values in an input in R. An `NA` can be assigned to a variable directly to create a missing value, but to test for missing values, the function `is.na()` must be used. R propagates missing values

throughout computations, so often, computations performed on data containing missing values will result in more missing values. Some of the basic statistical functions (like `mean`, `min`, `max`, etc.) have an argument called `na.rm`, which, if set to `TRUE`, will remove the `NA` from your data before calculations are performed.

Note, for all data types R has neat functions for checking and converting datatypes. The conversions start with 'as', such as `as.numeric()` or `as.character()` and the checking functions start with 'is' such as `is.numeric()` or `is.character()`.

```
> x = 10 # assign 10 to an object called 'x'
> class(x)
[1] "numeric"
> as.character(x) # convert it to a character
[1] "10"
```


Exercise 1:

Add (+) a numeric to a character and inspect the error output. Can you find a workaround?

HINT: the `as.numeric()` function could be useful here.

Data structures

In addition to data types, R also has some **data structures**. These data types are introduced below, but first, let's introduce a **very important** concept, the dollar (`$`) sign. The `$` allows us to extract elements by name from a named list or dataframe. For example:

```
x <- list(a=1, b=2, c=3)

x$b # extract the object called b
[1] 2
```

Note, there are four forms of the extract operator in R: `[`, `[[`, `$`, and `@`. We'll touch on the square brackets (`[`, `[[`) later.

R has many data structures including:

- **Vector**
  A vector in R is a combination of several scalars (single numbers) in a single object. For example, the numbers from one to ten could be a vector of length 10. For example:

```
> vec = c(1:10) # create a vector called vec

> length(vec) # check the length of vec
[1] 10
```

  There are many ways to create vectors. We can use `runif()` to generate random uniform numbers, `rnorm()` to generate numbers from a normal distribution.

  We can then use functions on these vectors. Two common ones are `unique()` and `table()`.

```
> vector_of_letters = c('a', 'b', 'c', 'd')

 # the unique function allows us to check the unique values>
> unique(vector_of_letters)
[1] "a" "b" "c" "d"

# or table to count values in the vector
> table(vector_of_letters)
[1] vector_of_numbers
a b c d
1 1 1 1
```


  Exercise 2:

  Use the `c()` function to create a coin with two outcomes ('H', 'T'). Next use the `sample()` function to flip the coin 10 times. How many of each did you get?

  HINT: the help function could be useful here, do you want to set the replacement value to TRUE or FALSE?

- **Matrices & Dataframes**
  Scalar and vector objects are useful, but they soon become insufficient for many data tasks. Thankfully, R has objects that allow large data structures to be managed and manipulated: **matrices** and **dataframes.** Matrices and dataframes are comparable to spreadsheets in Excel or data files in SPSS. They are comprised of rows and columns (i.e. 2-Dimensional), compared to vectors (1-Dimensional).

  Both matrices and dataframes are similar, but they aren't the same. While a matrix can contain either character or numeric columns, a dataframe can contain both numeric and character columns. Dataframes are more flexible (but more computationally expensive). Most real-world datasets contain numeric (e.g. age, height, weight) and character (e.g. gender, socioeconomic status) data which will be stored as a dataframe in R.

  Matrices and dataframes are just combinations of vectors, so R has some useful functions for creating matrices and data frames. `cbind()` and `rbind()` both create matrices by combining several vectors of the same length. `cbind()` combines vectors as columns, while `rbind()` combines them as rows.

```
# create vectors
x <- 1:5
y <- 6:10
z <- 11:15

# Create a matrix where x, y and z are columns
cbind(x, y, z)
     x  y  z
[1,] 1  6 11
[2,] 2  7 12
[3,] 3  8 13
[4,] 4  9 14
[5,] 5 10 15

# Create a matrix where x, y and z are rows
rbind(x, y, z)
   [,1] [,2] [,3] [,4] [,5]
x    1    2    3    4    5
y    6    7    8    9   10
z   11   12   13   14   15
```

  Matrices can either contain numbers or characters. If you try to create a matrix with both numbers and characters, it will turn all the numbers into characters:

```
# Creating a matrix with numeric and character columns will make everything a char
acter:
cbind(c(1, 2, 3, 4, 5),
      c("a", "b", "c", "d", "e"))
     [,1] [,2]
[1,] "1"  "a"
[2,] "2"  "b"
[3,] "3"  "c"
[4,] "4"  "d"
[5,] "5"  "e"
```

We can create **Dataframes** with the data.frame() function. For example, we create a dataframe of exam scores below and then inspect the data frame with the `str()` function. We will explain `str()` later, but for now just know that is an important function for inspecting the structure of an object.

```
exam_score <- data.frame("student" = c(1, 2, 3, 4, 5),
                         "sex" = c("m", "m", "m", "f", "f"),
                         "exam_score" = c(73, 57, 93, 93, 73))


> str(exam_score)

'data.frame':
5 obs. of  3 variables:
$ student   : num  1 2 3 4 5
$ sex       : Factor w/ 2 levels "f","m": 2 2 2 1 1
$ exam_score: num  73 57 93 93 73
```

- **Lists**
  In R, a list is a very powerful object. You will use lists more and more as you become proficient programmers.

