
BELGIUM CAMPUS
iTversity
It's the way we're wired
Faculty of Information Technology
BELGIUM CAMPUS
ITVERSITY
Bothlale Village

BIN381

Learning objectives

- Learn how to install R.
- Learn how to program in R.

What is R

- The R programming language is an offshoot of a programming language called S.
- It was developed by Ross Ihaka and Robert Gentle-man from the University of Auckland,
- R currently is free to download and use.

How to Install R

- R works on many operating systems including Windows, Macintosh, and Linux.
- You can download from http://www.r-project.org/index.html
- to install R all you need to double-click on the executable file and follow the instructions on screen.

Programming in R

- R has all the functions of a normal calculator; here we'll quickly look at how to use these functions.
- > 1 + 1
- The output should look like this:
- [1] 2
- All arithmetic operators can be used
- The operators available are + (addition), - (subtraction), / (division), * (multiplication), and ^ (raise to a power).

Working with scripts and markdown files

- When working with **R**, you can type everything into the console
- However, you've probably already noticed this has some disadvantages
  - It's easy to make mistakes, and annoying to type everything over again to just correct one letter
  - It's also easy to loose track of what you've written.
- As you move on to working on larger, more complicated you will quickly find that you need a better way to keep track of your analyses
- Luckily **R** and RStudio provide a number of good ways to do this.

R scripts

- Scripts offer the simplest form of repeatable analysis in **R**.
- Scripts are just text files that contain code and comments. Script files should end in .R.
- In RStudio, open a new script using the File menu

: 

- Save it in your current working directory with an appropriate name
- Note that your working directory is the default location
- Use the History tab in RStudio to save some time copying your code from console to script
- Select one line by clicking on it, and send it to your script file using To Source.
- To select more than one line at a time, hold down Shift key as you select.

Using scripts to program with objects

- Objects are named variables that can hold different values.

```
# Working with objects

# Define two objects
x <- 5
y <- 2 * x

# ... and use them in basic a calculation.
x + y
```

