Getting started with R

Introduction

Programming allows you to automate and calculate using a particular grammar called a programming language. This is not a course about programming. It is a course that utilises programming and will demand you to read and write small snippets of code. Because of its relevance to data science we shall use the “R” programming language.

What we are going to assume that you have programmed before. You may have hated the experience. This should be different in that we are programming with a purpose. That purpose is to spare us hours of using a calculator. We are not expecting you to be a good R programmer at the end of this course. Learning R is a journey, not a destination.

Goals

In this tutorial we are going to learn

- What makes R different to other programming languages you might already know
- What variables are and their ability to store multiple values
- What the basic data types are
- What operations we can do with variables
- How missing values are handled
- How to write an R function

RStudio

There is a software tool for writing and executing R code. This is called RStudio which is available on the university computers if you are curious and want to drive in at the deep end. The rest of us will avoid using RStudio and instead focus on less complex ways of writing and running R code.

Below is a box containing some R code. Notice that the comments (bits that play no part in the code) are shown in light green. Comments begin with the hash character (#). You should press the “Run Code” button to execute this code.

R-code Start Over Run Code

What have you learned?

That's it, you have run your first R code. You now know several things:

- How to create a line comment
- How to create a end-of-line comment
- How to print things

Next we will learn about variables

Variables

In mathematics, a variable is an unknown quantity that we represent with a letter.
In programming, a variable is a named thing that is a place-holder for something else.

In R we can assign a variable (myVar) the numeric value 27 using the the code below. In order to show that we have done this successfully, we shall print the variable in the following statement. Press "Run Code" to execute these two statements

R-code Start Over Run Code

In R there are several ways to achieve what you want to do.

Are there other ways to assign values to variables?

Answer

Case sensitive

Variables in R are case sensitive. That means **myVar** and **myvar** would be separate variables. This is a common mistake, especially for beginners.

What do you expect the following code will print out? Run it and check whether you were right.

R-code Start Over Run Code

Did you notice that variables can be place-holders for text as well as numbers?

Valid variable names

Not any name can be used as a variable name. Some names, like "print", might be valid but hugely ambiguous because there is also a function called "print". In R a variable must begin with a letter (a to Z) or underscore (_) and may contain letters, underscores, digits and dots. Variable names cannot contain spaces.

Suppose you really, really, really wanted a variable that breaks these rules. How would you do this?

Answer

Next we will look at how many values a variable can hold.

Vectors

Variables can hold any number of values, which includes 1 (as a special case) so a variable can contain a single value. In R, variables typically hold more than a single value. This is very convenient because in statistics data normally consists of the series of values.

In other programming languages, variables normally contain a single value unless they are a special type called an array. In R everything is array-like. In R we use the term “vector” to describe array-like variables.

One way to create a vector of values is the `c()` function.

R-code Start Over Run Code

You can interpret the `c()` function as meaning “column” even though the correct interpretation is that the variable is a “vector” (which mathematically speaking, goes down rather than across).

Notice the [1] the prefixes the printed output. What is that doing there? We talk about that next.

Print layout

When we print vectors they are printed across the page like words in a book, to save space.

What is going on with the [1]? The “[number]” represents the index of the first element (i.e. number, character, factor or logical) on that line.

Suppose a vector was 100 long. How would you locate the 76th item. Let’s try that

R-code Start Over Run Code

Using the [??] indices it is possible to locate the 76th case as “4.52055981”

Random numbers

What was the stuff with `set.seed()` and `runif()` ? What was that about?

`runif()` is an R function that randomly samples from a continuous uniform distribution (https://en.wikipedia.org/wiki/Continuous_uniform_distribution). It is an abbreviation of "**r**andom **unif**orm". It is called a pseudo-random sample because it is not truly random. I know that the next number it will generate is going to be 2.872463
If the numbers were truly random, there would be no way to know the value in advance. Let's check:

R-code Start Over Run Code

The `set.seed()` function controls the particular sequence that will be generated. Any number can be used. We used 99 in the previous code but we could have used any integer. Reusing the same integer generates the same sequence.

