Table of Contents

README before starting!

Welcome to Environmental Data Science's introducing R session. In this session we start from scratch and assume no prior knowledge of R. We will cover the very basics of the R language. Things like Rstudio and ggplot2 will be covered in future sessions.

Fill in the blanks as you go to test your understanding, and keep this handy as a reference for basic terminology.

Used R before?

It could still be worth going through the whole notebook or just particular sections; there might even be something you skipped over first time round.

On the other hand if you understand and can explain each of the following terms and how they relate to R, then perhaps this tutorial is not for you. In this session we will go over:

- variable and operators
- data types
- Data structures
- lists
- factors
- dataframes
- matrices
- functions
- applying functions.

By the end of the session you should know what each of these terms refers to.

As an aside, the *dictionary* of programming languages borrows heavily from logic and mathematics. This is especially true for R, which is a language designed with applied mathematics (statistics) in mind. Knowing the mathematical meaning of a word, e.g. a matrix is (imprecisely) a grid of numbers, helps with knowing what R is aiming for.

Let's get started!

1. Variables and operators

1.1. Variables

A **variable** holds a value, and has a name. The value of a variable can change, hence being variable and not constant. In R we assign a variable using the following symbols <- , ->, = , such symbols are referred to as **operators**, there are lots of different types of operators (more on those later!).

Let's name our variable *x* and assign it the value 1:

```
x<-1
```

If we want to show the contents of x, we will have to use the *print()* function

```
print(x)
```

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

The name of a variable should be unique, and can consist of letters, numbers, underline (_) or dots (.). The variable should start with a letter or a dot (this dot cannot followed by a number). Variables cannot start with a number or underline. Can you find which variable name is incorrect?

- x.1<-1
- x_2<-2
- .x<-3
- 4y<-5
- _z<-6
- _2<-7
- .2a<-8*

Try each statement out here:

When something is wrong R will print an error message to screen. Make sure to read this message carefully, and to digest and understand the computer speak. If you do not understand then Google away!

In the example above the final four statements are incorrect: * 4y<-5
* *z<-6*
* 2<-7
* .2a<-8*

and result in an "unexpected symbol" error message.

1.2. Data types

Every variable has a **data type**. The data type of a variable is its defining quality, it is what it is! In R, there are several basic data types that we typically use:

- character: a
- numeric: 2,0, or 5.6

- integer: 4L (the L means R will store 4 as integer)
- logical: TRUE, FALSE

The above datatypes are referred to as *atomic* in R speak. We will sometimes drop the data part of the word and simply refer to the **type** of a variable when we talk about datatypes.

Create a variable of type character, and name it y. Print it out in the same box:

```
y <- "EnvDataSci"
print(y)
```

```
## [1] "EnvDataSci"
```

To show the type of a variable you can use **class()** command. What is the type of **x** from the above example?

```
class(x)
```

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

x is a different type ("numeric") to y.

1.3. Operators

We already met assignment operators above e.g. <-. But there are lots of other operators too. Operators connect variables by doing something to them. Perhaps the most obvious example is mathematical operations, which are performed by the **arithmetic operators**.

1.3.1.Arithmetic operators

We can add, substract, mutiply, divide, equal, expeonent variables in R using the following **arithmetic operators**: +, -, *, /, =, ^

```
math1=5+6*8-9+5/6
print(math1)
```

```
## [1] 44.83333
```

```
4^3
```

```
## [1] 64
```

1.3.2.Relational operators

Relational operators are used to compare variables. The output of such a comparison is a *boolean* i.e. a *TRUE* or *FALSE* value. We can use such operators to test statements such as is z greater than x? First we have to assign z a value:

```
z=20
```

now let's test our hypothesis:

```
z>x
```

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

here is a list of operators:

- ">" greater than
- "<" less than
- ">=" greater than or equal to
- "<=" less than or equal to
- "==" equal to
- "!=" not equal to

1.3.3. Logical operators

More logic here! You will notice that logic is a recurring theme, as computer languages are rooted in logic, 0s and 1s etc.

