Portfolio Report 1: An Introduction to R

Jake Spiteri

02/01/2020

R is a high-level programming language widely used for statistical computing. Base R provides users with a large variety of statistical and graphical tools, which are easily extended via the use of packages.

R is an interpreted programming language. This means that R users execute code via a command-line interpreter and code is interpreted into low-level instructions one line at a time. Integrated development environments (IDEs) such as RStudio provide easy access to such an interpreter through the use of a console. Interpreted languages are looked at in more depth in a later report, but it is important to note that interpreted languages such a R are slow in comparison to compiled languages. Due to this, a majority of the base R code is written in low-level compiled languages — approximately 50% of the base R code is written in C/C++, 30% in Fortran, and 20% in R.

How to run R code

There are many ways that we can work with R. We can work directly from a terminal, in which we can interactively input commands which are then interpreted as R code, or we can directly run an R script in the terminal (a text file with extension '.R'). Once R is installed one can initialize a session in the terminal by simply typing `R`, and we can run an R script with the command `Rscript path/to/Rscript`. Running lines of code in the terminal is useful for quickly testing ideas, but a script is preferred for analyses which require more thought as a script provides access to and stores all code that has been run.

We can improve our workflow by using an IDE such as RStudio. RStudio greatly improves upon the R coding experience in a number of ways:

1. **Panes** — RStudio's default interface consists of four 'panes' which provide access to a script window, console window, environment window, and a files/plots/help window. These panes are also *highly* customizable. Panes allow us to efficiently switch between writing a script, running code in the console, and viewing plots or function documentation.
2. **Text Editor** — RStudio's text editor provides many of the features you would expect from an IDE, such as syntax highlighting, code completion, and find and replace with regular expressions.
3. **Autocompletion** — A particularly useful key in RStudio is the Tab key. When typing a function or object name, pressing the Tab key will autocomplete the name; once a function has been specified e.g. `lm()`, pressing the Tab key once more will list all of the possible arguments for the given function.

Fundamentals

We now look at some fundamentals of R programming. This will only be a brief introduction but should cover a wide-array of topics.

Variable names and types

In R a value can be assigned to a variable using the assignment operator `<-`. It is possible to use `=` to assign a value to a variable, but it is not recommended. If the specified variable already exists, this will overwrite the existing variable. For example,

```r
x <- 1
x
```

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

```r
x <- 2
x
```

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

As seen above, simply assigning a variable does not produce an output. We can call the variable to implicitly print the variable's value.

Variables in R can take on many different data types. These data types can be seen in Table 1 below.

Data Type	**Example**
Character	"abc"
Numeric	123, 1.23
Integer	1L, 123L
Logical	TRUE, FALSE
Complex	1 + 2i

Table 1: Data types

As seen in the table we can also assign strings and logical values to variables.

```r
x <- "hello world"
x
```

```
## [1] "hello world"
```

```r
x <- TRUE
x
```

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

A variable name can consist of letters, numbers, dots, and underscores.

Arithmetic

As you would expect, R is capable of performing many operations. Some are shown below.

```r
# addition
1 + 1
```

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

```r
# subtraction
3 - 1
```

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

```r
# multiplication
1 * 2
```

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

```r
# division
6 / 3
```

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

```r
# exponentiation
2^2
```

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

```r
# integer division
5 %/% 2
```

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

```r
# integer modulus
7 %% 5
```

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

Of course, R implements the order of operations which we all know. We can change this behavior by using parentheses.

```r
2 + 4 / 2
```

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

```r
(2 + 4) / 2
```

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

Conditional Statements

R can execute certain commands conditional upon a logical value by using `if/else` statements. These statements are commonly of the form

```
if (<logical value>) {
  <evaluate when logical value is TRUE>
} else {
  <evaluate when logical value is FALSE>
}
```

