An Introduction to R and RStudio[1]

Mark Andrews
Psychology Department, Nottingham Trent University

[1] These slides are not intended to be self-contained and comprehensive, but just aim to provide some of the workshop's content. Much more will be provided in the workshop itself.

What is R and why should you care

- R is a program for doing statistics and data analysis.
- R's advantages or selling points relative to other programs (e.g, SPSS, SAS, Stata, Minitab, Python, Matlab, Maple, Mathematica, Tableau, Excel, SQL, and many others) come down to three inter-related factors:
  - It is immensely powerful.
  - It is open-source.
  - It is very and increasingly widely used.

R: A power tool for data analysis

The range and depth of statistical analyses and general data analyses that can be accomplished with R is immense.

- Built into R are virtually the entire repertoire of widely known and used statistical methods.
- Also built in to R is an extensive graphics library.
- R has a vast set of add-on or contributed packages. There are presently 16466 additional contributed packages.
- R is a programming language that is specialized to efficiently manipulate and perform calculations on data.
- The R programming language itself can be extended by interfacing with other programming languages like C, C++, Fortran, Python, and high performance computing or big data tools like Hadoop, Spark, SQL.

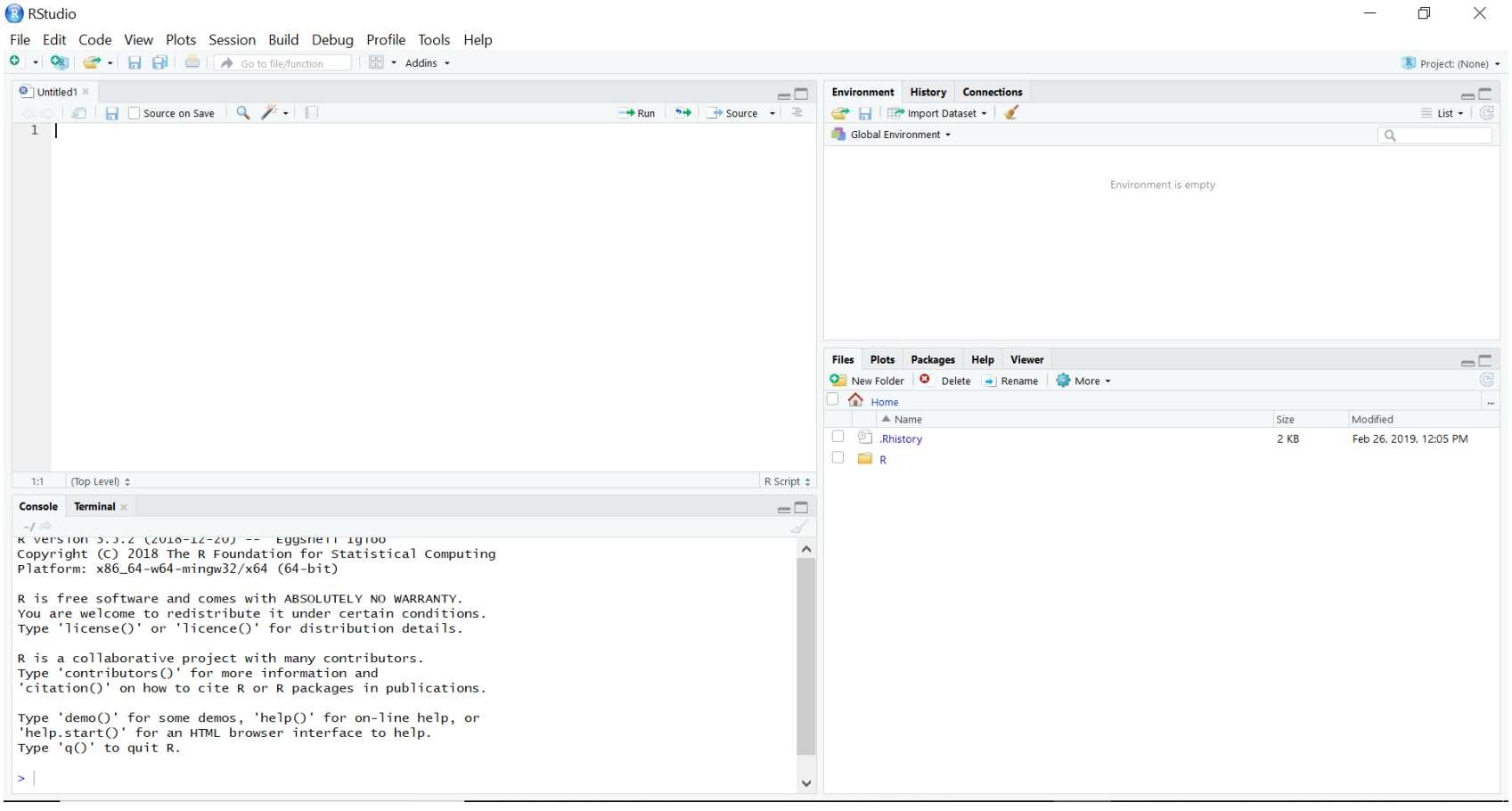
R: Open source software

- R is free and open source software, distributed according to the GNU public license.
- Likewise, virtually all of contributed R packages are likewise free and open source.
- In practical terms, this means that is freely available for everyone to use, now and forever, on more or less any device they choose.
- Open source software always has the potential to *go viral* and develop a large self-sustaining community of user/developers. This has arguably happened with R.

R: Popularity and widespread use

- When it comes to the computational implementation of modern statistical methods, R is the de facto standard. For example, the Journal of Statistical Software is overwhelmingly dominated by programs written in R.
- R is also currently very highly ranked according to many rankings of widely used programming languages of any kind. It ranked in the top 10 or top 20 most widely used programming languages.
- R is ranked as one of the top five most popular data science programs in jobs for data scientists, and in multiple surveys of data scientists, it is often ranked as the first or second mostly widely used data science tool.

A guided tour of RStudio



Introducing R commands

- A useful way to think about R, and not an inaccurate one either, is that it is simply a calculator.

```
