Introduction to R

Adam Rawles

Overview

- What is R and RStudio?
- Basic arithmetic operators
- Variable assigment
- Data types
- Data structures (including subsetting)
- Functions

Learning Objectives

- Understand the basics of RStudio
- Know the difference between the data types and structures in R
- Understand the concepts of how functions operate in R
- Know where/how to look things up!

What is R?

- Background
  - Public license programming language
  - Used mostly for statistical computing
  - Supported with many packages
- Interface
  - R uses a command-line interface
  - But there are many GUIs available (such as RStudio)

RStudio

- RStudio is one of the most popular R environments
- RStudio has 4 main panes:

RStudio

- Source
  - This is where your scripts (pre-written lines of codes) are displayed
  - If you click on a variable in the Environment pane, this will also be displayed in the source window
- Console
  - This is where the commands are actually executed
  - When you run a script from the source, it is passed to the console line by line

RStudio

- Environment
  - This pane displays any variables or data structures you've created
  - This pane will also display any custom functions you've created, which we'll move onto in later modules
- History/Files/Plots/Help
  - This pane can display a number of things
  - The most useful of which are the Files tab, which displays all the files in the folder you're working in
  - and the Plots tab, which will show any plots that you've created

Basic arithmetic

- The basic arithmetic operators in R are very similar to Excel

```
2 + 2
```

```
## [1] 4
```

```
2 * 3
```

```
## [1] 6
```

Basic arithmetic

```
6/2
```

```
## [1] 3
```

```
10 - 5
```

```
## [1] 5
```

Logical operators

- Logical operators in R are a bit different

```
1 == 1
```

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

```
1 != 2
```

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

```r
2 > 1
```

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

Logical operators

```r
1 < 2
```

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

```r
1 == 1 | 1 == 2
```

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

```r
1 == 1 & 1 == 2
```

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

Variable assignment

- Assigning variables allows you store values for later use

```r
variable_1 <- 5
variable_2 <- 10

variable_3 <- variable_2/variable_1
print(variable_3)
```

```
## [1] 2
```

- For variable assignment, the "<-" and "=" operators can be used interchangeably
- But it's better to use "<-"
- Note: the value of a variable is not printed when it is assigned

Variable assigment

- Variables can also be modified after they've been assigned

```r
variable_1 <- 1
print(variable_1)
```

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

```r
variable_1 <- "hello world"
print(variable_1)
```

```
## [1] "hello world"
```

Data types

- Data comes in lots of different forms
- Is "True" equal to TRUE?
- The main data types are:
  - logical; TRUE, FALSE
  - numeric; 12.5, 1, 999
  - integer; 2L, 34L, 1294L
  - character; "hello world", "True"

Data types

- Assigning the right data type is important, as it determines how the data is stored and how data can be manipulated

```
variable_1 <- 5
variable_2 <- "5"

variable_1 + variable_2
```

```
## Error in variable_1 + variable_2: non-numeric argument to binary operator
```

Data types

- R will automatically decide the data type when you assign a variable, but you can force R to store the value as a different data type using the as.xxxxx functions

```
variable_1 <- as.numeric("5")
variable_2 <- as.integer("5")

variable_1 + variable_2
```

```
## [1] 10
```

```
variable_1 <- as.numeric("hello world")
```

```
## Warning: NAs introduced by coercion
```

```
variable_1
```

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

Data types - Factors

- There is another data type which is fairly common in R called a factor
- A factor is made up of one or more levels that represent some form of grouping
- For example, if we had a dataframe containing information about all the meters in a GSP Group, we might store "Measurement Class" as a factor, with the levels A, C, E, F, G
- Note: when importing data into R, R will automatically try and convert columns containing strings into factors (unless you specify otherwise)

Data structures

- Values in R need to be stored in a specific way
- There are 4 main data structures in R:

Data structures - Vectors

- Vectors
  - Vectors in R are arrays of data in one dimension
  - They are atomic (not recursive)
  - Even variables with a single value will be stored as a vector with length 1

```
vector_1 <- c(1,2,3,4,5,6)
print(vector_1)
```

```
## [1] 1 2 3 4 5 6
```

```
vector_2 <- 1
print(vector_2)
```

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

Data structures - Matrices

- Matrices
  - Matrices are two-dimensional arrays that hold values of the same type
  - Matrices can have named rows or columns, but they cannot be subsetted by those names (more later on)

```
matrix_1 <- matrix(c(1,2, 3,4), nrow = 2, ncol = 2)
print(matrix_1)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Data structures - Data frames

- Data frames
  - Most data you'll be working with in R will be held in a data frame
  - Data frames are two dimensional arrays that can hold values of different types and have named columns and rows

```
dataframe1 <- data.frame(numeric_col = c(1,2), character_col = c("hello", "world"), stringsAsFactors = F
print(dataframe1)
```

```
##   numeric_col character_col
## 1           1         hello
## 2           2         world
```

- Important notes:
  - Values in the same column must be of the same type
  - Each column must have the same number of rows

Data structures - Lists

- Lists
  - Lists are similar to vectors
  - However, lists can contain values of any type, including other lists
  - This means you can have a list of data frames, or a list of lists of data frames!

