Starting Out in R

Introduction

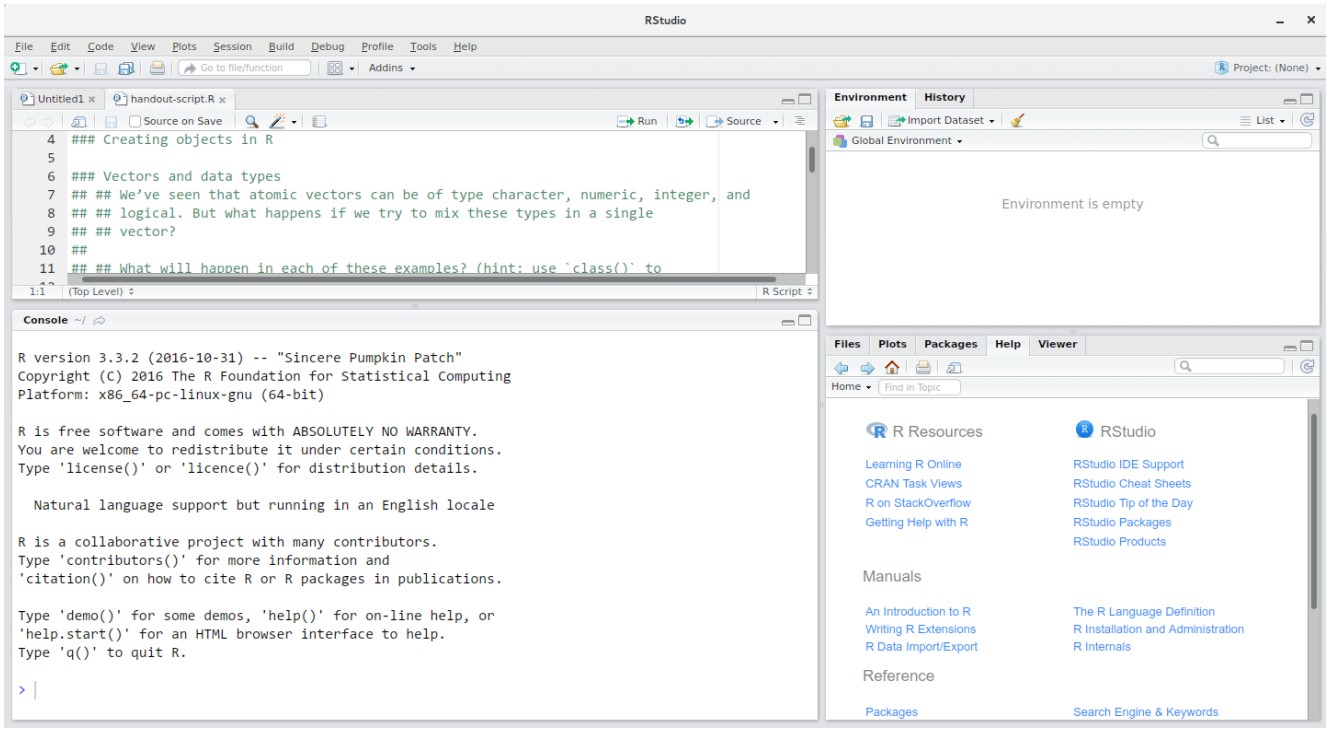
R is a coding language focussed on the interactive analysis of data. It was written as an open-source alternative to the *S* language and along with Python is the most popular language for applied data science.

RStudio and the R console

To work and run *R* code we will use RStudio. This application is an IDE (Integrated Development Environment) that allows us to organise projects files, edit code, to run code, view results and even to write up our work.

Note: RStudio has other useful functionality for more advanced development work for example e.g., version control, developing R packages, writing interactive data dashboards and web applications.

Typically when we work in RStudio we might see the window divided into four panes:



RStudio interface screenshot. Clockwise from top left: Source, Environment/History, Files/Plots/Packages/Help/Viewer, Console.

pane	default location
the **source pane** for your scripts and documents	top left
the **Environment/History** pane	top right
the **Files/Plots/Packages/Help/Viewer** pane	bottom right
the **R-Console** pane	bottom left

The placement of these panes and their content can be customized (see menu, Tools -> Global Options -> Pane Layout).

One of the advantages of using RStudio is that all the information you need to write code is available in a single window. It also has useful data and editing tools, keyboard shortcuts, autocompletion, and code debugging tools.