Logical operators can be used to chain operators together. Again the output is a Boolean *TRUE* or *FALSE*. For example, to test if x is greater than 0 and less than 10:

```
x>0 & x<10
```

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

There are 3 operators in the above **syntax** (bit of code). Two relational operators and one logical, which is the & symbol. There is also an or symbol: |.

How do you test if z is greater than 100 **or** greater than 1000?

```
z> 100 | z>1000
```

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

z has the value 20 which is neither greater than 100 nor greater than 100. The statement above is FALSE.

1.3.4.Miscellaneous Operators

There are also a few other operators that don't fit neatly into any other bracket. The colon operator **:** is very handy as it is used to generate a sequence of numbers:

```
x1 <- 1:10
print(x1)
```

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

The **%in%** operator can be used to check if the elements that occur in ** A ** also occur in ** B **

```
A<-1:10
B<-3:6
print(A%in%B)
```

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

2. Data structures

Put simply **data structures** are groups of variables. You can group variables of different data types, however the simplest data structures occur when you group variables of the same type. These simple structures are called **vectors**.

2.1. Vectors

A vector is a one dimensional array that can consist of numeric, character,or logical data. All the elements inside a vector should be a single data type only. A vector is formed using function **c()**.

Let's try the following out:

1. create a vector named as a, store letters of b to e into a
2. print a
3. check data type of a
4. check the structure of a.

```
#(1)create a character vector: a
a<-c('b','c','d','e')
#(2)print: a
print(a)
```

```
## [1] "b" "c" "d" "e"
```

```
#(3)check type of a
class(a)
```

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

```
#(4)check structure of a
str(a)
```

```
##  chr [1:4] "b" "c" "d" "e"
```

The structure command displays the structure of the object in question. In this case **a** is character vector.

You should try the following:

1. create a vector named as a1, store first 10 numbers into a1
2. print a1
3. check type of a1, and
4. structure of a1.

```
a1 <-  c(1:10)
print(a1)
```

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

```
class(a1)
```

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

```
str(a1)

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

In the above case a1 is a vector of type integer (whole numbers).

2.2. Matrices

If we extend our concept of a vector into two dimensions we end up with a 2D grid of the same type; this is a **matrix**. Matices are created by the matrix() function:

```
d<-c(1:20)
#create a matrix
matr1<-matrix(d,nrow = 5,ncol=4,byrow = T,
              dimnames =list(c('r1','r2','r3','r4','r5'),
                             c('c1','c2','c3','c4')))
matr1

##    c1 c2 c3 c4
## r1  1  2  3  4
## r2  5  6  7  8
## r3  9 10 11 12
## r4 13 14 15 16
## r5 17 18 19 20
```

Let's checkout the structure of our matrix:

```
str(matr1)

##  int [1:5, 1:4] 1 5 9 13 17 2 6 10 14 18 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:5] "r1" "r2" "r3" "r4" ...
##   ..$ : chr [1:4] "c1" "c2" "c3" "c4"
```

2.3. Arrays

An array is similar to a matrix, but can have more than two dimensions. Arrays are created with the **array()** function

```
x<-c('a1','a2','a3')
y<-c('b1','b2')
z<-c('c1','c2','c3','c4','c5')
mydata<-c(1:30)
array1<-array(mydata,c(3,2,5),
dimnames = list(x,y,z))
array1

## , , c1
##
##    b1 b2
## a1  1  4
## a2  2  5
## a3  3  6
##
## , , c2
##
```

```
##    b1 b2
## a1  7 10
## a2  8 11
## a3  9 12
##
## , , c3
##
##    b1 b2
## a1 13 16
## a2 14 17
## a3 15 18
##
## , , c4
##
##    b1 b2
## a1 19 22
## a2 20 23
## a3 21 24
##
## , , c5
##
##    b1 b2
## a1 25 28
## a2 26 29
## a3 27 30
```

2.4. Lists

Next we come to **lists**, these are different to matrices and vectors because they can store objetcts of different type.

Let's make a list containing lots of different types and display the results ...

