INST0065 – Week 2: R practical

This week's practical

This week's practical is aimed to introduce and explain some of the basic capabilities and language characteristics of R. It is largely based on chapter 2 of Humanities Data in R[1]. Unfiortuantely, this book is not available online via the UCL library. We will refer to this book elsewhere, but will draw more heavily on other books that are available online.

Example commands are shown like this:

```
> 1 + 2
[1] 3
```

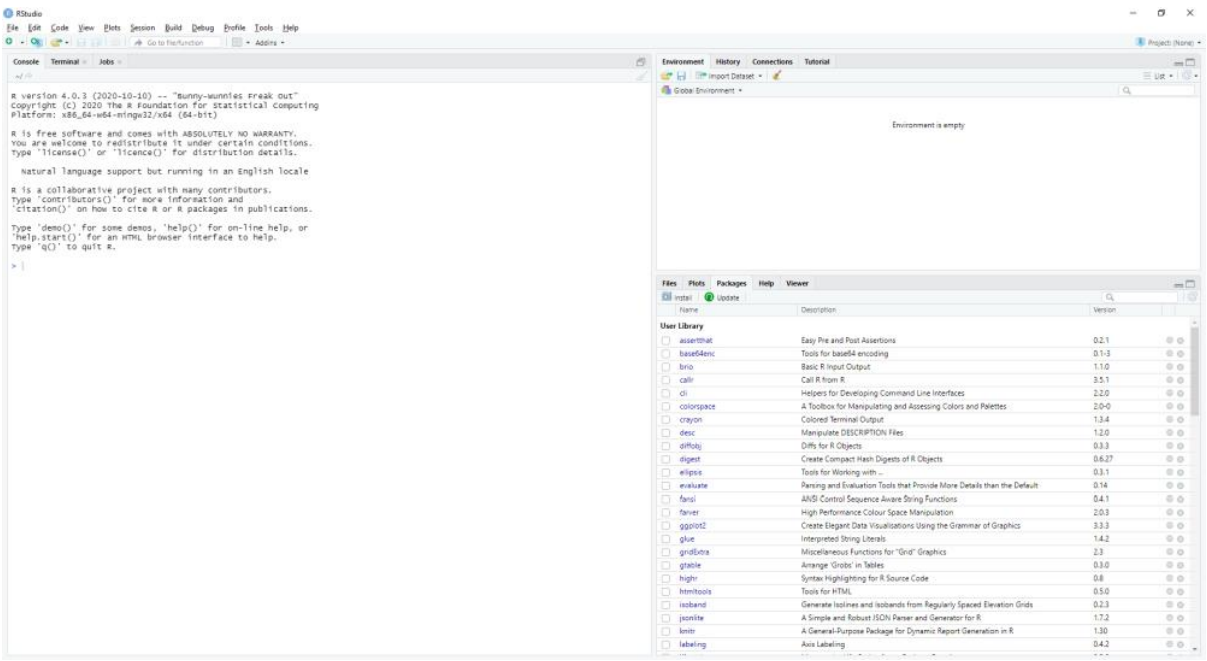
You should repeat these in the R console.

There will also be questions in the text that ask you to try typing further things at the prompt.

Starting R

It is assumed that you have installed R and RStudio, or have access to these via the Desktop@UCL Anywhere remote service. Please see last week's practical handout for details on how to do this if you have not done so already.

You should now start up RStudio. You will see a display similar to the one below.



The difference between R and RStudio

RStudio provides a front end to R; it is R that this is the language (and sets of libraries etc) that provides the capabilities we wish to use. If you were to just start up R (rather that RStudio), you would see a console display similar to that seen on the left hand side of the RStudio window.

[1] Arnold, T., & Tilton, L. (2016). Humanities Data in R: Exploring Networks, Geospatial Data, Images, and Text (Softcover reprint of the original 1st edition 2015). Springer International Publishing.

Using the console

The simplest way in which we can use R is to type expressions at the console prompt, such as this:

```
> 1 + 2
[1] 3
```

Here, the '>' symbol is the command prompt that shows you where to type; you should not type it yourself.

