Practical Session 1

David Barron

Hilary Term 2018

Introduction to R

This is a brief introduction to basic R use. The document is also available at http://users.ox.ac.uk/~jesu0073/.

Arithmetic operators

All the basic arithmetic operators work as you would expect:

```
2 + 3
```

```
[1] 5
```

```
5 - 3
```

```
[1] 2
```

```
4 * 9
```

```
[1] 36
```

```
8/2
```

```
[1] 4
```

```
2^3
```

```
[1] 8
```

```
7%%3
```

```
[1] 1
```

```
7%/%3
```

```
[1] 2
```

The last two may be a little less familiar: they are called the *modulus* and *integer division*, respectively.

Assignment to variables

There are two assignment operators, `<-` and `=`. The advantage of the latter is that it only takes one key stroke (and it may be natural to you if you use other programming languages). The advantage of the former is that it makes your code easier to read, as `=` is used for other purposes as well. Most R coders recommend the use of `<-`.

```
x <- 4
x
```

```
[1] 4
```

```
y <- 4 * 5
y
```

```
[1] 20
```

```r
x + y
```

```
[1] 24
```

Data types

The basic data type is the *vector*. A single number is just a vector of length 1. Vectors can store numbers (either integers or decimals), character strings, or `TRUE`/`FALSE` values. The simplest way to create a vector is using the concatenation operator, `c`:

```r
a <- c(2, 4, 6)
mode(a)
```

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

```r
b <- c("two", "four", "six")
mode(b)
```

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

```r
c <- c(TRUE, FALSE, TRUE)
mode(c)
```

```
[1] "logical"
```

```r
c(a, b)
```

```
[1] "2"    "4"    "6"    "two"  "four" "six"
```

```r
c(a, c)
```

```
[1] 2 4 6 1 0 1
```

You will notice that the last two cases involve *coercion*. In a vector, all the values have to be of the same mode, so if you attempt to join to vectors with different modes, one is coerced to the type of the other. The rule is that the most "general" mode wins. For example, you can represent numbers as characters, and logical values as 0 or 1.

Matrices, lists and data frames

There are four other classes of data type that you are likely to come across. This is a summary:

Class	Characteristics	Dimensions
Vector	All components have the same mode	1
Matrix	A set of vectors of the same mode and length	2
Data frame	A set of vectors of the same length	2
List	A set of objects of any mode or length	

Examples:

```r
print(m <- cbind(a, b, c))  # Matrix
```

```
     a   b      c
[1,] "2" "two"  "TRUE"
[2,] "4" "four" "FALSE"
[3,] "6" "six"  "TRUE"
```

```r
print(d <- data.frame(a, b, c))  # Data frame
```

```
  a    b     c
1 2  two  TRUE
2 4 four FALSE
3 6  six  TRUE
```

```r
print(l <- list(a, b, c))  # List
```

```
[[1]]
[1] 2 4 6

[[2]]
[1] "two"  "four" "six"

[[3]]
[1]  TRUE FALSE  TRUE
```

To begin with you are most likely to use data frames (as these are passed to the functions used to actually do statistics!). Lists are often used by these functions to store output if you need to do more than just look at the printed output. (A data frame might look like a sort of matrix, but actually it is a sort of list.)

Notice that the data frame has added names at the top of the column (by default the names of the vectors, but this can be over-ridden as we'll see in a minute). It is natural to think of the columns as variables (and hence these are *variable names*) and the rows as observations.

Selecting components of data

Unlike some other statistical analysis systems, you work with multiple data sets at once. That saves a lot of opening and closing of files, but it does make accessing components of data more difficult as you have to tell R not just the same of a variable, but also the data frame in which that variable is located. There are a number of ways of doing this:

1. Indexing You can specify the content of any component of a data frame (or matrix or vector) by using the row and/or column index in square brackets:

```r
d[1, 1]
```

```
[1] 2
```

```r
d[, 1]
```

```
[1] 2 4 6
```

```r
d[3, ]
```

```
  a   b    c
3 6 six TRUE
```

```r
d[1:2, ]
```

```
  a    b     c
1 2  two  TRUE
2 4 four FALSE
```

```r
d[, c(1, 3)]
```

```
  a     c
1 2  TRUE
```

```
2 4 FALSE
3 6  TRUE
```

```
a[3]
```

```
[1] 6
```

Notice that you can specify all the cases in one dimension simply by leaving it blank. So the second example above shows all the rows for one variable while the second shows all the columns (variables) for one observation. You can select multiple rows and/or columns by using the `:` operator, which is a short cut to specify a sequence of integers (e.g., 1, 2, 3) or a vector, as in the fifth example above. Indexing a vector is done in the same way, but there is no comma as there is only one dimension.