```
alist<-list(c('b','c','d','e'),
            c('control','control','drought'),
            c(1:10),
            c(1L,3L),
            88,
            '%&()',
            c(0.3,2.34,6.57,0),
            3+9,
            c('4','9'),
            list(c(1,2,'g')))
print(alist)
```

```
## [[1]]
## [1] "b" "c" "d" "e"
##
## [[2]]
## [1] "control" "control" "drought"
##
## [[3]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[4]]
## [1] 1 3
##
## [[5]]
## [1] 88
##
```

```
## [[6]]
## [1] "%&()"
##
## [[7]]
## [1] 0.30 2.34 6.57 0.00
##
## [[8]]
## [1] 12
##
## [[9]]
## [1] "4" "9"
##
## [[10]]
## [[10]][[1]]
## [1] "1" "2" "g"
```

```
class(alist)
```

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

```
str(alist)
```

```
## List of 10
##  $ : chr [1:4] "b" "c" "d" "e"
##  $ : chr [1:3] "control" "control" "drought"
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : int [1:2] 1 3
##  $ : num 88
##  $ : chr "%&()"
##  $ : num [1:4] 0.3 2.34 6.57 0
##  $ : num 12
##  $ : chr [1:2] "4" "9"
##  $ :List of 1
##   ..$ : chr [1:3] "1" "2" "g"
```

2.5. Advanced data structures: Dataframes

It is no exaggeration to say that **Dataframes** are probably the most important data structure used in R. Dataframes are liked matrices, with the key difference being that we can mix up datatypes in the same object. This turns out to be incredibly useful in a number of standard data analysis problems, in short our data is never simple and dataframes take care of this.

Dataframes are generated by the *data.frame()* function. In a dataframe:

- the columns are variables
- the rows are observations (the row numbers will be object numbers)
- all the variables are of same length.

```
#create a dataframe
x=c(1:3)
y=c('a','b','c')
z=c(30,40,50)
df1<-data.frame(x,y,z)
#print df1
print(df1)
```

```
##   x y  z
## 1 1 a 30
## 2 2 b 40
## 3 3 c 50
```

```
#check the structure of df1
str(df1)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ z: num  30 40 50
```

Try to create a new dataframe called **mydata**. This dataframe should have 2 variables and 4 observations. Mix up the types of the variables:

```
x.1 <- c(1.2,1.3, 4.2, 99.0)
y.1 <- c("1.2","1.3", "4.2", "99.0")

mydata <- data.frame(x.1,y.1)
str(mydata)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ x.1: num  1.2 1.3 4.2 99
##  $ y.1: Factor w/ 4 levels "1.2","1.3","4.2",..: 1 2 3 4
```

notice that y.1 is type factor.

2.6 Additional data type: factors

A *factor* is a kind of advanced vector, that is useful in some analyses.

All the elements inside a factor are characters. We start by creating a normal character vector:

```
#create a vector
treatment<-c('control','control','drought','nutrient','drought')
#check the type of treatment
class(treatment)
```

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

Next we convert our character vector to a factor using the **factor()** function:

```
#create a factor
treatment.factor<-factor(treatment)
#check the type
class(treatment.factor)
```

```
## [1] "factor"
```

The difference between factors and character vectors, is that factors contain levels which can be examined using **levels()** function

```
#check levels of treatment.factor
levels(treatment.factor)
```

```
## [1] "control"  "drought"  "nutrient"
```

R also stores factors as numbers rather than characters, this has some consequences which might not be obvious. See here for more details on factors
https://swcarpentry.github.io/r-novice-inflammation/12-supp-factors/

3. Control flow: if statements and loops

Now that we have defined our basic structures (e.g. vectors) we can start to apply some fancier programming constructs. These fall under the broad category of Control Flow, as they can be used to direct your program.

In R, the body (main bit) of the control flow statement is usually places in curly braces *{ code here }*. (note howwever that these braces can be omitted if your statement only spans a single line).

3.1. The if statement

The **if statement** is one of the most widely used constructs in most programming langauge. Put simply it allows the program to decide whether to do something only if some condition has been fufilled.

For example if x is below some number, we want to print a statement that is true. To achieve this we would do the following:

```
x<-10
if (x<50) {print('X is smaller than 50')}
```

```
## [1] "X is smaller than 50"
```

We use the **else** keyword to chain together multiple clauses. This saves us retyping the whole construct over and over.

First change the value of x to 100, then combine the previous **if** statement with a similar **else** statement,

```
x<-100
if (x<50) {print('X is smaller than 50')
}else{print('X is greater than 50')}
```

```
## [1] "X is greater than 50"
```

We can also use a the *ifelse()* function to achieve a similar result to control flow statements. This works as follows:

```
z<-c(1,10,30)
ifelse (z==10|z>15,'yes','no')
```

```
## [1] "no"  "yes" "yes"
```

3.2. Loops

For loops are programming contructs used to repeat stuff over and over, in programming terms this is referred to as **iteration**. Iteration is at the heart of solving problems using computing, the computer is put to task to repeat tasks that are either boring or impossible for humans to accomplish in a reasonable time frame!

A for loop iterates over a sequence one element at a time. The for loop contains the *for* statement, a *variable* in some normal braces, and the body (main bit) of the loop is placed between the curly braces. The body is where the repetiton occurs.

If we take an example, here x.seq is a vector that we plan to iterate over, at each step (iteration) we want to add a new value (y) to the element in x that is the subject of the iteration. At each step we also print the variable.

```
x.seq<-c(1:10)
y<-2
for (i in x.seq){
 z<-y+i
 print(z)
}
```

```
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
```

Go ahead and write a for loop where we square "i in x.seq", and also print the resulting squared variable at each step.

```
x.seq<-c(1:10)
y<-2
for (i in x.seq){
 z<-i^2
 print(z)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

While loops are similar to for loops, except in these loops iteration continues until some condition is reached. In the following example we keep looping until our x variable hits the value of 8 (10-2), then the loop stops.

```
x<-2
while(x<10) {
  print('yes')
  x=x+2
  }
```

```
## [1] "yes"
## [1] "yes"
## [1] "yes"
## [1] "yes"
```

There is another type of loop called a *repeat loop*, these are similar to while loops but to stop the loop we use the keyword *break* and supply a condition within the loop itself:

```
x<-9

repeat{
    print (x)
    x <- x - 1
    if (x==8){
    break
    }

}
```

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

In most computing languages the above loops are similar in importance to if statements. R is a little different in this respect, as there are different methods of iteration that are just as powerful and sometimes preferable. These alternate iteration methods are the **apply** functions discussed in the next section of the course. For programmers coming from other langauges, it is important to remember that in R, an apply function will typically take the place of a for loop.

4. Functions and Applying functions

Functions are central to the way R works; think of them as the building blocks of larger programs. Just like building blocks, functions are elements of a larger whole. If you want to take the next step beyond writing simple scripts, then functions are your friend!

In simple terms, a function takes an input, does something to it, then returns something else which is the results of that something.
In R language, functions consist of *arguments* which are the input, the *body* which is the bit where things happen, and the *name*, which is self evident!

4.1. Built-in functions

Most things in R involve functions, typically these are built-in to the language or perhaps part of a package that you have installed.

When you print something to the screen then you are using the **print** function:

```
print("print is a commonly used function!")
```

```
## [1] "print is a commonly used function!"
```

You can take a look at the contents of a function by typing the function name, but missing out the paranthesis:

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x29a4000>
## <environment: namespace:base>
```

You can get information (help) on any function by placing the question mark operator at the front of a function (e.g "?function"), try it out with print:

```
#?print
```

The code inside the function is called the *body*, the input to the functions is referred to as the *argument*(s).

Sometimes there is more than one function that achieves a similar result. Can you think of another function that prints messages to the screen?