Why do we not generate truly random numbers rather than pseudo-random numbers?

Answer

We will encounter `runif()`, `rnorm()` functions again and again.

Lists

Suppose we consider grocery shopping lists. We might think of things on the list as being the things we need to buy. For example:

- Coffee
- Milk
- Biscuits (chocolate)
- Carrots
- Rice (basmati)
- Juice (apple)

This looks like a character variable. Let's create this in R and print it.

R-code Start Over Run Code

Suppose we want to reuse this list from one week to the next. It becomes about whether an item is actually needed this week. How might we do this in R?

We want to shift from a list of characters to a logical list *that is labelled*.

R-code Start Over Run Code

We have created a list of TRUE/FALSE entries against the names of the shopping items. We seem to be saying that we don't need carrots, rice or juice this week. That was easy. This style of using a function (in this case `c()`) where you supply a parameter label and a value for each parameter is rather different to other programming languages.

Lists of lists

Suppose we have kept the shopping lists we have used for the last 12 months. Also, suppose we labelled these lists with the date. The code below is how we might populate variables in R with this data. This code assumes we never misspell a grocery item or describe the same thing differently - put that objection out of our heads for now.

R-code Start Over Run Code

Obviously we are going to stop at 3 lists because this is a teaching exercise and not an endurance test.

How do we combine these things into a list of lists?

R-code Start Over Run Code

Hold on, earlier we used `c()` and now we have switched to `list()`. What is going on?

A “vector” is a special list in which the values are:

- basic things like numbers, characters, factors, logical
- all of the same type

A “list” is a more general collection of anything (including other lists) and the entries can be different types. In this case the entries are all logical vectors.

We were able to label our shopping items; can we label our shopping lists? Let’s try:

R-code Start Over Run Code

What have you learned?

- Everything in R is a sequence of things
- A vector is a special list of the same type of basic thing (e.g. all numeric values)
- Lists of things can be labelled - this is optional and can be useful for navigating through the list

What else can variables “hold”? We will explore that next.

Data types

So far we have encountered numeric and text variables. Let’s explore these in more detail and discover other data types.

Numeric

A numeric variable can contain zero, 1 or many numbers. These can be integers or decimal numbers.

When we print decimal numbers the decimals places shown will be driven by a global setting. This global setting is called `options("digits")` and applies to all printing.

R-code Start Over Run Code

```
options(digits=7)
print(3.14159265358979323846264338327950288419716939937510)
options(digits=3)
print(3.14159265358979323846264338327950288419716939937510)
```

```
[1] 3.141593
```

```
[1] 3.14
```

When we print variables, another global setting is relevant. This limits how many of the elements to actually print. This global setting is called `options("max.print")` and applies to all printing.

R-code Start Over Run Code

```
options("max.print")


```

```
$max.print
[1] 99999
```

Notice, by the way that there was no print function and yet the result was printed. It turns out that if you calculate a result and do not assign it to a variable, R guesses that you will want to see the result printed.

Using the previous “digits” example can you write the code to change the “max.print” option to 50? Press “Submit Answer” when you are done.

R-code Start Over Hint Run Code Submit Answer

```
options("max.print"=50)
```

Smashing!

Character

Character variables should NOT be thought of as a sequence of letters that make up a character string. In many programming languages this is the paradigm - not so in R. In other programming languages this data type is common called an “array of string”

In R, a character vector is sequence of variable-length character strings.

R-code Start Over Run Code

```
V = c("Chapter 1", "The best day of my life.")
length(V)

```

```
[1] 2
```

The number of characters in each of the variable-length strings can be found using the `nchar()` function:

R-code Start Over Run Code

```
nchar(V)

```

```
[1]  9 24
```

To split a string into words we can use strsplit()

R-code Start Over Run Code

```
strsplit(V, split = " ")

```

```
[[1]]
[1] "Chapter" "1"

[[2]]
[1] "The"   "best"  "day"   "of"    "my"    "life."
```

Notice that this is a naive implementation as punctuation and new-line characters also determine string splitting. This issue takes us into the fascinating territory of *regular expressions* but sadly that is not going to happen today.