  Unfortunately, a vector, matrix or dataframe has limited storage because their size is fixed (i.e. two vectors of different lengths can't be joined together). The solution to this problem is to use a `list()`. A list can store *anything*. You can have a list that contains several vectors, matrices, images, dataframes of any size or even other lists. We can create a list as follows:

```
# Create a list with vectors of different lengths
random_list <- list(
```

```
  "l1" = c('hello', 'world'), # item 1
  "l2" = c('hello', 'world', 1, 2, 4), # item 2
  "l3" = rnorm(n= 35), # item 3
  "l4" = 'this is a random list') # item 4

# check the structure with the str() function
> str(random_list)
List of 4
$ l1: chr [1:2] "hello" "world"
$ l2: chr [1:5] "hello" "world" "1" "2" ...
$ l3: num [1:35] 0.879 0.815 1.335 -0.961 -0.891 ...
$ l4: chr "this is a random list"

# extract the second element
> random_list[2]
$l2
[1] "hello" "world" "1"     "2"     "4"

# dollar sign
> random_list$l3

[1]  0.87871606  0.81465749  1.33457195 -0.96138212 -0.89145622  0.55214355 -0.9332270
0
[8]  0.07774538 -1.28520251  0.57243284 -0.06939105 -0.54662245 -0.09202019 -0.1590317
7
[15]  0.90370566 -1.00602077  0.41332782 -0.84219270 -1.23102574 -0.80526330 -0.655475
55
[22]  0.96110153 -0.26834846  0.82878870 -0.25676086  0.28468226  0.61926121 -1.351780
29
[29]  1.27399966 -0.71227228 -0.78177779  0.05805996 -1.00259436 -0.56896605 -1.691886
53

# double square brackets
> random_list[[4]]
[1] "this is a random list"
```

Slicing/indexing

Often we will want to select certain columns and rows from a dataframe, rather than the whole dataset. Indexing allows us to define the rows and columns we want to be returned. To do this, use the notation `df[row, col]`. For example:

```
# Return row 5 but nothing else
df[5, ]

# Return column 5 but nothing else
df[, 5]

# Rows 1:3 and column 2
df[1:3, 2]
```

```
# Rows 1:3 and column 2:7
df[1:3, 2:7]

# Rows 1:3 and columns 1, 3, 7, 9
df[1:3, c(1, 3, 7, 9)]

# Drop columns 1, 3, 4
df[,-c(1,3,4)]
```

We can also use a technique called logical slicing. This allows us to return only the instances that evaluate to `TRUE`. Let's demonstrate this with the exam data we created earlier, select only the males (saving as a new object), then select those scoring above 80. Note the `,` after our conditional statement and be sure to use `==` to check if the condition is met.

```
exam_score <- data.frame("student" = c(1, 2, 3, 4, 5),
                         "sex" = c("m", "m", "m", "f", "f"),
                         "exam_score" = c(73, 57, 93, 93, 73))

exam_scores_males =  exam_score[exam_score$sex == "M", ] # get instances where sex ==
 'M'

exam_scores_more_80 =  exam_score[exam_score$exam_score > 80, ]# get instances where s
core > 80

exam_scores_males_less_60 =  exam_score[exam_score$sex == "M" &
exam_score$exam_score > 80, ] # get instances where score > 80 and sex == 'M'
```

Arithmetic

R makes it easy to do arithmetic operations. They include Addition, Subtraction, Division, Multiplication, Exponent, Integer Division, and Modulus. If performed on a vector, arithmetic is performed member-by-member.

For example:

```
a = c(1, 1, 5)
b = c(1, 8, 3)

a+b
```

```
[1] 2 9 8
```

If we apply a single arithmetic operation to a vector, the result will be a new vector with the operation applied to each member of the vector.