The second line of this example shows us the response – in this case, the result of the expression '1+2' that we have typed.

R supports a similar range of mathematical operators that you will have met in other programming languages, or in school maths lessons, and as with any other maths, there is an order of precedence in which operators are carried out. You may well be familiar from school with mnemonics such as 'BODMAS' or 'PEMDAS' (different countries tend to use different variants).

BODMAS stands for "**B**rackets **O**rder **D**ivision **M**ultiplication **A**ddition **S**ubtraction"

PEMDAS stands for "**P**arentheses **E**xponents **M**ultiplication **D**ivision **A**ddition **S**ubtraction"

'Order' and 'Exponents' both refer to exponentiation (i.e. raising a number to a power, applying a square root etc etc). It is worth noting that BODMAS and PEMDAS reverse the *apparent* order of multiplication and division, and it should be observed that in fact these two have the precedence, and are carried out left to right. Similarly, additional and subtraction have the same priority, and are also carried out left to right. In other words, if we see: "1 – 2 + 3" we simply carry this out left to right; we do *not* do 2+3 first, and then subtract it from 1.

Thus, in this example, (again from Arnold and Tilton chapter 2) we would do the part in brackets first '(2+17)', then the square root operation 'sqrt(2)' (although we do not need the result immediately), then the division operation, and then subtract/add from left to right.

```
> 1 / (2 + 17) - 1.5 + sqrt(2)
[1] -0.03315486
```

As with using logic statements in a programming language, we can put in additional brackets in order to make it clearer to ourselves hwo a calculation is carried out. It is important to remember however, that different patterns of brackets might give different overall results.

- Using R, compare the result of the following two expressions:
  - `1 / (2 + 17) - 1.5 + sqrt(2)`
  - `1 / (2 + 17) - (1.5 + sqrt(2))`

R operators

The most common operators are shown in the table below[2]

Operator	Description
+	addition
-	subtraction
*****	multiplication
/	division

[2] As with the following table, taken from https://www.statmethods.net/management/operators.html

^ or **	exponentiation
x %% y	modulus (x mod y) 5%%2 is 1
x %/% y	integer division 5%/%2 is 2

You will see that the basic operators are as you would expect. Note that R uses '%%' as a modulus operator; you may have previously used '%' in programming languages such as JavaScript. The modulus operator gives us the remainder of the left hand argument when it is divided by the right hand argument. Thus, for any number *x*, we can say that x is even if *x%%2* gives the result 0 – we have divided by 2, and have got no remainder.

The final operator in this table is integer division. You may have met this idea in some programming languages, but are less likely to have met it in everyday maths, where the distinction between integers (whole numbers) and real numbers (with a decimal part) is less common. Using integer division will return only the integer part of the answer, and discard any decimal fraction. Note that it does not round to the nearest integer.

- How do you think integer division will work for negative values?
- Using R, compare the result of the following two expressions:
  - `5 %/% 2`
  - `-5 %/% 2`

R also provides logical operators; again, you may have met these in other languages.

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

As with other languages, it is important to remember that we check for equality using '==' and not the single equals character '='. Whereas the numerical operators will return a number as their result, the logical operators will return a logical (TRUE or FALSE) value.

- Using R, try typing some expressions that use several of these operators

Obviously, using R as a simple calculator is very limited; we can also use a wide range of functions which will do far more for us than simply report a result.

Assignment

As well as directly carrying out expressions, we can also do assignment operations in R. This is shown in most books as '<-', and we shall use that here. It is also possible to use '=' for assignment (but, as

above, remember that assignment (=) is not the same as testing for equality (==). Using '<-' helps us to avoid confusing these two ideas.

```
> x <- 1 + 2
```

Note that unlike the expressions you have typed above, this does not show a 'result', it simply returns to the command prompt. We have assigned the result of 1 + 2 to the variable *x*.