Notice that the output is a list of character vectors. We can turn this into a single character vector with the `unlist()` function.

R-code Start Over Run Code

```
words <- unlist(strsplit(V, split = " "))
words

```

```
[1] "Chapter" "1"       "The"     "best"    "day"     "of"      "my"
[8] "life."
```

✔❌ Logical

Logical variables are often generated by testing a condition upon another data type.

For example:

- testing whether a factor vector is a particular level
- testing whether a number vector is greater than some value
- testing whether a character vector contains some sub-string

We can also switch between numeric and boolean provided we use conditional statements. Here we are testing whether values of A are greater than 10.

R-code Start Over Run Code

```
A = c(19,21,4,17,5)
A > 10

```

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

Factor

Factors are variables that contain labels that have a finite range of values.

For example, in an electoral register, voters are assigned to an electorate. The data that records voters near Christchurch might look like the following table:

Name	Electorate	Year
John Smith	Ilam	2016
Mary Smith	Ilam	2016
Peter Jones	Wigram	2016
Michael Tibbet	Banks Peninsula	2012
Michael Tibbet	Christchurch Central	2016
Jane Tibbet	Christchurch Central	2016
Simon Hall	Christchurch East	2012

The *Electorate* column is a bit like a character variable. But it is also different. The variable cannot contain just any words - it must contain valid electorate names. We call variables like this “Factors”. These factors are a basic data type in R.

The *Electorate* variable would be created as a factor with the following code.

R-code Start Over Run Code

```
X = c("Ilam", "Ilam", "Wigram", "Banks Peninsula", "Christchurch Central", "Christchurch
Electorate = factor(X)
Electorate
```

```
[1] Ilam                Ilam                Wigram
[4] Banks Peninsula     Christchurch Central Christchurch Central
[7] Christchurch East
5 Levels: Banks Peninsula Christchurch Central Christchurch East ... Wigram
```

Notice that when printed, a factor is indistinguishable to a character variable. Why then, do we bother with factors?
Let's count the reasons:

1. Integrity is enforced. If I add further voters to the data, I will get an error message if I use an electorate name that is not one of the existing ones.
2. Memory efficiency. The variable is stored internally as a list of integers (as well as a translation table) so that memory is minimised.
3. Self description. The variable can be asked whether it is a factor. This affects how you might want to display or process the information.
4. Cardinality. The variable can instantly reveal the number of levels without having to scan through all the values.

How do we reveal cardinality? The code beneath prints the number of levels assigned to the factor.

R-code Start Over Run Code

```
1 nlevels(Electorate)
2
3
```

```
[1] 5
```

You may be surprised that R variables can be place-holders for functions. Not many other languages allow this.

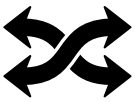
Consider the code below and run it. We are assigning the *myVar* variable to be a place-holder of the `print()` function.

R-code Start Over Run Code

```
1 myVar = print
2 myVar("This actually works")
3
```

```
[1] "This actually works"
```

There will be more on functions later. Next we will turn our attention upon operators - things that manipulate our variables.



Mixed data types

How do we attempt to mix data types? What happens when we try?
One way to mix data types is to merge pairs of vectors using the `c()` function. This happens on line 3 of the code blocks that follow.

Let's try mixing numbers and character vectors together and then enquire what the new variable's type is.

Before you run this code, make a prediction as to the data type of the merged variables.

R-code Start Over Run Code

```
num_var = c(12, 56, 17.3, 6.9, 22)
char_var = c("alpha", "beta", "gamma")
new_var <- c(num_var, char_var)
new_var
class(new_var)
```

```
[1] "12"    "56"    "17.3"  "6.9"   "22"    "alpha" "beta"  "gamma"
```

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

What was the mixed data type?

Answer

Notice that the `class()` function returns the base type of the vector.

Let's try mixing logical and character vectors together and then enquire what the new variable's type is. Before you run this code, make a prediction as to the data type of the merged variables.