For example, if we divide the vector `a` by 10:

```
a = c(10, 20, 30)

a/10

[1] 1 2 3
```

Recycling rule:

Something to be careful of is the ***Recycling rule.*** If two vectors are of unequal length, the shorter one will be recycled in order to match the longer vector. For example, the following vectors `u` and `v` have different lengths, and their sum involved recycling values of the shorter vector `u`.

```
u = c(10, 20, 30)
v = c(1, 2, 3, 4, 5, 6, 7, 8, 9)

u + v
[1] 11 22 33 14 25 36 17 28 39
```

Below is a useful table of common arithmetic operators:

Arithmetic operators to know

Aa OPERATORS	OPERATION	EXAMPLE
+	Addition	15 + 5 = 20
−	Subtraction	15 − 5 = 10
*	Multiplication	15 * 5 = 75
/	Division	15 / 5 = 3

Aa OPERATORS	OPERATION	EXAMPLE
%/%	Integer Division	16 %/% 3 = 5. If you divide 16 with 3 you get 5.333, but the Integer division operator trims the decimal values and outputs the integer
^	Exponent	15 ^ 3 = 3375 (It means 15 Power 3 or 103).
%%	Modulus	15 %% 5 = 0 (Here remainder is zero). If it is 17 %% 4, then result = 1.

Exercise 3:

Create a vector of all of the individual digits of your birthdate (this will be 8 digits). Save this in an object called `my_birthday_digits`. Next, multiply each element of by the mean of `my_birthday_digits` and save this object as `my_birthday_digits_mult`
Next, apply modulo 2 to `my_birthday_digits_mult` and save in `my_birthday_digits_mult_mod` Lastly, extract the second element and save it in `result`.

Conditional statements

Another important thing to know if you are to be a strong R user is conditional statements. Conditional statements are expressions that perform different computations or actions depending on whether a predefined condition is `TRUE` or `FALSE`. They are incredibly powerful means of efficiently performing an action. Going into detail is slightly beyond this course, but we want to introduce one very powerful function for vectors (and dataframes), called `ifelse()`.

So how does `ifelse()` work?

```
ifelse(test_expression, x, y)
```

Here, `test_expression` is the condition you want to test, for example, is age > 21. This returns a vector with the same length as `test_expression`. This vector has elements of `x` telling us if the corresponding value of `test_expression` is `TRUE` or from `y` if the

corresponding value of `test_expression` is `FALSE`.

Here's an example of how we use `ifelse` we are using it to create a simple classification of ages.

```
age_df = data.frame(ages = c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100))

age_df$classification = ifelse(age_df$ages > 50,"less young","young")

# check the result
> print(age_df)
   ages classification
1    10          young
2    20          young
3    30          young
4    40          young
5    50          young
6    60     less young
7    70     less young
8    80     less young
9    90     less young
10  100     less young
```

Exercise solutions

```
# Solution 1
> x=4

> y='4'

> x+y
Error in x + y : non-numeric argument to binary operator

> x+as.numeric(y)
[1] 8
```

```
# Solution 2
> coin = c('h', 't')

> flips=sample(coin, 10, replace = T)

> flips
[1] "t" "h" "h" "h" "t" "t" "t" "h" "t" "t"

> length(flips[flips == 'h'])
[1] 4
```

```
> table(flips)
flips
h 4
t 6
```

```
# solution 3
> my_birthday_digits = c(1,8,0,9,1,9,9,2)
> my_birthday_digits_mult = my_birthday_digits * mean(my_birthday_digits)
> my_birthday_digits_mult_mod = my_birthday_digits_mult %% 2
> result = my_birthday_digits_mult_mod[2]
> print(result)

[1] 1
```

Summary

You learnt:

- What an object is and how to differentiate data types and data structures
- How to extract an object from a dataframe and create a subset of a dataset
- How to do basic arithmetic with R
- How to use `ifelse`

Additional exercises

1. Create the dataframes below (data_1 & data_2):

	id	age	sex
1	1	10	M
2	2	20	F
3	3	30	M
4	4	40	F
5	5	50	M

	id	age	sex
1	6	15	M
2	7	25	F
3	8	35	M
4	9	45	F
5	10	55	M

2. Bind the two dataframes to create a new dataframe (data_3)

3. Create a new column in data_3 which returns 0 if the age is even or 1 if the age is odd

4. Create 4 new data frames from data_3:
Even aged men (1) /women (2), odd aged men (3)/women (4)

You can find the solutions to these exercises in the github repository, or the project folder

---

Materials used in this course can be cloned directly by clicking here. Alternatively, you can view the repository online:

> *https://github.com/RJODRISCOLL/Introduction-to-R*

For any help and advice, We can be contacted at:

> *R.ODriscoll@leeds.ac.uk* *pspjo@leeds.ac.uk*