We can see the value of a variable by typing its name:

```
> x
[1] 3
```

We can perform expressions on *x* that will give us the results we expect

```
> x/2
[1] 1.5
```

Note that the above example does not change the stored value of x. We can assign the result to another variable:

```
> y = x/2
> y
[1] 1.5
```

We can change the value of x by assigning the result to itself:

```
> x <- x/2
> x
[1] 1.5
```

Here, we have divided x by 2, then assigned that result to x, over-writing the original value of 3.

Classes

R consists of objects and data structures. We will meet these is more detail later on, but it is useful at this point (especially as we are following the Arnold and Tilton chapter) to know about the function 'class'. This tells us what the class of an object is. Using the above examples, we have created two objects *x* and *y*.

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

This tells us that 'x' is a numeric object. Note that functions (such as class) can use placeholder variables to refer to their parameter(s). Thus, we could say:

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

In this case, we are calling the class function, and saying that it has a parameter 'x', and that we are assigning to that parameter our object 'x'. Clearly, this example is confusing! The first x refers to the parameter (and only to the parameter – it does not refer to our object); the second 'x' refers to our object.

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

Here, we are looking at the class of our object 'y'; again, the 'x' in 'x=' is not connected to our object called 'x'.

So far we have only looked numeric objects, but these are not the only data types that R understands.

```
> xx <- "aa"
> class(xx)
[1] "character"
```

Here, we are creating an object 'xx', and assigning to it the value 'aa'. We then use 'class' to tell us what type of object is it, and get the answer 'character'.

Data types and data structures

R has the following data *types*:

- Numeric
- Character
- Logical
- Integer
- Complex (used for *complex numbers* in maths)

We have met numeric and character types above, and have also referred to logical values.

These data types can be held in a number of *data structures*; so far we have met the basic structure of a *vector*. All our examples above are vectors.

Vectors are similar to what most other programming languages term arrays; confusingly R also has a (different) data structure called an 'array'.

A vector is a collection of values in which each cell or element has the same data type (i.e. a vector of numeric values, a vector of character values etc). The examples we have used above have all been vectors with a single element. In the results, we have seen lines such as '[1] 3'. It is obvious that the '3' in this example is the result; the '[1]' tells us that we are looking at the first element in a vector.

An important distinction between R and other languages is that there is no concept of an atomic 'single' piece of data – everything is a vector (or one of the other data structures, all of which potentially contain multiple elements).

Another distinction between R and many other languages (but not all) is that the numbering of elements starts at 1, rather than 0.

Moving on, we can use the function 'c' to combine several values into a vector, and here we shall start to see more of what R does.

```
> vecObj <- c(1,10,100)
> vecObj
[1] 1 10 100
```

On the first line here, we are using c() to create a vector that has three values – 1, 10 and 100 – and then assigning that (using '<-') to vecObj. Just as with our simple example 'x <- 1+2' above, vecObj is created at this stage, we do not need a separate line to say that we are going to use a variable called 'vecObj'.

On the second line, we type the name of the object, and just as with the 'x' example, R tells us the contents of that object.

We can perform operations on vectors in the same way that we did with x (this is of course obvious: we have just said that 'x' is a single-element vector, so we've *already* been doing operations on vectors!).

```
> vecObj + 10
[1] 11 20 110
> vecObj + vecObj
[1] 2 20 200
```

On the first line, we have added '10' to our vector, and when we look at the results, we can see that R shows us all three elements, each of which have had '10' added to them.

On the second line, we have added the first element to itself, the second element to itself, and so on.

When we do operations on vectors, R will *recycle* components. Thus, in the first line, we can think of '10' as being a single value element (because that is what it is!). As R processes vecObj, it adds the two vectors together. In the first line, we go back to the beginning of the second vector ('10') for each new value of vecObj.

In the second line, we do not run out of inputs in the same way, so do not need to recycle. We take the first element of the left hand side, add to it the first element of the right hand side (left and righ hand side are both 'vecObj'), then take the second element of the left hand side and add to it the second element of the right hand side, and so on.

R will always recycle shorter vectors.