2. Naming You can select columns (variables) by name in two ways:

```
d[, "a"]
```

```
[1] 2 4 6
```

```
d$a
```

```
[1] 2 4 6
```

```
d[, c("b", "c")]
```

```
     b     c
1  two  TRUE
2 four FALSE
3  six  TRUE
```

Using the `$` operator is more convenient for selecting a single variable; the other method is more flexible as you can select more than one variable at a time.

3. Using the `with` or `transform` functions

```
a.sq <- with(d, a^2)
d  # Doesn't change d, can add it explicitly
```

```
  a    b     c
1 2  two  TRUE
2 4 four FALSE
3 6  six  TRUE
```

```
d$as.sq <- a.sq

d <- transform(d, a.squared = a^2)
d
```

```
  a    b     c as.sq a.squared
1 2  two  TRUE     4         4
2 4 four FALSE    16        16
3 6  six  TRUE    36        36
```

Notice that, although `tranform` does add a new variable, you have to explicitly assign this to a name to store it permanently. You can either overwrite the existing data frame or create a new one.

4. Use the `attach` function This makes the variable names temporarily available as if they were separate vectors.

```
attach(d)
```

```
The following objects are masked _by_ .GlobalEnv:
```

```
a, b, c
```

```r
log(a)
```

```
[1] 0.6931472 1.3862944 1.7917595
```

```r
detach(d)
```

The advantage of the `attach` function is you can then work with variable names in a way that might seem natural to you, but it can be risky if there is more than one object with the same names. It is also easy to forget to detach it. I generally avoid this method in favour of either 2 or 3.

Functions

We've already seen some functions (e.g., `log`, `transform`). Functions are what do the hard work in R, particularly the ones that have been written to do statistical analyses like regression. The function is always followed by ( ), inside which are given the *arguments*: the pieces of data or information R needs for the function to work. Here are some useful functions for taking an initial look at some data.

```r
head(mtcars)  # The first 6 rows of data
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
tail(mtcars, 10)  # The last 10 rows of data
```

```
                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
AMC Javelin     15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28      13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9       27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2   26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa    30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L  15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino    19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora   15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E      21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```r
summary(mtcars)
```

```
      mpg             cyl             disp             hp       
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
 Median :19.20   Median :6.000   Median :196.3   Median :123.0  
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
      drat             wt             qsec             vs        
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
```

```
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
       am              gear            carb
 Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

And here are some useful functions for getting descriptive statistics:

```r
mean(mtcars$mpg)
```

```
[1] 20.09062
```

```r
sd(mtcars$mpg)
```

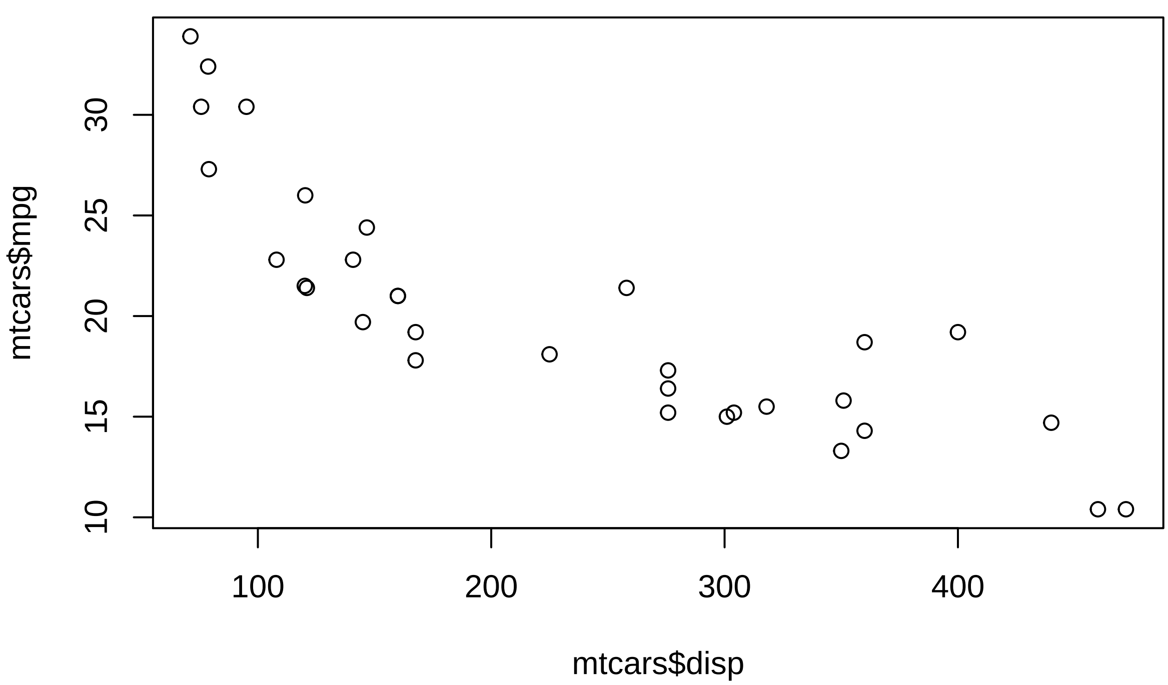
```
[1] 6.026948
```