R-code Start Over Run Code

```
log_var = c(T,F,F,T)
char_var = c("alpha", "beta", "gamma")
new_var <- c(log_var, char_var)
new_var
class(new_var)
```

```
[1] "TRUE"  "FALSE" "FALSE" "TRUE"  "alpha" "beta"  "gamma"
```

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

What was the mixed data type?

Answer

Merging logical and character vectors produces a character vector

---

Let's try mixing logical and numeric vectors together and then enquire what the new variable's type is. Before you run this code, make a prediction as to the data type of the merged variables.

R-code Start Over Run Code

```
log_var = c(T,F,F,T)
num_var = c(12, 56, 17.3, 6.9, 22)
new_var <- c(log_var, num_var)
new_var
class(new_var)
```

```
[1]  1.0  0.0  0.0  1.0 12.0 56.0 17.3  6.9 22.0
```

```
[1] "numeric"
```

What was the mixed data type?

Answer

Merging logical and numeric vectors produces a numeric vector

---

We have not merged all the possible vector types but these are the common ones that might need to be merged.

Next we shall look at operators in R.

+ − ÷ × Operators

Operators manipulate and transform our variables. They are data type specific. For example, it is an error to try to multiply to character variables together.

Numeric operators are processes like **multiplication, addition, power, log, sin**, etc
Logical operators are processes like **and, or, not** etc
Character operators are processes like **concatenation, splitting, length** etc

The challenge with R is to adjust to using these operators in the context of a sequence of things.

Numeric operators

Numeric operators work with numeric vectors. Suppose we populate variables A and B in the following way:

R-code Start Over Run Code

```
1 A = c(19,21,4,17,5)
2 B = c(5,11,9,13,19)
3
```

Let's compute A times B. In R, the multiplication operator is the asterisk character "*".
What does the multiplication of A and B look like? What does it produce?

Have a go at writing the statement below. Press "Submit Answer" when you are done.

R-code Start Over Hint Run Code Submit Answer

```
1 A * B
2
3
```

```
[1]  95 231  36 221  95
```

Splendid! This is your first piece of hand-written R

The result is an element-wise multiplication. By extension, you can guess that division will be an element-wise division.

Let's compute A cubed. In R, the power operator is the hat character "^".
What does the cube of A look like? What does it produce? Have a go at writing the statement below.
Press "Submit Answer" when you are done.

R-code | Start Over | Hint | Run Code | Submit Answer

```
1 A ^ 3
2
3
```

```
[1] 6859 9261   64 4913  125
```

Absolutely fabulous!

The result is an element-wise cubing.

Let's compute A to the power of B. What does A to the power of B look like? What does it produce?

R-code | Start Over | Hint | Run Code | Submit Answer

```
1 A ^ B
2
3
```

```
[1] 2.476099e+06 3.502775e+14 2.621440e+05 9.904578e+15 1.907349e+13
```

You should be proud.

The result is 19^5, 21^11, 4^9 etc

This should be starting to look sensible. It is not always so obvious though.

Suppose I want the largest value in A:

R-code | Start Over | Run Code

```
1 max(A)
2
3
```

```
[1] 21
```

We are not done yet. Suppose I want the largest value in A or B. There are two ways to interpret that:

1. The largest value of max(A) or max(B)
2. The largest value of each pair of values from A and B

The former can be calculated this way (although other ways are possible too).

R-code Start Over Run Code

```
max(max(A),max(B))
```

```
[1] 21
```

The latter can be calculated as a pair-wise max function called `pmax()`

R-code Start Over Run Code

```
pmax(A,B)
```

```
[1] 19 21  9 17 19
```

What we have avoided using is `max(A,B)` . So what does this produce?

R-code Start Over Run Code

```
max(A,B)
```

```
[1] 21
```

Clearly this is the same as `max(max(A),max(B))` which does not do the pairwise operation - that requires `pmax(A,B)`

Let's consider what happens when the lengths of the two vectors is not the same. Firstly lets use a vector of length 1.

In the code below, the value 10 is treated as a numeric vector of length 1.