> 2 + 2 # addition
#> [1] 4
> 3 - 5 # subtraction
#> [1] -2
> 3 * 2 # multiplication
#> [1] 6
> 4 / 3 # division
#> [1] 1.333333
> (2 + 2) ^ (3 / 3.5) # exponents and brackets
#> [1] 3.281341
```

Equality/inequality operations

- Testing for the equality or inequality of pairs of numbers, already starts to go beyond the usual capabilities of handheld calculator.

```
> 12 == (6 * 2)             # test for equality
#> [1] TRUE
> (3 * 4) != (18 - 7)       # test for inequality
#> [1] TRUE
> 3 < 10                    # less than
#> [1] TRUE
> (2 * 5) <= 10             # less than or equal
#> [1] TRUE
```

Logical values and logical operations

- In the previous step, the results are returned as either `TRUE` or `FALSE`. These are logical or *Boolean* values.
- Just as we can represent numbers and operations on numbers, so too can we have two logical values, `TRUE` and `FALSE` (always written in all uppercase), and Boolean operations (*and*, *or*, and *not*) on logical values.

```
> TRUE & FALSE  # logical and
#> [1] FALSE
> TRUE | TRUE   # logical or
#> [1] TRUE
> !TRUE         # logical not
#> [1] FALSE
> (TRUE | !TRUE) & !FALSE
#> [1] TRUE
```

Variables and assignment

- If we type the following at the command prompt and then press Enter, the result is displayed but not stored.

```
> (12/3.5)^2 + (1/2.5)^3 + (1 + 2 + 3)^0.33
#> [1] 13.6254
```

- We can, however, assign the value of the above calculation to a variable named x.

```
> x <- (12/3.5)^2 + (1/2.5)^3 + (1 + 2 + 3)^0.33
```

- Now, we can use x as is it were a number.

```
> x
#> [1] 13.6254
> x ^ 2
#> [1] 185.6516
> x * 3.6
#> [1] 49.05145
```

Assignment rules

- In general, the assignment rule is

```
name <- expression
```

The `expression` is any R code that returns some value.

- The `name` must consist of letters, numbers, dots, and underscores.

```
x123   # acceptable
.x
x_y_z
xXx_123
```

- It must begin with a letter or a dot that is not followed by a number.

```
_x   # not acceptable
.2x
x-y-z
```

- The recommendation is to use names that are meaningful, relatively short, without dots (using _ instead for punctuation), and primarily consisting of lowercase characters.

Vectors

- Vectors are one dimensional sequences of values.
- For example, if we want to create a vector of the first 6 primes numbers, we could do the following.

```
> primes <- c(2, 3, 5, 7, 11, 13)
```

- We can now perform operations (arithmetic, logical, etc) on the `primes` vector.

```
> primes + 1
#> [1]  3  4  6  8 12 14
> primes / 2
#> [1] 1.0 1.5 2.5 3.5 5.5 6.5
> primes == 3
#> [1] FALSE  TRUE FALSE FALSE FALSE FALSE
> primes >= 7
#> [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Indexing vectors

- For any vector, we can refer to individual elements using indexing operations.

