Introduction to Geoinformatics

tbd

04 November, 2020

Table of Contents

# Introduction to R

## About this module

This module will provide you with the fundamental skills in

* basic programming in R
* data wrangling

**Learning objectives**

* xxx
* xxx

Upon the completion of this lesson, you will ....

## R programming language

One of the most widely used programming languages and an effective tool for *(geospatial)* data science

* data wrangling
* statistical analysis
* machine learning
* data visualisation and maps
* processing spatial data
* geographic information analysis

## Schedule

The lectures and practical sessions have been designed to follow the schedule below

xxx

## Reference books

Suggested reading

* *Programming Skills for Data Science: Start Writing Code to Wrangle, Analyze, and Visualize Data with R* by Michael Freeman and Joel Ross, Addison-Wesley, 2019. See book [webpage](https://www.pearson.com/us/higher-education/program/Freeman-Programming-Skills-for-Data-Science-Start-Writing-Code-to-Wrangle-Analyze-and-Visualize-Data-with-R/PGM2047488.html) and [repository](https://programming-for-data-science.github.io/).
* *R for Data Science* by Garrett Grolemund and Hadley Wickham, O'Reilly Media, 2016. See [online book](https://r4ds.had.co.nz/).
* *Discovering Statistics Using R* by Andy Field, Jeremy Miles and Zoë Field, SAGE Publications Ltd, 2012. See book [webpage](https://www.discoveringstatistics.com/books/discovering-statistics-using-r/).
* *Machine Learning with R: Expert techniques for predictive modeling* by Brett Lantz, Packt Publishing, 2019. See book [webpage](https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788295864).

Further reading

* *The Art of R Programming: A Tour of Statistical Software Design* by Norman Matloff, No Starch Press, 2011. See book [webpage](https://nostarch.com/artofr.htm)
* *An Introduction to R for Spatial Analysis and Mapping* by Chris Brunsdon and Lex Comber, Sage, 2015. See book [webpage](https://uk.sagepub.com/en-gb/eur/an-introduction-to-r-for-spatial-analysis-and-mapping/book241031)
* *Geocomputation with R* by Robin Lovelace, Jakub Nowosad, Jannes Muenchow, CRC Press, 2019. See [online book](https://bookdown.org/robinlovelace/geocompr/).

## R

Created in 1992 by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand

* Free, open-source implementation of *S*
  + statistical programming language
  + Bell Labs
* Functional programming language
* Supports (and commonly used as) procedural (i.e., imperative) programming
* Object-oriented
* Interpreted (not compiled)

## Interpreting values

When values and operations are inputted in the *Console*, the interpreter returns the results of its interpretation of the expression

2

## [1] 2

"String value"

## [1] "String value"

# comments are ignored

## Basic types

R provides three core data types

* numeric
  + both integer and real numbers
* character
  + i.e., text, also called *strings*
* logical
  + TRUE or FALSE

## Numeric operators

R provides a series of basic numeric operators

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Meaning | Example | Output |
| + | Plus | 5 + 2 | 7 |
| - | Minus | 5 - 2 | 3 |
| \* | Product | 5 \* 2 | 10 |
| / | Division | 5 / 2 | 2.5 |
| %/% | Integer division | 5 %/% 2 | 2 |
| %% | Module | 5 %% 2 | 1 |
| ^ | Power | 5^2 | 25 |

5 + 2

## [1] 7

## Logical operators

R provides a series of basic logical operators to test

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Meaning | Example | Output |
| == | Equal | 5 == 2 | FALSE |
| != | Not equal | 5 != 2 | TRUE |
| > (>=) | Greater (or equal) | 5 > 2 | TRUE |
| < (<=) | Less (or equal) | 5 <= 2 | FALSE |
| ! | Not | !TRUE | FALSE |
| & | And | TRUE & FALSE | FALSE |
| | | Or | TRUE | FALSE | TRUE |

5 >= 2

## [1] TRUE

## Summary

An introduction to R

* Basic types
* Basic operators

**Next**: Core concepts

* Variables
* Functions
* Libraries

# Core concepts

## Recap

**Prev**: An introduction to R

* Basic types
* Basic operators

**Now**: Core concepts

* Variables
* Functions
* Libraries

## Variables

Variables **store data** and can be defined

* using an *identifier* (e.g., a\_variable)
* on the left of an *assignment operator* <-
* followed by the object to be linked to the identifier
* such as a *value* (e.g., 1)

a\_variable <- 1

The value of the variable can be invoked by simply specifying the **identifier**.

a\_variable

## [1] 1

## Algorithms and functions

*An* **algorithm** *or effective procedure is a mechanical rule, or automatic method, or programme for performing some mathematical operation* (Cutland, 1980).