R-code Start Over Run Code

```
B * 10
```

```
[1]  50 110  90 130 190
```

This should be identical to B * c(10). Let's check…

R-code Start Over Run Code

```
B * c(10)
```

```
[1]  50 110  90 130 190
```

When the lengths do not match the shorter vector is repeated until it reaches the right length. In this case the 10 was repeated five times to match the length of B.

Now let's try multiplying by c(10,100)

R-code Start Over Run Code

```
B * c(10,100)


```

```
Warning in B * c(10, 100): longer object length is not a multiple of shorter
object length
```

```
[1]   50 1100   90 1300  190
```

Notice two things:

1. The calculation proceeded and produced: [1] 50 1100 90 1300 190
2. There was a warning objecting to what you were trying to do: *longer object length is not a multiple of shorter object length*

Since, 5 is not a multiple of 2, the rule about repeating the shorter variable does not make much sense - hence the warning.

Logical operators

Numeric operators work with logical vectors. Logical vectors can contain a sequence of two values: TRUE or FALSE. These can be abbreviated to T and F. Note (as always) that these are case sensitive.

Suppose we populate variables J and K in the following way:

R-code Start Over Run Code

```
J = c(T,F,F,T,F)
K = c(F,T,F,T,T)

```

The characters "!" "&" "|" have a special meaning in logical expressions (as they do in most programming languages).

To logically "and" I and J together we use the "&" operator.

R-code Start Over Run Code

```
J & K


```

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

To logically "or" I and J together we use the "|" operator.

R-code Start Over Run Code

```
J | K


```

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

To evaluate “not” J we use the “!” operator.

R-code Start Over Run Code

```
!J


```

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

How do we check if J has any TRUE values? The `any()` function is what we need to use.

R-code Start Over Run Code

```
any(J)


```

```
[1] TRUE
```

How do we check if K has all FALSE values? The `all()` function is what we need to use.

R-code Start Over Run Code

```
all(!K)


```

```
[1] FALSE
```

Notice that **any()** and **all()** only ever return a single value (i.e. a vector of length 1)

When we use a logical variable in a numeric expression it is tolerated just fine. We can think of TRUE being 1 and FALSE being 0. Do make sure that this effect is what you intend and not just a typographical error.

R-code Start Over Run Code

```
A = c(19,21,4,17,5)
J * A


```

```
[1] 19  0  0 17  0
```

This trick make it hard to follow the code so use this with caution.

Character operators

The big issue with character variables is how non-intuitive concatenation can be. Once you master this you have passed the difficult part.

R-code Start Over Run Code

```
V = c("Chapter 1", "The best day of my life.")
words = unlist(strsplit(V, " "))
words
```

```
[1] "Chapter" "1"       "The"     "best"    "day"     "of"      "my"
[8] "life."
```

Now we turn to concatenation. Suppose we wanted to join the words we just split, back into a single string using a dot as a word separator. We need to employ a function called `paste()`

R-code Start Over Run Code

```
paste(words, collapse = ".")


```

```
[1] "Chapter.1.The.best.day.of.my.life."
```

The collapse parameter controls whether the concatenation is going to be element wise or vector wise. In this case we are collapsing the list into a single value.

We could have chosen to append something to the end of each word (say a dash).

R-code Start Over Run Code

```
paste(words, "-", sep = "")


```

```
[1] "Chapter-" "1-"       "The-"     "best-"    "day-"     "of-"      "my-"
[8] "life.-"
```

What have you learned?

- Variables honour element-wise behaviour - be aware of scenarios related to `max()` and `pmax()`.
- Length mismatches do NOT fail - warnings are there to guide you.
- Logical variables can be used in numeric expressions.
- Character variables are lists of strings not necessarily letters/digits etc.
- `paste()` will do concatenation but you need to think about how the concatenation should work.