For now we will concentrate only in the R Console panel, which allows us to type in and run commands.

```
R version 3.5.2 (2018-12-20) -- "Eggshell Igloo"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

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

Maths in R

We can use R as a scientific calculator using commands like:

```
+ addition
- subtraction
/ division
* mutliplication
```

Try entering the following commands into the RStudio console:

```
1 + 2
```

```
999999 / 810000
```

Some other useful R commands when carrying out calculations are:

`**` or `^`	raise to the power	`10**3` and `10^3` give `1000`
`sqrt`	square root	`sqrt(16)` gives `4`
`exp`	e^x	`exp(2)` gives `7.389056`
`log10`	logarithm base 10	`log10(1000)` gives `3`
`log`	natural logarithm	`log(7.389056)` gives `2`
`abs`	absolute value	`abs(-2.5)` gives `2.5`

R uses `e` -notation for entering and displaying very large or small numbers in scientific format:

`1e6`	means 1,000,000	1×10^6
`2.5e-3`	means 0.0025	2.5×10^{-3}
`-4e-4`	means -0.0004	-4×10^{-4}

```
> 1000*1000
> 2.5/100000
```

Calculations are completed according to the priority defined by the *PEDMAS* system:

Order of priority	**e.g.**
Parentheses or brackets	`(4 + 1) * 3 = 15`

Order of priority	e.g.
Exponentiation	`3^2 = 9`
Division:	`3 / 4 = 0.75`
Multiplication:	`7 * 7 = 49`
Addition:	`3 + 3 = 6`
Subtraction:	`5 - 2 = 3`

Exercises A.

1. Calculate the total length in km if all the double decker buses in London were parked end to end. (6785 buses of length 10.5m).
2. Calculate the average distance between stops on the Victoria line. (16 stations, total line length 21km)
3. Calculate the cost of getting a black cab 6.1 km from Brick Lane to UCL (cost is £3.00 for first 162m then 20p for every 81m)
4. Estimate the number of biscuits in a pack of *McVities Rich Tea* .(length of pack 23.5cm, length of biscuit 0.6cm)

Variable assignment in R

To help us work with information we store it as a variable. In most programming languages we assign data using the equals sign `=` , this can also be used in R, but the convention is to use an arrow `<-` for assignment. Type the following in the console:

```
a <- 3
b <- 5
a + b
```

Other allowed ways to assign data to variables in R:

```
c = 4
5 -> d
e <- c + d
e
```

Note that R displays the output of any command, except when we store the output into a variable then R.

Variable naming schemes

Variable names should be chosen to be concise but informative. They must start with a letter (not a number) and only contain letters, numbers and the dot `.` or underscore `_` character.

Variable names cannot use spaces so their are a few naming conventions for getting around this:

format	example
`snake_case_uses_underscores`	`population_by_country`
`camelCaseUsesCapitals`	`populationByCountry`
`another.style.using.periods`	`population.by.country`

You are free to choose which you prefer but the important thing is to try to work consistently within your code files.

Note: take care to avoid variable names that might overwrite inbuilt commands, e.g. `plot`, `mean`, `max`. A simple way to do this is to add a my prefix e.g. `my_plot`. When typing

Types of variables

We need to deal with many different types of data (e.g. numerical, text, category), and also we need to deal with collections of data (e.g. lists, tables).

R contains object classes that let us store and work with these different types of data. Below are some of the common object classes we will work with:

Classes in R: types of data

name	detail	example
integer	whole numbers	`1`, `-999`, `6752`
numeric	floating point numbers	`1.00`, `-9.99`, `0.6752`
logical	boolean values	`TRUE`, `FALSE`

name	detail	example
character	text strings	`"some text"`, `"R is great!"`
factor	associates to category labels	`"healthy"`, `"diseased"`

To check the class of a variable we can use the `class` function:

```
> a <- 'a text string'
> class(a)
[1] "character"
```

Note that we can use single `'` or double `"` quote marks to define the start and end of the string e.g.
character string. Either can be used to .