We can also nest `if` statements to obtain more complicated logic. A simple example is shown below.

```r
x <- TRUE
if (x) {
  print("x is TRUE")
} else {
  print("x is FALSE")
}
```

```
## [1] "x is TRUE"
```

```r
x <- FALSE
if (x) {
  print("x is TRUE")
} else {
  print("x is FALSE")
}
```

```
## [1] "x is FALSE"
```

The curly brackets are not always necessary but are always recommended in order to avoid bugs. For example

```r
# setup
x <- 4
```

```
# brackets are not needed
if (x<5) print("x is less than 5")
```

```
## [1] "x is less than 5"
```

```
# brackets are needed
if (x>5)
  print("nice")
  print("x is more than five")
```

```
## [1] "x is more than five"
```

The logical value for the last `if` loop is `FALSE`, so we would expect nothing to be printed. The error occurs because the last `if` statement is equivalent to

```
if (x>5) {
  print("nice")
}
print("x is more than five")
```

```
## [1] "x is more than five"
```

Whereas what we actually wanted was

```
if (x>5) {
  print("nice")
  print("x is more than five")
}
```

Relational and logical operators

We often want to generate a `TRUE` or `FALSE` value dependent on some variables, like the `x < 5` seen above. The relational operators are

Operator	Meaning
$<$	Less than
$>$	Greater than
$<=$	Less than or equal to
$>=$	Greater than or equal to
$==$	Equal to
$!=$	Not equal to

Table 2: Relational operators.

We may want to combine logical values using logical operators. The logical operators can be found in Table 3 below.

Operator	Meaning
!	Not
&	And
|	Or
&&	Short-circuit and
\|\| | Short-circuit or |

Table 3: Logical operators.

Functions

We can easily create functions in R. Functions allow us to simplify our code by putting repeated computations into a function. The syntax used to create a function is

```
<function_name> <- function(<arguments>) {
  <computation>
}
```

For example, suppose we wanted to easily find the p-norm of a vector.

```
lp_norm <- function(x, p) {
  norm <- sum(abs(x)^p)^(1/p)
  return(norm)
}

# l1 norm
lp_norm(c(-1,1), p=1) # should be 2
```

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

```
# l2 norm
lp_norm(c(3,4), p=2) # should be 5
```

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

It is good practice to `return` the final argument you want to output.

Data structures: vectors, matrices, and lists

There are many data structures that we frequently use in R. Below is a table categorizing some of the most common. Some structures are homogeneous in that every element within them has the same data type, and some are heterogeneous in that their elements can be of different types.

	Homogeneous	**Heterogeneous**
1D	Atomic vector	List
2D	Matrix	Data frame
nD	Array	

Table 4: Common data structures.

Below we generate some of the above data structures.

```
## vectors
# generate sequence from 1 to 10
x <- 1:10
print(x)
```

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

```
# get the length of x
length(x)
```

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

```
# get the data type of x
typeof(x)
```

```
## [1] "integer"
```

```r
# generate new x of type
x <- c("a", "b", "c")
print(x)
```

```
## [1] "a" "b" "c"
```

```r
# get the data type of x
typeof(x)
```

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

```r
# index into x
x[1]
```

```
## [1] "a"
```

```r
# reassign an index
x[2] <- "d"
print(x)
```

```
## [1] "a" "d" "c"
```

```r
# we cannot change the data type
# below we try to input a numeric 1, but R inputs a STRING "1"
x[2] <- as.numeric(1)
print(x)
```

```
## [1] "a" "1" "c"
```

```r
typeof(x[2])
```

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

```r
## matrices
# generate a matrix
x <- matrix(c(1, 0, 0, 1), 2, 2)
x
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

```r
# check dimensions
dim(x)
```

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

```r
# get first row of x
x[1, ]
```

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

```r
# assign colnames
colnames(x) <- c("column 1", "column 2")
x
```

```
##      column 1 column 2
## [1,]        1        0
## [2,]        0        1
```