Missing values

When a value in a vector is not present, it is said to be missing. This may be due to several reasons:

1. The value is not currently available e.g. Customer declined to answer a question in a survey

2. The value is not applicable (by definition) for this case e.g. shoe size of a double foot-amputee

When data arrives (say, via CSV files), missing values can be implied by a gap in the data or by a place-holder (like -99). In either case the right way to record this in R is with the special value **NA**

Suppose a variable has a missing value:

R-code Start Over Run Code

```
Var = c(12,4,56,8,9,NA,41,30)
Var

```

```
[1] 12  4 56  8  9 NA 41 30
```

We can detect this with a function

R-code Start Over Run Code

```
is.na(Var)

```

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

This is how we might make this message more readable using `paste()` and `sum()`:

R-code Start Over Run Code

```
paste("Variable Var has", sum(is.na(Var)), "missing values")

```

```
[1] "Variable Var has 1 missing values"
```

In this case, the intermediate logical vector was treated as 1 for TRUE and 0 for FALSE when summing.

Operations with NA

Any computation involving NA generates a result of NA.

R-code Start Over Run Code

```
Var * 5

```

```
[1]  60  20 280  40  45  NA 205 150
```

R-code Start Over Run Code

```
mean(Var)

```

```
[1] NA
```

If we want to calculate a statistic like `mean()` that is potentially tolerant of missing values, we can often instruct it to be tolerant:

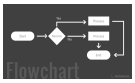
R-code Start Over Run Code

```
1 mean(Var, na.rm = TRUE)
2
3
```

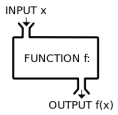
```
[1] 22.85714
```

What have you learned?

- Missing values are built into R
- Operations honour NA.
- Use `is.na()` to locate missing values



Program control



Functions

Conceptually, a function is a block of reusable code that receives zero, 1 or more parameters and may return some result.

For example, `print()` does not return anything.
In contrast, `max()` returns a numeric vector of length 1.

As you have noticed, a function's parameters are always contained within matching round-brackets.

Many functions are provided already in R. It is also possible to write your own functions in R.

We shall write a function next.

Looping

Like all programming languages R supports looping. There is much written on the internet about avoiding looping in R at all cost. This should not concern you - looping may not be the fastest mechanism that R provides but it is much times clearer than the alternatives.

Suppose we want to create a function that negates every second entry in a numeric vector. We shall call our function `negate2nd()`. The code below is our first attempt. There are many features of this code that will be unfamiliar. There are clarifications following the code.

R-code Start Over Run Code

```
1 negate2nd = function(x) {
2   for (i in 1:length(x)) {
3     if (i %% 2 == 1) {
4       x[i] = -x[i]
5     }
6   }
7   x
8 }
```

When this code is run, nothing happens. Why?

Answer

We are creating the function by running it. We won't see anything useful until we use the function.

Some code clarifications:

- %% is the modulus operator 1%%2 is 1, 2%%2 is 0, 3%%2 is 1 etc
- == is an equality test - the values of the variables must be equal
- The for loop is contained by its brackets - there is no "endfor"
- The if statement is contained by its brackets - there is no "endif" but there are *else if* and *else* elements available.
- 1:length(x) is shorthand for creating c(1, 2, 3…number of elements in x)
- The for loop is going to take consecutive elements from the list *1:length(x)*
- x[i] relates to the single entry from x at position i
- x[i] can be used to both read and write to x
- x is the last line executed in the function so this is what the function will return

Let's try it now.

R-code Start Over Run Code

```
1 myvar = c(9,5,7,3,0,6,6,1)
2 negate2nd(myvar)
3
```

```
[1] -9  5 -7  3  0  6 -6  1
```

Whoops, it seems like it is negating from the first element. **Can you fix the function?** Press "Submit Answer" when you are done.

R-code Start Over Hints Run Code Submit Answer

```
negate2nd = function(x) {
  for (i in 1:length(x)) {
    if (i %% 2 == 0) {
      x[i] = -x[i]
    }
  }
  x
}
myvar = c(9,5,7,3,0,6,6,1)
negate2nd(myvar)
```

```
[1]  9 -5  7 -3  0 -6  6 -1
```

You should be proud.

Wrapup

What have you learned?

- How to assign variables
- How functions are constructed
- How loops operate
- How conditional statements operate

In case this on-line tutorial is not available in future, you may want to keep a PDF copy of this material for reference purposes. Export as PDF

Topic not yet completed...