A **program** is a specific set of instructions that implement an abstract algorithm.

The definition of an algorithm (and thus a program) can consist of one or more **function**s

* set of instructions that preform a task
* possibly using an input, possibly returning an output value

Programming languages usually provide pre-defined functions that implement common algorithms (e.g., to find the square root of a number or to calculate a linear regression)

## Functions

Functions execute complex operations and can be invoked

* specifying the *function name*
* the *arguments* (input values) between simple brackets
  + each *argument* corresponds to a *parameter*
  + sometimes the *parameter* name must be specified

sqrt(2)

## [1] 1.414214

round(1.414214, digits = 2)

## [1] 1.41

## Functions and variables

* functions can be used on the right side of <-
* variables and functions can be used as *arguments*

sqrt\_of\_two <- sqrt(2)  
sqrt\_of\_two

## [1] 1.414214

round(sqrt\_of\_two, digits = 2)

## [1] 1.41

round(sqrt(2), digits = 2)

## [1] 1.41

## Naming

When creating an identifier for a variable or function

* R is a **case sensitive** language
  + UPPER and lower case are not the same
  + a\_variable is different from a\_VARIABLE
* names can include
  + alphanumeric symbols
  + . and \_
* names must start with
  + a letter

## Libraries

Once a number of related, reusable functions are created

* they can be collected and stored in **libraries** (a.k.a. *packages*)
* install.packages is a function that can be used to install libraries (i.e., downloads it on your computer)
* library is a function that *loads* a library (i.e., makes it available to a script)

Libraries can be of any size and complexity, e.g.:

* base: base R functions, including the sqrt function above
* rgdal: implementation of the [GDAL (Geospatial Data Abstraction Library)](https://gdal.org/) functionalities

## stringr

R provides some basic functions to manipulate strings, but the stringr library provides a more consistent and well-defined set

library(stringr)  
str\_length("Leicester")

## [1] 9

str\_detect("Leicester", "e")

## [1] TRUE

str\_replace\_all("Leicester", "e", "x")

## [1] "Lxicxstxr"

## Summary

Core concepts

* Variables
* Functions
* Libraries

**Next**: Tidyverse

* Tidyverse libraries
* *pipe* operator

# Data Types

## Recap

**Prev**: Introduction

* 101 Lecture: Introduction to R
* 102 Lecture: Core concepts
* 103 Lecture: Tidyverse
* 104 Practical session

**Now**: Data types

* vectors
* factors
* matrices, arrays
* lists

## Vectors

**Vectors** are ordered list of values.

* Vectors can be of any data type
  + numeric
  + character
  + logic
* All items in a vector have to be of the same type
* Vectors can be of any length

## Defining vectors

A vector variable can be defined using

* an **identifier** (e.g., a\_vector)
* on the left of an **assignment operator** <-
* followed by the object to be linked to the identifier
* in this case, the result returned by the function c
* which creates a vector containing the provided elements

a\_vector <- c("Birmingham", "Derby", "Leicester",  
 "Lincoln", "Nottingham", "Wolverhampton")  
a\_vector

## [1] "Birmingham" "Derby" "Leicester" "Lincoln"   
## [5] "Nottingham" "Wolverhampton"

## Creating vectors

* the operator :
* the function seq
* the function rep

4:7

## [1] 4 5 6 7

seq(1, 7, by = 0.5)

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0

seq(1, 10, length.out = 7)

## [1] 1.0 2.5 4.0 5.5 7.0 8.5 10.0

rep("Ciao", 4)

## [1] "Ciao" "Ciao" "Ciao" "Ciao"

## Selection

Each element of a vector can be retrieved specifying the related **index** between square brackets, after the identifier of the vector. The first element of the vector has index 1.

a\_vector[3]

## [1] "Leicester"

A vector of indexes can be used to retrieve more than one element.

a\_vector[c(5, 3)]

## [1] "Nottingham" "Leicester"

## Functions on vectors

Functions can be used on a vector variable directly

a\_numeric\_vector <- 1:5  
a\_numeric\_vector + 10

## [1] 11 12 13 14 15

sqrt(a\_numeric\_vector)

## [1] 1.000000 1.414214 1.732051 2.000000 2.236068

a\_numeric\_vector >= 3

## [1] FALSE FALSE TRUE TRUE TRUE

## Any and all

Overall expressions can be tested using the functions:

* **any**, TRUE if any of the elements satisfies the condition
* **all**, TRUE if all of the elements satisfy the condition

any(a\_numeric\_vector >= 3)

## [1] TRUE

all(a\_numeric\_vector >= 3)

## [1] FALSE

## Factors

A **factor** is a data type similar to a vector. However, the values contained in a factor can only be selected from a set of **levels**.