```
first_name <- "harry"
last_name <- 'potter'
# start and end quote types must match!
```

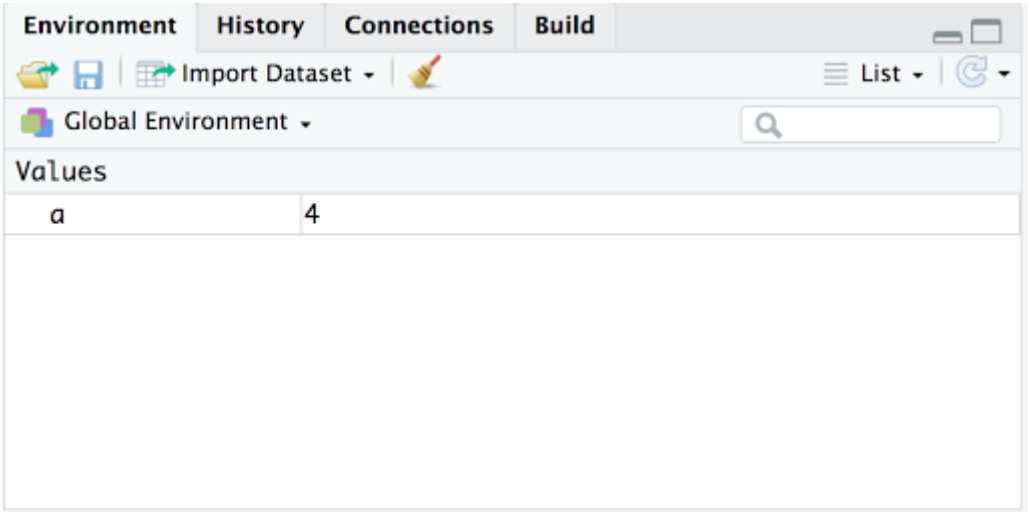
Classes in R: collections of data

name	detail
vector	series of data items (must be same type)
list	series of data items (can be of different types)
dataframe	object storing a data table (columns can be different types)

Ways to investigate variables

We can look at the contents of a variable using the `print` command. You may also see code examples that use alternative commands: `cat` (con*cat*entate and print) or `str` (display *str*ucture).

When we create variables we say they are added to our Environment. We can see the variables defined in our environment using the `ls` command or to look in the Environment tab in top right RStudio panel:



We can remove a variable from the environment using the `rm` command:

```
a <- 4
ls()
rm(a)
ls()
```

To delete all variables from our environment we can use:

```
rm(list = ls())
```

RStudio will help autocomplete your commands and variable names as you type e.g. enter the following code:

```
very_long_variable_name <- 999
```

One the next line if you type `very` you should see the autocompleted options pop up, and you can select an item using the tab key. Similarly if you type `sqr` you should see the autocomplete suggest the command `sqr_root`.

Vectors

The simplest way to store a set of data in R is to store it in a vector. The `c` or concatenate/combine command can be used to define a vector.

```
my_vector <- c("a", "b", "c", "d", "e")
print(my_vector)
class(my_vector)
str(my_vector)
```

Note: if you forget to use `c` when defining a function you will see an error like:

```
> my_vector <- ("a", "b", "c", "d", "e")
Error: unexpected ',' in "my_vector <- ("a","Error: unexpected ',' in "my_vector
```

This can be a fairly common typo (other languages don't use this format), so try to make note of the above message and how to fix it!

There are some useful functions that operate on a vector:

```
sum , length , min , max , range
mean , median , var , sd
quantile , sort
```

Exercises B.
A seller sells iphones on ebay and wants to explore their sales data (cost of second hand *IPhone XS 64GB* in GBP £:

```
iphone_sales <- c(621, 455, 650, 600, 625, 646, 640, 750, 480, 629, 690, 600)
```

Use the above commands to answer the following questions:

1. What is the total money that has been raised?
   Hint: `sum(iphone_sales)`
2. How many iphones sales are in the data?
3. What was the mean price paid?
4. What was the median price paid?
5. What was the maximum price paid?
6. What was the minimum price paid?

The following data records the birth weight in kg of 10 babies (check resources for full data set).

```
birth_weight <- c(2.1, 1.57, 1.99, ... )
```

Use R commands to answer the following questions:

7. What was the mean birth weight?
8. What was the variance and standard deviation in the birth weight distribution?
9. Check that the values for the last question are in agreement. (Hint: recall the relationship between standard deviation and variance).
10. What was the range of birth weights?
11. What weight range accounts for 95% of all births (so that 2.5% are lower and 2.5% are higher)

12. How can we obtain a sorted set of the data?

Operations on vectors

We can also apply mathematical operators to vectors:

Create the following vectors in R:

```
a <- c(1,2,3,4,5)
b <- c(6,7,8,9,10)
```

Try out the following commands to understand how R handles calculations with vectors:

```
a + 2
a**2
log(a)
c <- a+b
c
d <- a**b
d
```

If we ask R to perform a calculation with vectors of different lengths R will **recycle** the numbers from the shorter vector to carry out the calculation e.g.