```
list_1 <- list(numbers = c(1,2,3,4), characters = (c("hello", "world")))
print(list_1)
```

```
## $numbers
## [1] 1 2 3 4
##
## $characters
## [1] "hello" "world"
```

Subsetting data structures

- More often than not, you'll only want to access one or more of the values in a data structure
- For example, you may only want to see the first 10 values of a very long vector
- For vectors, this is done by suffixing the variable with a [] containing the index of values you want

```
values <- c(6,45,7,54,99,31,12,15,67,100)
values[1]
```

```
## [1] 6
```

Subsetting data structures

```
values[c(1,10)]
```

```
## [1]   6 100
```

```
values[1:5] ##this is the same as values[c(1,2,3,4,5)]
```

```
## [1]  6 45  7 54 99
```

Subsetting vectors - exercise

- Create a vector
  - Return the 5th value
  - Return the 1st and 3rd value
  - Return the 1st, 2nd, 3rd, 4th, 5th and 7th value

```
vector1 <- c(1,5,8,9,67,5,3,2,5,6)
vector1[5]
```

```
## [1] 67
```

```
vector1[c(1,3)]
```

```
## [1] 1 8
```

```
vector1[c(1:5, 7)]
```

```
## [1]  1  5  8  9 67  3
```

Subsetting matrices

- For data structures with more that one-dimension, we have to provide some more information
- In these cases, we specify an index for each dimension, starting with the row

```
matrix_1 <- matrix(c(1,2,3,4), ncol = 2, nrow = 2)
matrix_1[1,2]
```

```
## [1] 3
```

```
matrix_1[,2]
```

```
## [1] 3 4
```

- Matrices can also be subsetted by providing the names of the row/column instead of the number

Subsetting data frames

- Unlike matrices, data frame columns can be referred to be name using the $ operator
- You can combine the $ and [] operators to access specific values in a column

```
dataframe_1 <- data.frame(numeric_col = c(1,2), character_col = c("hello", "world"), stringsAsFactors =
dataframe_1$character_col
```

```
## [1] "hello" "world"
```

```
dataframe_1$character_col[1]
```

```
## [1] "hello"
```

- Note: data frames can also be subsetted using the [x, y] format, with names or indices

Subsetting dataframes - exercise

- Create a data frame with 2 columns
  - Return one column using the $ operator
  - Return the other column using the [,] approach

```
df1 <- data.frame(numeric_col = c(1,2,3), logical_col = c(TRUE, FALSE, TRUE), stringsAsFactors = FALSE)
df1$numeric_col[1]
```

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

```
df1[,2]
```

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

Subsetting lists

- Because lists have named values, they can also be subsetted using the $ and [] operators

```
list_1 <- list(numbers = c(1,2,3,4), characters = (c("hello", "world")))

list_1$numbers
```

```
## [1] 1 2 3 4
```

```
list_1$characters[1]
```

```
## [1] "hello"
```

Subsetting lists

- Although lists are two-dimensional, they do not have rows and columns and so cannot be subsetted using the [x, y] format
- Instead, lists are subsetted using the [[]] operator
- This [[]] operator can then be combined with whatever operator is appropriate for the type of structure in the list (e.g. $ for a data frame, or [] for a vector)

```
list_1 <- list(numbers = c(1,2,3,4), characters = (c("hello", "world")))
list_1[[1]][2]
```

```
## [1] 2
```

Functions

- Any operation, such as loading a dataset or finding the mean, are defined as functions in R
- A function is a set of steps that takes an input, does some form of computation, and returns an output
- For example, the mean function takes values and returns the mean of those values

```
values <- c(10,20,30)
mean(values)
```

```
## [1] 20
```

- You may have noticed that "c()" also acts as a function, creating a vector with the provided values

Functions - input parameters/arguments

- Almost all functions will require some input to produce an output
- These are the function's input parameters or arguments
- For example, in the previous slide, the mean() function required a vector of values to calculate the mean

Functions - input parameters/arguments

- Some functions will have optional input parameters
- To see what arguments/parameters a function requires, type the function name preceded by a "?" in the R console (e.g. ?mean)

Functions - exercise

- Find the help page for the "sample" function
- Use the "sample" function

Functions - naming input parameters

- When you provide unnamed values to a function, R will automatically assign them to an input argument in the order they appear on the function's help page
- This can be hard to keep track of if you're providing multiple arguments
- Best practice, therefore, is to be explicit when providing your input arguments
- Let's use the substr() function as an example

Functions - naming input parameters

- The substr() function returns part of a string from a start and stop index

```
string1 <- "hello world"
substr(string1, 1,5)
```

```
## [1] "hello"
```

- The substr() function expects 3 arguments;
  - The string (x)
  - The start point (start)
  - The end point (stop)
- As shown above, we can just pass those arguments without naming them, in that order

Functions - naming input parameters

- It's best practice, however, to name the arguments as you pass them...

```
string1 <- "hello world"
substr(x = string1, start = 1, stop = 5)
```

```
## [1] "hello"
```

Recap

- R is a free statistical programming language
- One of the most popular environments for R is RStudio
- Mathematical operators in R are mostly the same as in Excel (e.g. +, -, *, /...)
- Values in R can take different forms, and it's important to get the form correct!
- Values can then be stored in lots of different structures
- These structures can be subsetted using different approaches ([], [x,y], $, [[]])
- Almost every action in R is performed via a function
- A function takes an input, does some computation and returns an output
- It's always good to be explicit when providing arguments to functions