Now, consider this example from Arnold and Tilton

```
> c(1,2,3,4,5,6) + c(100,200)
[1] 101 202 103 204 105 206
```

We are adding together two vectors, both of which are created as expressions. Our results will be the length of the longer vector. In order to add them together, we recycle the second vector. So, we take '1' from the first vector, and add '100' from the second vector, then '2' from the first and add '200' from the second. We then take '3' from the first vector. We have used up all of the second vector values, so we return to the start of that vector and add '100'.

- In the above example, the two vectors 'nest' neatly. What happens if vectors are not a clear multiple number of elements of each other?
- Try this in R: c(100,200,300) + c(1,2,3,4,5)+c(10,20,30,40,50,60,70)
- You should get a warning, telling you that the vectors are not a multiple of each other's lengths, but you will also get some results.

- Why do you think that you get the results that you see? We are adding single units, tens, and hundreds, so it should be possible to see how each vector gets recycled.
- What is the recycling pattern for hundreds? Is this as you expected? If not, why do you think that the results are as you see.

R also provides a short hand operator for producing series of integer, the colon operator ':'. x:y generates a set of integers from x to y, counting up or down as appropriate.

```
> 1:5
[1] 1 2 3 4 5
```

This allows us to create long sequences, and thus to illustrate the way that the value in [] on output lines work:

```
> 1:40
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
[21] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

Here, it can be seen that we are being told that the first output line starts with element 1, and the second output line starts with element 21.

Logical vectors

The above vectors are of numeric data, but as we have seen we can have vectors of any type, including logical vectors.

Consider this example from Arnold and Tilton:

```
> numericVec <- 1:10
> logicalVec <- (numericVec >= 5)
> logicalVec
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> class(logicalVec)
[1] "logical"
> numericVec == 4
 [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

On the first line, we've created a sequence of numbers from 1 to 10, and assigned that to the vector numericVec.

On the second line, we aagain do an assignment. We should examine the right hand side of this line, and work out what it does. The result, whatever it is, gets assigned to logicalVec. So, what is happening? We have the expression (numericVec >= 5). This is a comparison: is numericVec greater than, or equal to, 5? Remember that numericVec contains 10 values. This comparison is therefore applied to each value in turn. This generates a vector of 10 results, which gets assigned to logicalVec. In or next line, we look at the content of logicalVec – 4 'FALSE' values (corresponding to the first 4 values of numericVec all of which have a value less than 5, followed by 6 'TRUE' values (because those elements of numericVec are greater than or equal to 5.

We then check the class of logicalVec.

Finally, we look at the expression numericVec == 4. Just as with the first case, we apply this to the whole of numericVec, and get a vector of 10 TRUE/FALSE results.

- Try generating a set of integers, and then use the modulus function (%%) to create a logical vector which is FALSE if the corresponding value of your input set is divisible by three, and TRUE otherwise.

Subsetting

R provides a number of ways in which we can create a subset of a vector, as we shall see.

The simplest way is to use index notation; this is similar to the handling of arrays in some other languages.

```
> vectorObj <- 11:20
> vectorObj[5]
[1] 15
```

Here, we create vectorObj as a sequence of integers from 11 to 20. Our second line asks for the value of element number 5, giving the answer '15'.

We can supply multiple values here:

```
> vectorObj[c(1,3,5)]
[1] 11 13 15
> vectorObj[7:10]
[1] 17 18 19 20
```

In the first example, we create a vector (using 'c') of the values 1, 3 and 5; R then returns the values of elements 1, 3 and 5 of vectorObj. Note that directly saying: vectorObj[1,3,5] does not work – '1, 3, 5' is not a vector (which is what we want as a parameter). The second example generates a vector of numbers from 7 to 10, and then returns the values of elements 7, 8, 9 and 10 of vectorObj.

We can also use index notation to assign values:

```
> vectorObj[2] <- -5
> vectorObj[2:5] <- -5
```

On the first line, we assign the value -5 to element number 2; on the second line we assign -5 to all elements in the range 2 to 5. Here we are recycling a vector of length one.

If we use negative values in our subsetting indexes, then R will return the original vector except the elements corresponding to those (negative) values.