```
> c(1,2,3,4,5,6) + c(10,0)
[1] 11  2 13  4 15  6
```

This can be useful, but can also hide issues (e.g. R won't necessarily tell us when the vectors in a calculation have different lengths).

Generating vectors

The `rep` command lets us generate a vector by replicating a single value (or set of values):

```
# vector filled with 0 repeated five times
a <- rep(0,5)
# vector filled with 0 1 repeated three times
b <- rep(c(0,1),3)
```

The `seq` function lets us generate a sequence in R. To use it you must include a start and end value:

```
> seq(1,10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Here 1 and 10 are the `seq` function's arguments. In general to find out what arguments a function uses we look at the function documentation using the `help` command.

```
help(seq)
# shortcut
?seq
```

When a function can take multiple/optional arguments we specify the argument names in our function calls:

```
seq(from = 1, to = 5)
```

Look at the documentation for `seq` and use this to generate sequences:

1. sequence from 0 to 100 in steps of 10
2. sequence from 0 to 10 in steps of 2.5
3. sequence running from 1 to 2 that has length 11 items

As a shortcut R can also use the colon `:` to specify a number sequence:

```
1:10
```

Generating random sets

It can also be useful to generate vectors filled with random numbers:

```
# random number drawn from uniform distribution from interval 0 to 1
a <- runif(10)
```

```
# random number drawn from normal distribution with mean 0 and standard deviatio
b <- rnorm(20)
```

To generate a set of random integers we can sample values from a generated sequence:

```
lottery_numbers = 1:49
winning_set = sample(lottery_numbers, size=6, replace=FALSE)
```

Note computers generate pseudo-random numbers according to an algorithm. The starting point for the algorithm is called the random seed. If the seed is unspecified this is set to an

arbitrary number (e.g. last random value generated or the current time) but for reproducibility it can be useful to specify a starting seed:

```
set.seed(100)
runif(1)
```

```
[1] 0.3077661
```

```
set.seed(100)
runif(1)
```

```
[1] 0.3077661
```

You can choose any number to be the seed. What is important is that someone using the same seed can reproduce your work. As an interesting story the fact that numbers generated by computers are only pseudorandom has been used to "win the lottery"

Loops

Loops allow us to cycle through a set of instructions, which can also be a useful tool to generate data:

```
output_vals <- rep(0,10)
output_vals[1] <- 1
for(i in 2:10) (
  output_vals[i] <- output_vals[i-1]*2
)
output_vals
```

Note that in R, it is common to create the object to store the output of a *for* loop before the loop is started (this helps ensure code runs efficiently).

Exercises C.
Write R code to generate the following vectors:

1. Your name e.g.

```
print(my_name)
[1] "Philip" "William" "Lewis"
```

2. The numbers 1 to 6
3. Numbers 0 to 2 in steps of 0.2

4. Five numbers drawn at random between between 0 and 10.
5. Four items drawn from the vector:

```
coin_flip <- c("Heads", "Tails")
```

6. Three numbers from a normal distribution with mean = 100, and standard deviation = 30, with initial seed set to `1`.

Retrieving data from vectors

We can access an item in a vector using the item index (first item has index 1, second index 2 etc):

single index method

```
my_vector <- c("a", "b", "c", "d", "e")
my_vector[1]
```

multiple index method

Or use a vector of indices to select multiple items:

```
my_vector <- c("a", "b", "c", "d", "e")
my_vector[c(1,3,5)]
```

index slice method

Or using `:` to make a sequence of indices to select:

```
my_vector[1:3]
```

negative index method

If we want all items *except* certain indicex, we can specify the selction using the `-` operator:

```
my_vector[-1]
```

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

```
my_vector[-c(2,4,5)]
```

```
[1]  "a", "c"
```

Exercises D.

`letters` is an inbuilt vector that contains the letters of the alphabet.

```
print(letters)
[1] "a" "b" "c" "d" ...
```

Write R code to make the following selections:

1. Letter 18 in the alphabet.
2. The first three letters in the alphabet.
3. Just letters in positions 1, 5, 9, 15, 21.
4. All letters except those in positions 1, 5, 9, 15, 21.
5. The last 3 letters in the alphabet.
6. Can you:
   a. Repeat part 5. so that it would also work to select the last 3 items of a differently length list?
   b. Make a selection that translates to:
   `"h", "e", "l", "l", "o", "w", "o", "r", "l", "d"`

Hint. The command `View(letters)` will open up an indexed view of the letters vector so you can select the right ones easily.