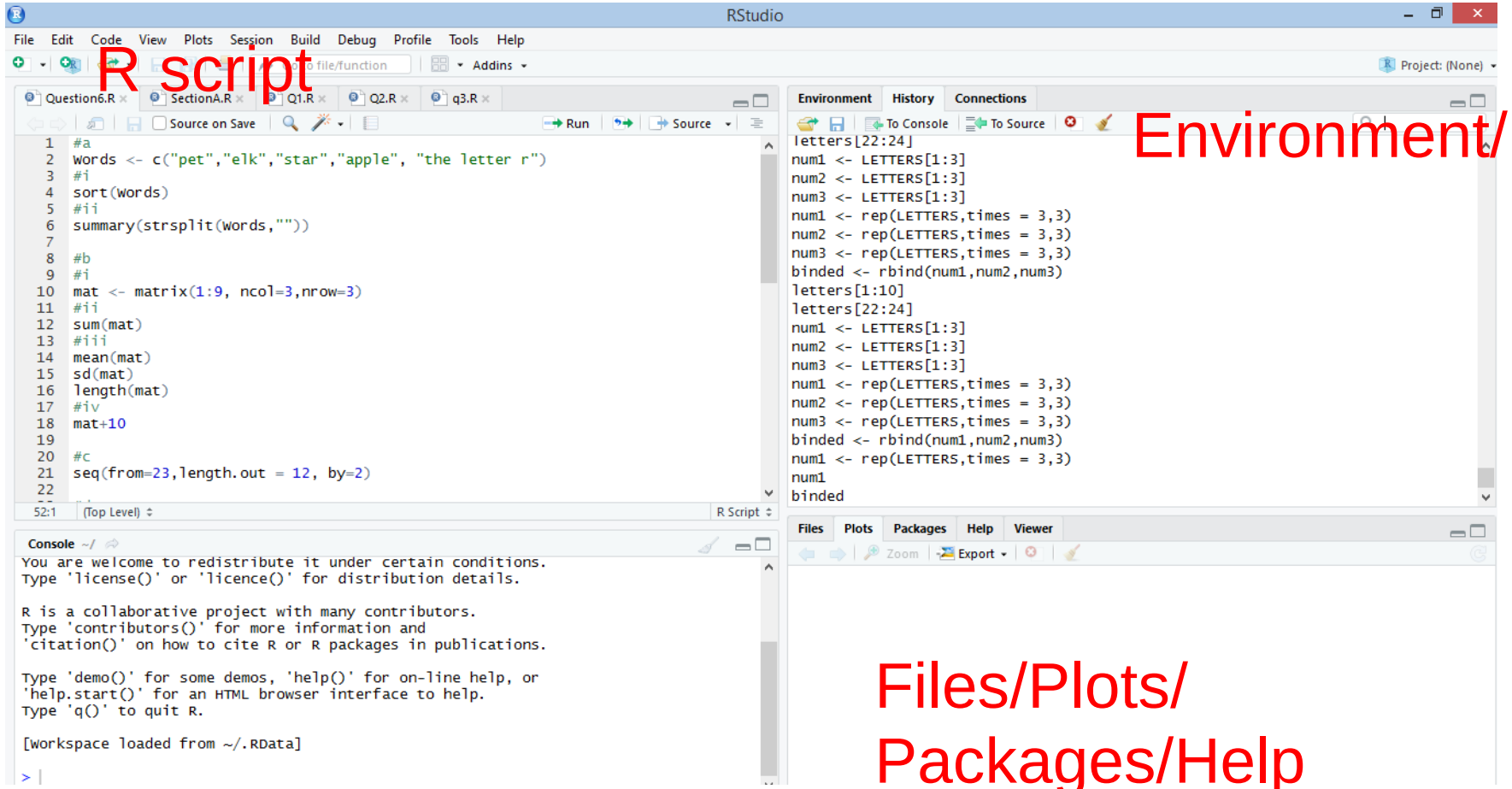
- You can run your entire script by clicking the Source button in the top right and chose source with echo. Or, more conveniently, you can select sections of your script with the mouse, and click Run , or by pressing Ctrl + Enter

Using scripts to program with objects

- Note that RStudio has four panes: the R script, the R console, Files/Plots/ Packages/Help and Environment/History.

- You can change the placement of the panes in Tools Global options... Pane layout . Change the layout to your liking.
- You can also change the appearance of code in the script pane and the R console pane