```r
table(mtcars$cyl)
```

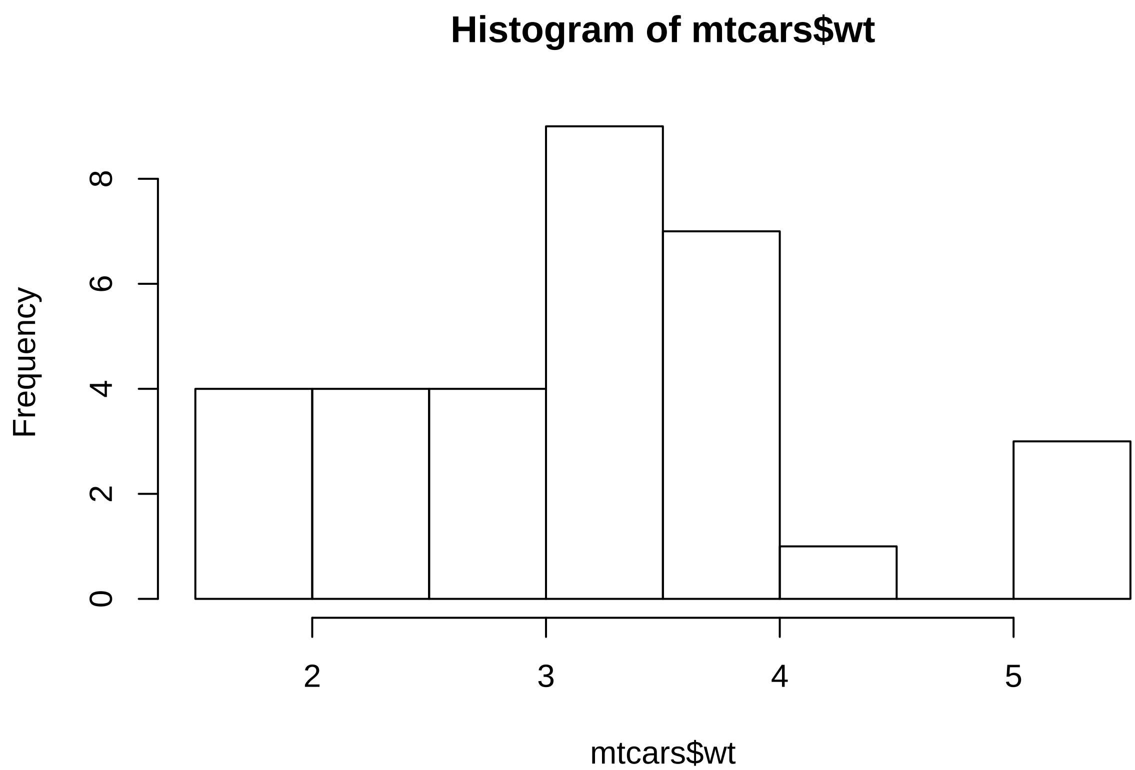
```

 4  6  8
11  7 14
```

R has very powerful graphical capabilities. Here are some simple examples:

```r
plot(mtcars$disp, mtcars$mpg)  # scatter plot
```



```r
hist(mtcars$wt)  # histogram
```



Missing values

It is very common for data to have some observations that are missing for some reason. In surveys, some people might refuse to answer certain questions or just not know the answers, for example. Most statistical packages have a special code for such cases, and in R that code is `NA` (which I guess stands for "not available"). Be careful, though. Often when you obtain data from archives it uses special numbers to indicate missing because it doesn't want to rely on something specific to one package.

When data contains missing values, the default behaviour is for R to fail when you try to do a statistical analysis. This is a good thing as you want to be warned that there are missing cases. However, usually you will want to override this default, and you can do this by adding another argument to the function call:

To illustrate this, I'll turn the first value in the `mtcars` data frame into `NA`:

```r
mtcars.tmp <- mtcars
mtcars.tmp[1, 1] <- NA
mtcars.tmp[, 1]
```

```
 [1]   NA 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
[15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
[29] 15.8 19.7 15.0 21.4
```

```r
mean(mtcars.tmp$mpg)  # Fails
```

```
[1] NA
```

```
mean(mtcars.tmp$mpg, na.rm = TRUE)  # Works!
```

```
[1] 20.06129
```

```
rm(mtcars.tmp)
```

The option here is `na.rm` ("NA remove"), which I set to `TRUE`. This then removes cases that are `NA` before calculating the mean. To know what options are available for a function, you can look at its help page by using the command `?mean` or `help('mean')`. Accessing the built-in help is essential; there's far too much to remember!

Packages

Much of the power of R comes from packages written to perform numerous tasks. R ships with a number of built in packages, others can be installed from within R. Once installed, they can easily be accessed. Below I give an example of a package that enables files in various data formats to be uploaded into R.

Data import/export

We need to be able to read data stored on a computer into R, and also to save our work to disk. It is straightforward to read data in a plain text format, including the very common .csv format. We can also use the `foreign` package to read in data stored in a variety of data formats, such as STATA.

```
## Load a .csv file
peru <- read.csv("C:\\Users\\dbarron\\Dropbox\\Advanced Quant\\PeruMicroWeek1.csv")

# This installs the package from a website. If you haven't specified one,
# you will be prompted to choose a repository. Any one will do.
# install.packages('haven', repos='http://cran.rstudio.com') Then make the
# package available
library(haven)

lfs <- read_dta("C:\\Users\\dbarron\\Dropbox\\Teaching\\MSc teaching\\Advanced Quant\\data\\lfs2002.dta

write.csv(lfs, "C:\\Users\\dbarron\\Dropbox\\Teaching\\MSc teaching\\Advanced Quant\\data\\lfs2002.csv"
```

In these examples, notice the use of `\\` in the strings that specify the location of the files to read. They are required (or alternatively you can use `/`) because `\` has a special meaning inside a string.

Linear regression

The function for carrying out linear regression is `lm` (for "linear model"). Have a look at the help first: `?lm`. You will see that the first argument that is required is a `formula`. This is the standard way of passing information about a regression model to R functions. Most often you specify it as `outcome variable ~ explanatory variable 1 + explanatory variable 2 + ...`. You can refer to the variable names in the data frame because, as you can see, the next argument you have to pass is the name of the data frame. Here is an example:

```
lm1 <- lm(mpg ~ cyl + wt, data = mtcars)
summary(lm1)
```

```
Call:
lm(formula = mpg ~ cyl + wt, data = mtcars)
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-4.2893 -1.5512 -0.4684  1.5743  6.1004

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  39.6863     1.7150  23.141  < 2e-16 ***
cyl          -1.5078     0.4147  -3.636 0.001064 **
wt           -3.1910     0.7569  -4.216 0.000222 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.568 on 29 degrees of freedom
Multiple R-squared:  0.8302,    Adjusted R-squared:  0.8185
F-statistic: 70.91 on 2 and 29 DF,  p-value: 6.809e-12
```

Homework

Data are in the `data` directory on Weblearn: https://weblearn.ox.ac.uk/x/MbYn1T.

1. Import the dataset PeruMicroWeek1.csv. Review the dataset by looking at its structure, header, and summary.
2. Calculate the means and standard deviations of Average Loan Balance (avgloanbal) and Percentage of Female Borrowers (femaleperc) in the dataset.
3. Plot a histogram of Average Loan Balance.
4. Plot Average Loan Balance vs Self Sufficiency Ratio (selfsuff). Note: The Self Sufficiency Ratio measures how able a microfinance organization is to financially sustain its operations.
5. Run an OLS regression. Dependent variable: Self Sufficiency Ratio. Explanatory variables: Average Loan Balance and Percentage of Female Borrowers. View a summary of the regression results. What is the effect of the explanatory variables on an organization's Self Sufficiency Ratio?