For example, vectorObj[-1] returns all of vectorObj apart from the first value.

We can also use a logical vector as a mask to subset another vector. Where the logical vector contains a TRUE, then our subset will include the corresponding value.

```
> vectorObj <- 1:4

> vectorObj[c(TRUE,TRUE,FALSE,FALSE)]
```

```
[1] 1 2
```

Here, we start by creating vectorObj, which contains integers from 1 to 4. We then subset it by supplying some values inside square brackets []. The values provided are a logical vector. The first value in the logical vector is TRUE, so our subset includes the first value in vectorObj. The same is true for the second pair of values. The third value in our logical vector is false, and thus out subset does *not* include the third value in vectorObj, and so on.

A typical form of logical subsetting involves building a logical vector on the basis of a comparison against the original vector. For example:

```
> vectorObj <- 1:8
> logicalIndex <- vectorObj > 5
> logicalIndex
[1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
> newVectorObj <- vectorObj[logicalIndex]
> newVectorObj
[1] 6 7 8
```

Here, we start by creating vectorObj, containing values 1 to 8. On the second line, we assign to logicalIndex, the results of 'vectorObj > 5'. This test each value in vectorObj, returning TRUE if it is greater than 5. It will generate a vector of logical values the same length as vectorObj. We loko at the results of this on line 3.

On line 4, we assign to newVectorObj, the subset of vectorObj indicated by our logical vector logicalIndex. Where logicalndex contains TRUE, so we will include the correspondingly numbered element of vectorObj.

Note that we do not necessarily need to create the interim vectors to do subsetting in this way.

We can also use logical vectors in assignments. Again, this example comes from Arnold and Tilton:

```
> vectorObj <- 1:100
> vectorObj[vectorObj <= 25] <- 25
> vectorObj[vectorObj >= 75] <- 75
> vectorObj
 [1] 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
[21] 25 25 25 25 25 26 27 28 29 30 31 23 33 34 35 36 37 38 39 40
...
```

On the first line, we create a vector of numbers 1 to 100. On the second line, we assign the number '25' to a subset of vectorObj; the subset is defined by the logical vector 'vectorObj <= 25'. Thus, R will create a logical vector 100 elements long, which is TRUE where vectorObj <= 25 and FALSE otherwise, and use this to refer to a subset of vectorObj (we are using vectorObj to select elements of itself). The value 25 is assigned to those elements. The third line does a similar operation assigning '75' to elements with a value of 75 or more. The first two lines of output are shown; we can see that elements 1 to 25 have been assigned the value 25, and the remaining elements (until we get to number 75) remain as they are.

Character vectors

So far we have looked at numeric and logical vectors. R also supports character vectors – vectors of text elements.

These are constructed in a similar manner to other vectors, for example using 'c()'

```
> stringVec <- c("pear","apple","pineapple")
> class(stringVec)
[1] "character"
> stringVec
[1] "pear"  "apple"  "pineapple"
```

Whereas some languages use the '+' operator for joining strings (as well as being an addition operator for numbers), R uses paste() to join strings.

```
> paste(stringVec,"juice",sep=" ")
[1] "pear juice"  "apple juice"  "pineapple juice"
```

Here we are pasting the string "juice" to the elements in stringVec. As with numeric examples, "juice" is a single value vector, and gets recycled. If the second argument here was a vector with more than one value, we would iterate through the values (and recycle the shorter vector). The 'sep' argument tells R what character to use to separate the pasted elements, in this case a space. The separator can be empty: sep="".

Note that we can alternatively (or additionally) use the argument *collapse*. If collapse is included, w will join strings together into a single output value, separated by the collapse character,

- Try paste(stringVec,"juice",sep="",collapse=",")

Additional string functions include substr() and nchar(); substr() extracts a substring, whilst nchar() gives the number of characters in the string.

```
> substr(stringVec,start=2,end=4)
[1] "ear"  "ppl"  "ine"
> substr(stringVec,start=3,end=nchar(stringVec)
[1] "ar"  "ple"  "neapple"
```

The first of this pair of examples extracts a substring starting at position 2 (with numbering starting at 1), and ending at position 4. The second example starts at position 3, and ends at position nchar(stringVec) – i.e., the length of each string (for 'apple' nchar() returns the value 5, for pear 4, and so on. If the end value is longer than the string, then it will be capped at the string length.