```r
# check type of elements
typeof(x)
```

```
## [1] "double"
```

```
# many R functions can be applied elementwise
sin(x)
```

```
##      column 1 column 2
## [1,] 0.841471 0.000000
## [2,] 0.000000 0.841471
```

```
max(x)
```

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

```
# try to input a string as the top left entry
x[1,1] <- "test"
x
```

```
##      column 1 column 2
## [1,] "test"   "0"
## [2,] "0"      "1"
```

```
# we have successfully replaced the [1,1] element but the data type
# of every entry has changed
typeof(x)
```

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

```
# what happens if we try to find the maximum value now
max(x)
```

```
## [1] "test"
```

```
## lists
# generate a list
x <- list(numbers = c(1, 2, 3), letters = c("a", "b", "c"))
x
```

```
## $numbers
## [1] 1 2 3
##
## $letters
## [1] "a" "b" "c"
```

```
# index into a list using $
x$numbers
```

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

```
# index into a list using [[ ]]
x[[1]]
```

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

```
# index into a vector in a list
x[[1]][1]
```

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

```
# create new entries in a list on the fly
x$letters_backwards <- rev(x$letters)
x
```

```
## $numbers
```

```
## [1] 1 2 3
##
## $letters
## [1] "a" "b" "c"
##
## $letters_backwards
## [1] "c" "b" "a"
```

```
# reformat a name in a list
names(x)[3] <- "letters backwards"
x
```

```
## $numbers
## [1] 1 2 3
##
## $letters
## [1] "a" "b" "c"
##
## $`letters backwards`
## [1] "c" "b" "a"
```

```
# check the type of x
typeof(x)
```

```
## [1] "list"
```

```
# check the type of a named vector in x
typeof(x$numbers)
```

```
## [1] "double"
```

```
# lists can even contain functions
x <- list()
x$eval <- function(x, y) {
  return(x*y)
}

# use function
x$eval(2, 2)
```

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

Lexical scoping

R uses lexical scoping, which determines how a free variable within a function obtains a value. Consider the following function:

```
m <- function(x) {
  return(x * y)
}
```

This function has only one argument `x`. The function also requires another variable `y` which has not been defined locally in the function and is not given as a formal argument — this is called a *free variable*. The scoping rules implemented by a language determine how such a free variable obtains its value.

Lexical scoping means that the values of free variables are searched for in the environment in which the function was defined. If they are not found here, then R searches for their value in the parent environment.

After reaching the top-level environment, R will look down the search list (found using `search()`). If R arrives at the empty environment without finding values for free variables, we will get an error.

Let's implement the lp-norm again and demonstrate lexical scoping.

```
# create a constructor function
make_lp_norm <- function(p) {
  norm <- function(x) {
    return(sum(abs(x)^p)^(1/p))
  }
  return(norm)
}

# use constructor to make l1 norm function
l1_norm <- make_lp_norm(1)
l1_norm(c(1,1)) # should be 2
```

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

Let's look at the code for `l1_norm`.

```
l1_norm
```

```
## function(x) {
##     return(sum(abs(x)^p)^(1/p))
##   }
## <environment: 0x562acc673ae8>
```

We see that it depends on a free variable `p`. Recall that the value of the free variable is obtained from the environment in which the function was defined. In this case, `p` is equal to 1. Let's check this by exploring the environment of the `l1_norm`.

```
# list objects in l1_norm's environment
ls(environment(l1_norm))
```

```
## [1] "norm" "p"
```

```
get("p", environment(l1_norm))
```

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

A simpler example would be to define a variable in the global environment, and then use it in a function.

```
n <- 2
square <- function(x) {
  return(x^n)
}

# test the square function
square(2)
```

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

```
square(3)
```

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

A problem occurs when we expect a variable to be defined within a function, but it is defined in the global environment. Continuing with the above example, it seems plausible that we may reassign the value of `n` at some point in our analyses.

```r
# reassign n
n <- 3

# the square function no longer works as expected
square(2)
```

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