houses\_vector <- c("Bungalow", "Flat", "Flat",  
 "Detached", "Flat", "Terrace", "Terrace")  
houses\_vector

## [1] "Bungalow" "Flat" "Flat" "Detached" "Flat" "Terrace"   
## [7] "Terrace"

houses\_factor <- factor(c("Bungalow", "Flat", "Flat",  
 "Detached", "Flat", "Terrace", "Terrace"))  
houses\_factor

## [1] Bungalow Flat Flat Detached Flat Terrace Terrace   
## Levels: Bungalow Detached Flat Terrace

## table

The function **table** can be used to obtain a tabulated count for each level.

houses\_factor <- factor(c("Bungalow", "Flat", "Flat",  
 "Detached", "Flat", "Terrace", "Terrace"))  
houses\_factor

## [1] Bungalow Flat Flat Detached Flat Terrace Terrace   
## Levels: Bungalow Detached Flat Terrace

table(houses\_factor)

## houses\_factor  
## Bungalow Detached Flat Terrace   
## 1 1 3 2

## Specified levels

A specific set of levels can be specified when creating a factor by providing a **levels** argument.

houses\_factor\_spec <- factor(  
 c("People Carrier", "Flat", "Flat", "Hatchback",  
 "Flat", "Terrace", "Terrace"),  
 levels = c("Bungalow", "Flat", "Detached",  
 "Semi", "Terrace"))  
table(houses\_factor\_spec)

## houses\_factor\_spec  
## Bungalow Flat Detached Semi Terrace   
## 0 3 0 0 2

## (Unordered) Factors

In statistics terminology, (unordered) factors are **categorical** (i.e., binary or nominal) variables. Levels are not ordered.

income\_nominal <- factor(  
 c("High", "High", "Low", "Low", "Low",  
 "Medium", "Low", "Medium"),  
 levels = c("Low", "Medium", "High"))

The *greater than* operator is not meaningful on the income\_nominal factor defined above

income\_nominal > "Low"

## Warning in Ops.factor(income\_nominal, "Low"): '>' not meaningful for  
## factors

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

## Ordered Factors

In statistics terminology, ordered factors are **ordinal** variables. Levels are ordered.

income\_ordered <- ordered(  
 c("High", "High", "Low", "Low", "Low",  
 "Medium", "Low", "Medium"),  
 levels = c("Low", "Medium", "High"))  
income\_ordered > "Low"

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

sort(income\_ordered)

## [1] Low Low Low Low Medium Medium High High   
## Levels: Low < Medium < High

## Matrices

**Matrices** are collections of numerics arranged in a two-dimensional rectangular layout

* the first argument is a vector of values
* the second specifies number of rows and columns
* R offers operators and functions for matrix algebra

a\_matrix <- matrix(c(3, 5, 7, 4, 3, 1), c(3, 2))  
a\_matrix

## [,1] [,2]  
## [1,] 3 4  
## [2,] 5 3  
## [3,] 7 1

## Arrays

:::::: {.cols data-latex=""}

::: {.col data-latex="{0.5}"}

Variables of the type **array**are higher-dimensionalmatrices.

* the first argument is avector containing thevalues
* the second argument isavector specifying thedepth of each dimension

a3dim\_array <- array(1:24, dim=c(4, 3, 2))

a3dim\_array

:::

::: {.col data-latex="{0.5}"}

## , , 1  
##   
## [,1] [,2] [,3]  
## [1,] 1 5 9  
## [2,] 2 6 10  
## [3,] 3 7 11  
## [4,] 4 8 12  
##   
## , , 2  
##   
## [,1] [,2] [,3]  
## [1,] 13 17 21  
## [2,] 14 18 22  
## [3,] 15 19 23  
## [4,] 16 20 24

::: ::::::

## Selection

Subsets of matrices (and arrays) can be selected as seen for vectors.

a\_matrix[2, c(1, 2)]

## [1] 5 3

a3dim\_array[c(1, 2), 2, 2]

## [1] 17 18

## Lists

Variables of the type **list** can contain elements of different types (including vectors and matrices), whereas elements of vectors are all of the same type.

employee <- list("Stef", 2015)  
employee

## [[1]]  
## [1] "Stef"  
##   
## [[2]]  
## [1] 2015

employee[[1]] # Note the double square brackets for selection

## [1] "Stef"

## Named Lists

In **named lists** each element has a name, and elements can be selected using their name after the symbol $.