Introduction

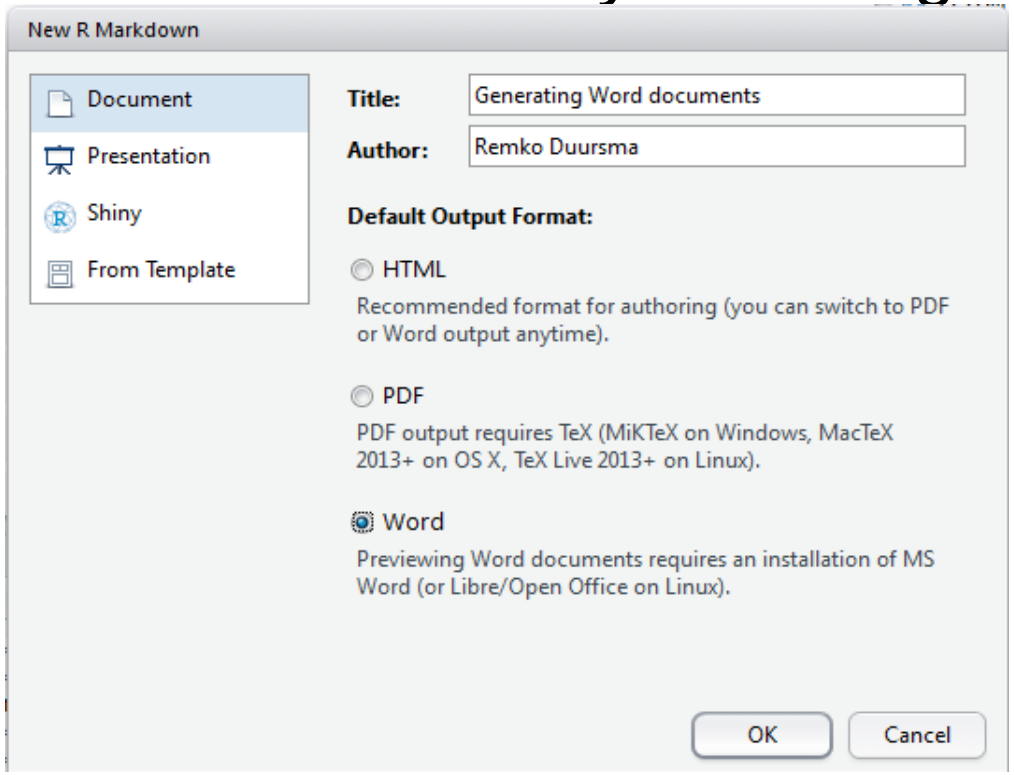


Working with markdown files

- Script files are good for simple analyses, and they are great for storing small amounts of code that you would like to use in lots of different projects. But, they are not the best way to share your results with others.
- R markdown makes a great way to keep track of your work as you are developing an analysis
- Markdown is a simple set of rules for formatting text files so that they are both human-readable and processable by software
- The knitr package (as used by RStudio) will take a markdown-formatted text file and produce nicely-formatted output.
- The output can be a Word document, HTML page, or PDF file

Working with markdown files

- RStudio offers a handy editor for markdown files. Start a new markdown file by choosing File ➔New File→R Markdown...



Go to File→ New File →R Markdown , enter a title for your document and select Word document

Working with markdown files

- The first thing in the file is the header.
- Includes your name, a title, the date you created the file,
- the type of output you would like to produce (in this case, a Word document)

```
---
title: "Basic R calculations in markdown"
author: "Remko Duursma"
date: "16 September 2015"
output: word_document
---
```

R code in markdown

- The first thing you will see under the header in your new markdown document is a grey box

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
```

- This box contains a **chunk** of R code to set the options for the markdown document and the software that processes it (a package known as knitr).
- Notice that the chunk starts and ends with three accent characters
- This tells the software that you are starting a chunk of R code.
- You can use this syntax to add your own chunks of code to the markdown 1le.

Options for code chunks within an markdown document.

Option	What it sets	Possible values
echo	Should the code be shown in the final document?	TRUE, FALSE
eval	Should the results be shown in the final document?	TRUE, FALSE

Type R code just as you normally would within the code chunks themselves

All of the text outside of the R code chunks is interpreted as body text

o

Creating more descriptive text blocks Basic markdown formatting

Formatting option	Symbols	Example
Headings	#	#Example Heading
Subheadings	##	##Subheading
Bold	**	**bold text**
Italic	*	*italic text*
Strike through	~~	~~crossed-out text~~
Superscript	^	x^2^
Subscript	~	CO~2~
Bulleted lists	*	* A list item * Another list item * Yet another list item
Numbered lists	1.	1. First list item 2. Second list item 3. Third list item
Horizontal rule	three or more -	---
Line break	two or more spaces plus return	

R Data structures

- R's base data structures can be organised by their dimensionality (1d, 2d, or nd)
- The data structures are either homogeneous (all contents must be of the same type) or heterogeneous (the contents can be [illegible])

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Almost all other objects are built upon these foundations
To get what data structures it's composed use str()..

Vectors

- The basic data structure in R is the vector and comes in two flavours:
  1. atomic vectors
  2. lists.
- Common properties:
  1. Type, typeof(), what it is.
  2. Length, length(), how many elements it contains.
  3. Attributes, attributes(), additional arbitrary metadata.

Differences

- All elements of an atomic vector must be the same type,elements of a list can have different types.
- **is.vector()** does not test if an object is a vector Instead it returns TRUE only if the object is a vector with no attributes apart from names.
- Use is.atomic(x) || is.list(x) to test if an object is actually a vector.

Atomic vectors

- Four common types of atomic vectors exist
  1. logical,
  2. integer,
  3. double (often called numeric),
  4. character.

  Atomic vectors are created with c()
- dbl_var <- c(1, 2.5, 4.5)
- # With the L suffix, you get an integer rather than a double
- int_var <- c(1L, 6L, 10L)
- # Use TRUE and FALSE (or T and F) to create logical vectors
- log_var <- c(TRUE, FALSE, T, F)
- chr_var <- c("these are", "some strings")

Atomic vectors

- Atomic vectors are always flat, even if you nest c()'s:
- c(1, c(2, c(3, 4)))
- #> [1] 1 2 3 4
- # the same as
- c(1, 2, 3, 4)
- #> [1] 1 2 3 4

Types and tests

- Given a vector, you can determine its type with typeof(),
- check if it's a specific type with "is" function:
- Eg:
    1. is.character(),
    2. is.double(),
    3. is.integer(),
        - **int_var <- c(1L, 6L, 10L)**
        - **typeof(int_var)**
        - **#> [1] "integer"**
    4. is.logical(),
    5. s.atomic()

Coercion

All elements of an atomic vector must be the same type, so when you

attempt to combine different types they will be **coerced** to the most

flexible type in the order (logical(least), integer, double, and character(most).)

- Eg combining a character and an integer yields a character