Finally, the function grep can be used to return the index values of elements of a string vector that match a pattern.

```
> index <- grep(pattern="pp",x=stringVec)
> index
[1] 2 3
```

Here, we are using grep to look for elements of stringVec (we have given this as an argument to grep) that match the pattern "pp" – i.e. that that pattern is found somewhere in the string.

The produces a vector of index values, and tells us that elements number 2 and 3 match that pattern. Selecting stringVec[index] would then produce the subset from stringVec that match the pattern (‘apple’ and ‘pineapple’).

We can use *regular expression* patterns in grep, although will not go into detail here. For example, the pattern “^p” matches the character ‘p’ at the beginning of a string.

```
> index <- grep(pattern="^p",x=stringVec)
> index
[1] 1 3
> stringVec[index]
[1] "pear"  "pineapple"
```

Exercises

These exercises are based on some given in Arnold and Tilton, however they also introduce a number of extra functions not described in the corresponding chapter. Some of these are mentioned here:

1) Choose a number *n*; generate the set of integers from 1 to n
2) Choose a number *n*; generate the set of integers from n to 1
3) Choose a number n; generate the subset of numbers between 1 and n that are even
4) Create a set of numbers from 1800 to 2020
    a. Generate the subset of numbers that are even
    b. Generate the subset of numbers that are divisible by 4 (leaving remainder 0)
    c. Generate the subset of numbers that correspond to leap years, using the rules that a year is a leap year if:
        i. It is divisible by 4
        ii. AND it is not divisible by 100
        iii. UNLESS it is also divisible by 400

The functions sum() and mean() respectively return the sum and arithmetic mean of a numeric vector.

1) Choose two numbers n and m; generate the set of integers from *n* to *m*
2) Find the mean of that set of numbers
3) Find the sum of that set of numbers
4) Choose a number *n*; calculate the sum of numbers 1 + 1/2 + 1/3 + ... +1/n

The function seq(start,end,increment) generates a sequence of numbers beginning with *start*, ending on (or not greater than) *end*, and incrementing by *increment*. For example seq(1,5,1) gives the sequence 1,2,3,4,5; seq(1,10,2) gives the sequence 1, 3, 5, 7, 9 (adding 2 each time, and not greater than 10).

The following template creates a function called ask(). It uses the builtin function readline() to ask the user to enter a value, and then the function *as.numeric* to convert that to a numeric.

```
> ask <- function() {
+ z <- readline("Enter a number: ")
+ z <- as.numeric(z)
+ num <- z
+ return(num)
+ }
```

The first line starts to define a function; note that we have an opening bracket '{'; RStudio may well try to helpfully add the closing bracket when you type this, but for the time being we do not want that.

Note that '+' as the prompt indicates that we are continuting the expression from the line above.

The second line prompts the user "Enter a number", and waits for input. That input is assigned to *z*.

The third line converts z to a numeric value.

The fourth line does a simple assignment; it assigns the value of z to num.

The fifth line returns the value of 'num'.

Finally, we close the curly bracket '}'.

We can now run this function by typing "ask()" at the command prompt. Note that creating the function by typing it in like this is unwieldy and error prone. We will only do this briefly. You will probably find it easy to write this out in a text editor, and paste it into the R console. IF doing this, you should not include the various prompt marks.

The key line is the fourth one, 'num <- z'. We can assign anything to num, using z as our input. For example, to create a function that squared the input number, we could say 'num <- z^2'. In a text editor, your function would look like this:

```
ask <- function() {
z <- readline("Enter a number: ")
z <- as.numeric(z)
num <- z^2
return(num)
}
```

We might want to use several lines generating vectors etc, before returning a result.

1) Convert your above answers so that they ask the user for an input number (in some cases, you may want more than one), and then generate the relevant results.