```
> primes[1]
#> [1] 2
> primes[5]
#> [1] 11
```

- If we want to refer to sets of elements, rather than just individual elements, we can use vectors (made with the `c()` function) inside the indexing square brackets.

```
> primes[c(3, 5, 2)]
#> [1]  5 11  3
```

- If we use a negative valued index, we can refer to or all elements *except* one.

```
> primes[-1]
#> [1]  3  5  7 11 13
> primes[-2]
#> [1]  2  5  7 11 13
```

Vector types

- A vector be a sequence of numbers, logical values, or characters.

```
> nation <- c('ireland', 'england', 'scotland', 'wales')
```

- We can index this vector as we did with a vector of numbers.

```
> nation[1]
#> [1] "ireland"
> nation[2:3]
#> [1] "england"  "scotland"
> nation == 'ireland'
#> [1]  TRUE FALSE FALSE FALSE
```

- The `class` function in R will identify the data type of the vector.

```
> class(primes)
#> [1] "numeric"
> class(nation)
#> [1] "character"
> class(nation == 'ireland')
#> [1] "logical"
```

Data frames

- Data frames are rectangular data structures; they have certain number of columns, and each column has the same number of rows. Each column is in fact a vector.
- Usually, data frames are created when read in the contents of a data file, but we can produce them on the command line with the `data.frame()`.

```
> Df <- data.frame(name = c('billy', 'joe', 'bob'),
+                  age = c(21, 29, 23))
> Df
#>    name age
#> 1 billy  21
#> 2   joe  29
#> 3   bob  23
```

Indexing data frames

- We can refer to elements of a data frame in different ways.
- The simplest is to use double indices, one for the rows, one for the columns.

```
> Df[3, 2] # row 3, col 2
#> [1] 23
> Df[c(1, 3), 2] # rows 1 and 3, col 2
#> [1] 21 23
> Df[1,] # row 1, all cols
#>    name age
#> 1 billy  21
> Df[, 2] # all rows, col 2
#> [1] 21 29 23
```

Indexing data frames (contined)

- We could also refer to the column by name. To do so, we could use the following `$` notation.

```
> Df$age
#> [1] 21 29 23
```

- An alternative syntax that accompishes the same thing is to use *double* square brackets as follows.

```
> Df[['age']]
#> [1] 21 29 23
```

- A *single* square brackets, we would obtain the following.

```
> Df['age']
#>   age
#> 1  21
#> 2  29
#> 3  23
```

Functions

- In functions, we put data in, calculations or done to or using this data, and new data, perhaps just a single value, is then returned.
- There are probably hundreds of thousands of functions in R.
- For example,

```
> length(primes)
#> [1] 6
> sum(primes)
#> [1] 41
> mean(primes)
#> [1] 6.833333
> median(primes)
#> [1] 6
> sd(primes)
#> [1] 4.400758
> var(primes)
#> [1] 19.36667
```

Custom functions

- R makes it easy to create new functions.

```
> my_mean <- function(x){ sum(x)/length(x)}
```

- This `my_mean` takes a vector as input and divides its sum by the number of elements in it. It then returns this values. The `x` is a placeholder for whatever variable we input into the function.
- We would use it just as we would use `mean`.

```
> my_mean(primes)
#> [1] 6.833333
```

Writing R scripts

- Scripts are files where we write R commands, which can be then saved for later use.
- You can bring up RStudio's script editor with Ctrl+Shift+N, or go to the File/ New File/ R script, or click on the New icon on the left of the taskbar below the menu and choose R script.
- In a script, you can have as many lines of code as you wish, and there can be as many blank lines as you wish.

```
1 composites <- c(4, 6, 8, 9, 10, 12)
2
3 composites_plus_one <- composites + 1
4
5 composites_minus_one <- composites - 1
```

- If you place the cursor on line 1, you can then click the Run icon, or press the Ctrl+Enter keys.

Writing R scripts (continued)

One reason why writings in scripts is very practically valuable, even if you don't wish to save the scripts, is when you are write long and complex commands.