Logical data, tests and operations

R lets us make comparison tests:

symbol	description	example: `TRUE`
`<`	less than	`1 < 2`
`<=`	less than or equal to	`1 <= 1`
`>`	greater than	`3 > 1`
`>=`	greater than or equal to	`4 >= 4`
`==`	is equal to	`2 == 2`
`!=`	is not equal to	`3 != 4`

Note: Be careful not to mix up `=` and `==` :

`=`	used to set a variable
`==`	used as a test for equality.

Here are some examples of usage:

```
a <- 4
b <- 100
c <- "harry"
```

Check if `a` is greater than `1` :

```
a > 1
```

Check if `a` is less or equal to `4`

```
a <= 4
```

Check if `c` is equal to `"ron"` :

```
c == "harry"
```

Check if `b` is not equal to `1000`

```
b != 1000
```

We can combine tests using `&` `|` and `!` :

AND `&`
means `TRUE` if *both* conditions are met
so `FALSE` if *either* of the conditions are not met

```
age <- 12
age > 5  & age < 18
[1] TRUE
```

OR `|`
means `TRUE` if *either* of the conditions is met
so `FALSE` if *both* conditions are not met

```
age <- 35
age < 18 | age > 65
[1] TRUE
```

NOT `!`
allows us to invert a condition, so will produce `TRUE` if the condition is *not* met.

```
risk_level <- "medium"
! risk_level=="low"
[1] TRUE
```

Brackets `(...)`
Just as brackets can be used to group mathematical expressions, they can also be used to group more complex logical tests:

```
(age < 18 | age >  65) & risk_level == 'high'
```

Working with logical data

We can store logical values into variables:

```
passed_exam <- TRUE
```

```
species = c("cat","dog","mouse")
goes_woof = c(FALSE, TRUE, FALSE)
likes_cheese = c(F, F, T)
```

Note how R allows us to use `T` and `F` as shorthand for `TRUE` and `FALSE`.

We can apply tests to whole vectors:

```
my_numbers <- seq(1,9)
my_numbers >= 8
```

Note that internally the values `TRUE` and `FALSE` correspond to numerical values `0` and `1`:

```
> TRUE + TRUE
[1] 2
```

This means we can also find the total number of `TRUE` entries in a vector by applying the sum function:

```
> item_in_stock <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
> length(item_in_stock)
[1] 5
> sum(item_in_stock)
[1] 3
```

Selection of data using a logical test

We can the result of a logical test to ***select*** or ***filter*** items from vectors:

```
item_name <- c("apple","banana","orange","pear","peach")
item_stock <- c(122, 0, 22, 0, 1)
item_stock == 0
```

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

```
item_name[item_stock == 0]
```

```
[1] "banana" "pear"
```

We can also filter based directly on a vector of `TRUE` or `FALSE` values:

```
> on_order <- c(F, T, T, T, T)
> item_name[on_order]
[1] "banana","orange","pear","peach"
```

`which` function

It can also be useful to identify which index satisfies a condition. For example:

```
students <- c("ann", "bob", "carl", "david", "elle")
results <-  c( 93,    65,    45,     85,      67)
which(results > 70)
```

```
[1] 1 4
```

We can use the resulting set of indices to select items e.g.

```
students[which(results > 70)]
```

```
[1] "ann"   "david"
```

Exercises E.

Copy and paste the R code to set up the students and grades data:

```
students <- c( "Lauren",  "Marcelino", ..
grades <- c(78, 83, 60, ...
```

Using the R commands we have been using answer the following:

1. What was the mean grade?
2. What was the maximum grade?
3. How many students scored over 70 or higher?
4. What were the scores that were below 60?
5. What were the names of the students that were below 60? Hint: Use the `which` command.
6. Create the lists `good_grades` and `good_students` containing the data of the students who scored more than the average grade.

Data for all exercises:

```
iphone_sales <- c(621, 455, 650, 600, 625, 646, 640, 750, 480, 629, 690, 600)
```

```
birth_weights <- c(2.57, 3.25, 2.51, 3.04, 2.12, 2.54, 2.39, 2.55, 3.2, 2.66, 2.
```

```
students <- c( "Lauren",  "Marcelino",  "Ramona",  "Lonna", "Justine",  "Sanjuan
```

```
grades <- c(78, 83, 60, 65, 67, 65, 56, 68, 86, 70, 55, 74, 73, 55, 83, 63, 60,
```