```
cat("Hello")
```

```
## Hello
```

4.2. User-defined functions

Built-in functions allow us to get stuff done, however user-defined functions allow us to build stuff!

Time to make a function. First we need our function to have a purpose. Maybe our instrument has a linear calibration function. The aim of our function is to calculate this calibration using our data i.e. takes an input value, and multiply by a calibration coefficient to yield the calibrated value.

```
cal.eq <- function(instrument.data,cal.coeff=2.4){
       instrument.data * cal.coeff
     }
```

Next we should breakdown the individual parts of our function. The name of the function is *cal.eq*, the body of the function is the part in between the curly braces. Our function has two input arguments, the first is *instrument.data*, and the other is our calibration coefficient *cal.coeff* which is set to a default value of 2.4.

Now try to test out the function with some data, x <- c(1:10)

```
output.x <- cal.eq(x)
print(output.x)
```

```
## [1] 19.2
```

The result of our function is in *output.x*, a new variable.

One important note about functions; the output of an R function is whatever is calculated on the last line of the body. In our case we only have one line so that is what we get back.

4.3. The apply family and anonymous functions

The *apply* family are a special group of functions that most R users are familiar with. In essence apply functions are similar to for loops, in fact in R when you find yourself reaching for a for-loop we often ask ourselves can we use apply instead?

Apply functions iterate (repeat) a function over a datastructure. Let's walk though an example. we can use the *paste* function to join two characters together e.g. 'hi ' and 'there'

```
paste('hi ','there')
```

```
## [1] "hi  there"
```

We could use apply to paste 'hi' to a vector of different words. We do this by defining a new *anonymous* function within the apply call. An *anonymous* function is simply a function with no name! Let's use the lapply function, as this is designed to work on vectors

```
sentenc.e <- c("This "," is a ", " setence." )


# now lets use apply to mess up our sentence!

nonsense <- lapply(sentenc.e, function(input) paste(input," oops ")  )
```

We have now turned out sentence into nonsense! However if you inspect the *class()* of nonsense you will notice that it is a list!

```
print(nonsense)
```

```
## [[1]]
## [1] "This   oops "
##
```

```
## [[2]]
## [1] " is a   oops "
##
## [[3]]
## [1] " setence.  oops "
```

```
class(nonsense)
```

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

This is because lapply returns a list. Lists are a little tricky to work with. To convert back to a vector you can use another function:

```
nonsense <- unlist(nonsense)
print(nonsense)
```

```
## [1] "This   oops "     " is a   oops "    " setence.  oops "
```

Now we have a character vector, just like our original sentence.

In R there are a bunch of apply functions, each one is intended for slighly different inputs and outputs. Here are a few that you may come across:

- lapply: input are vectors and lists. Outputs are list
- sapply: input are vectors and lists. Outputs are "user friendly"
- mapply: multivariate version of sapply. Outputs are "user friendly"

what is the difference in using sapply over lapply in the example above?

```
nonsense.s <- sapply(sentenc.e, function(input) paste(input," oops ")  )
print(nonsense.s)
```

```
##              This                is a            setence.
##     "This   oops "    " is a   oops " " setence.  oops "
```

```
str(nonsense.s)
```

```
##  Named chr [1:3] "This   oops " " is a   oops " " setence.  oops "
##  - attr(*, "names")= chr [1:3] "This " " is a " " setence."
```

sapply does the unlisting automatically.

5. Other stuff

We have only scratched the very basics of R in this tutorial, there is whole world of resources out there! Many of them free due to R's ethos as an open source language. For programmers coming from other languages, be sure to check out *The R inferno*. There are also a bunch of online tutorials for you to follow should you have time, Google is your friend here!

5.1 installing packages

Functions are bundled together in packages (libraries). Each package has a general theme e.g. ggplot2 is for improved plotting capabilites. You can download packages using the *install.packages* function.