```
Df <- data.frame(name = c('jane', 'joe', 'billy'),
                 age = c(23, 27, 24),
                 sex = c('female', 'male', 'male'),
                 occupation = c('tinker', 'tailor', 'spy')
)
```

We can execute this command as if it were on a single line by placing the cursor anywhere on any line and pressing Ctrl+Enter.

Code comments

- An almost universal feature of programming language is the option to write *comments* in the code files.
- A comment allows you write to notes or comments around the code that is then skipped over when the script or the code lines are being executed.
- In R, anything following the # symbol on any line is treated as a comment.

```
# Here is a data frame with four variables.
# The variables are name, age, sex, occupation.
Df <- data.frame(name = c('jane', 'joe', 'billy'),
                 # This line is a comment too.
                 age = c(23, 27, 24), # Another comment.
                 sex = c('female','male', 'male'),
                 occupation = c('tinker', 'tailor', 'spy')
)
```

Packages

- There are presently 16466 contributed packages in R.
- The easiest way to install a package is to click the Install button on the top left of the *Packages* window in the lower right pane.
- You can also install a package or packages with the `install.packages` command.

```
> install.packages("dplyr")
> install.packages(c("dplyr", "tidyr", "ggplot2"))
```

- Having installed a package, it must be loaded to be used. This can be done by clicking the tick box before the package name in the *Packages* window, or use the `library` command.

```
> library("tidyverse")
```

Reading in data

- R allows you to import data from a very large variety of data file types, including from other statistics programs like SPSS, Stata, SAS, Minitab, and so on, and common file formats like `.xlsx` and `.csv`.
- When learning R initially, the easiest way to import data is using the Import Dataset button in the Environment window.
- If we use the *From Text (readr)...* option, it runs the `read_csv` R command, which we can run ourselves on the command line, or write in a script.

```
> library(readr)
> test_data <- read_csv("../data/data01.csv")
```

Viewing data

- The easiest way to view a data frames is to type its name.

```
> test_data
#> # A tibble: 100 x 3
#>    sex       iq test_score
#>    <chr>  <dbl>      <dbl>
#>  1 male      74       14.1
#>  2 female    83       16.0
#>  3 female    88       15.6
#>  4 male     105       11.8
#>  5 male      92       13.5
#>  6 male      92       13.4
#>  7 male     102       12.7
#>  8 female    80       14.9
#>  9 male     102       14.1
#> 10 female   114       15.7
#> # ... with 90 more rows
```

Viewing data (continued)

- Another option to view a data frame is to `glimpse` it.

```
> library(dplyr)
> glimpse(test_data)
#> Rows: 100
#> Columns: 3
#> $ sex        <chr> "male", "female", "female", "male", "male"
#> $ iq         <dbl> 74, 83, 88, 105, 92, 92, 102, 80, 102, 114
#> $ test_score <dbl> 14.14571, 16.01274, 15.64022, 11.82935, 13
```

Summarizing data with `summary`

- An easy way to summarize a data frame is with `summary`.

```
> summary(test_data)
#>      sex                  iq          test_score
#>  Length:100         Min.   : 60.0   Min.   :11.49
#>  Class :character   1st Qu.: 91.0   1st Qu.:13.26
#>  Mode  :character   Median :102.0   Median :14.63
#>                     Mean   :100.4   Mean   :14.50
#>                     3rd Qu.:108.2   3rd Qu.:15.65
#>                     Max.   :138.0   Max.   :18.40
```

Summarizing data with `summarize`

- The `summarize` (or `summarise`) command is a flexible way of getting summary statistics.

```
summarize(test_data,
          average_iq = mean(iq),
          average_score = mean(test_score),
          iq_stdev = sd(iq),
          score_var = var(test_score),
          n = n()
)
#> # A tibble: 1 x 5
#>   average_iq average_score iq_stdev score_var     n
#>        <dbl>         <dbl>    <dbl>     <dbl> <int>
#> 1       100.          14.5     14.5      2.30   100
```

Summarizing data by_group

- We can group the data frame into subsets based on the value of sex and then summarize.

```
summarize(group_by(test_data, sex),
          average_iq = mean(iq),
          average_score = mean(test_score),
          iq_stdev = sd(iq),
          score_var = var(test_score),
          n = n()
)
#> # A tibble: 2 x 6
#>   sex    average_iq average_score iq_stdev score_var     n
#>   <chr>       <dbl>         <dbl>    <dbl>     <dbl> <int>
#> 1 female      101.           15.4     15.6      1.46    51
#> 2 male         99.6          13.6     13.4      1.48    49
```