```
str(c("a", 1))
#> chr [1:2] "a" "1"
```

Lists

- Lists are different from atomic vectors because their elements can be of any type, including lists.
- construct lists by using list() instead of c()
- x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
- Lists are sometimes called **recursive** vectors, because a list can contain other lists.
- x <- list(list(list(list())))
- x <- list(list(1, 2), c(3, 4))
- The typeof() a list is list.
- Test if with is.list() eg is.list(mtcars)
- Coerce to a list with as.list().
- Turn a list into an atomic vector with unlist().
- If the elements of a list have different types, unlist() uses the same coercion rules as c().

Exercises

- 1. What are the six types of atomic vector? How does a list differ from an atomic vector?
- 2. Test your knowledge of vector coercion rules by predicting the output of the following uses of c():
  a. c(1, FALSE)
  b. c("a", 1)
  c. c(list(1), "a")
  d. c(TRUE, 1L)
- 3. Why do you need to use unlist() to convert a list to an atomic vector?

  Why doesn't as.vector() work?
- 4. Why is 1 == "1" true? Why is -1 < FALSE true? Why is "one" < 2 false?

Attributes

- All objects can have arbitrary additional attributes, used to store metadata about the object.
- Attributes can be thought of as a named list (with unique names).
- Attributes can be accessed individually with attr() or all at once (as a list) with attributes().
- Both the names and the dimensions of matrices and arrays are stored in R as attributes of the object.
- Attributes can be seen as labelled values you can attach to any object
- The structure() function returns a new object with modified attributes:
- Most attributes are lost when modifying a vector
- Eg attributes(y[1]) #> NULL

Attributes

- The only attributes not lost are the three most important:
  1. Names, a character vector giving each element a name
  2. Dimensions, used to turn vectors into matrices and arrays
  3. Class, used to implement the S3 object system
- When working with these attributes use:
  1. names(x), not attr(x, "names"),
  2. dim(x) not attr(x, "dim").
  3. class(x), not attr(x, "class"),

Names

- You can name a vector in three ways:

1. When creating it: x <- c(a = 1, b = 2, c = 3).
2. By modifying an existing vector in place: x <- 1:3; names(x) <- c("a","b","c").
3. By creating a modified copy of a vector

- Not all elements of a vector need to have a name.vector: x <- setNames(1:3, c("a","b", "c")).
- If all names are missing, names() will return NULL.
- y <- c(a = 1, 2, 3)
- names(y)
- #> [1] "a" "" ""
- You can create a new vector without names using unname(x), or remove names in place with names(x) <- NULL.