employee <- list(employee\_name = "Stef", start\_year = 2015)  
employee

## $employee\_name  
## [1] "Stef"  
##   
## $start\_year  
## [1] 2015

employee$employee\_name

## [1] "Stef"

## Recap

Data types

* Vectors
* Factors
* Matrices, arrays
* Lists

**Next**: Control structures

* Conditional statements
* Loops

# Control structures

## Recap

**Prev**: Data types

* Vectors
* Factors
* Matrices and arrays
* Lists

**Now**: Control structures

* Conditional statements
* Loops

## If

Format: **if** (*condition*) *statement*

* *condition*: expression returning a logic value (TRUE or FALSE)
* *statement*: any valid R statement
* *statement* only executed if *condition* is TRUE

a\_value <- -7  
if (a\_value < 0) cat("Negative")

## Negative

a\_value <- 8  
if (a\_value < 0) cat("Negative")

## Else

Format: **if** (*condition*) *statement1* **else** *statement2*

* *condition*: expression returning a logic value (TRUE or FALSE)
* *statement1* and *statement2*: any valid R statements
* *statement1* executed if *condition* is TRUE
* *statement2* executed if *condition* is FALSE

a\_value <- -7  
if (a\_value < 0) cat("Negative") else cat("Positive")

## Negative

a\_value <- 8  
if (a\_value < 0) cat("Negative") else cat("Positive")

## Positive

## Code blocks

**Code blocks** allow to encapsulate **several** statements in a single group

* { and } contain code blocks
* the statements are execute together

first\_value <- 8  
second\_value <- 5  
if (first\_value > second\_value) {  
 cat("First is greater than second\n")   
 difference <- first\_value - second\_value  
 cat("Their difference is ", difference)  
}

## First is greater than second  
## Their difference is 3

## Loops

Loops are a fundamental component of (procedural) programming.

There are two main types of loops:

* **conditional** loops are executed as long as a defined condition holds true
  + construct while
  + construct repeat
* **deterministic** loops are executed a pre-determined number of times
  + construct for

## While

The *while* construct can be defined using the while reserved word, followed by the conditional statement between simple brackets, and a code block. The instructions in the code block are re-executed as long as the result of the evaluation of the conditional statement is TRUE.

current\_value <- 0  
while (current\_value < 3) {  
 cat("Current value is", current\_value, "\n")  
 current\_value <- current\_value + 1  
}

## Current value is 0   
## Current value is 1   
## Current value is 2

## For

The *for* construct can be defined using the for reserved word, followed by the definition of an **iterator**. The iterator is a variable which is temporarily assigned with the current element of a vector, as the construct iterates through all elements of the vector. This definition is followed by a code block, whose instructions are re-executed once for each element of the vector.

cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")  
for (city in cities) {  
 cat("Do you live in", city, "?\n")  
}

## Do you live in Derby ?  
## Do you live in Leicester ?  
## Do you live in Lincoln ?  
## Do you live in Nottingham ?

## For

It is common practice to create a vector of integers on the spot in order to execute a certain sequence of steps a pre-defined number of times.

for (i in 1:3) {  
 cat("This is exectuion number", i, ":\n")  
 cat(" See you later!\n")  
}

## This is exectuion number 1 :  
## See you later!  
## This is exectuion number 2 :  
## See you later!  
## This is exectuion number 3 :  
## See you later!

## Loops with conditional statements

3:0

## [1] 3 2 1 0

#Example: countdown!  
for (i in 3:0) {  
 if (i == 0) {  
 cat("Go!\n")  
 } else {  
 cat(i, "\n")  
 }  
}

## 3   
## 2   
## 1   
## Go!

## Summary

Control structures

* Conditional statements
* Loops

**Next**: Functions

* Defining functions
* Scope of a variable

# Functions

## Summary

**Prev**:Control structures

* Conditional statements
* Loops

**Now**: Functions

* Defining functions
* Scope of a variable

## Defining functions

A function can be defined

* using an **identifier** (e.g., add\_one)
* on the left of an **assignment operator** <-
* followed by the corpus of the function

add\_one <- function (input\_value) {  
 output\_value <- input\_value + 1  
 output\_value  
 }

## Defining functions

The corpus

* starts with the reserved word function
* followed by the **parameter(s)** (e.g., input\_value) between simple brackets
* and the instruction(s) to be executed in a code block
* the value of the last statement is returned as output

add\_one <- function (input\_value) {  
 output\_value <- input\_value + 1  
 output\_value  
 }

## Defining functions

After being defined

* a function can be invoked by specifying
* the **identifier**
* the necessary **parameter(s)**

add\_one(3)

## [1] 4

add\_one(1024)

## [1] 1025

## More parameters

* A function can be defined as having two or more **parameters**
* by specifying more than one parameter name (separated by **commas**) in the function definition
* A function always take as input as many values as the number of parameters specified in the definition
* otherwise an error is generated

area\_rectangle <- function (hight, width) {  
 area <- hight \* width  
 area  
}  
area\_rectangle(3, 2)

## [1] 6

## Functions and control structures

Functions can contain both loops and conditional statements

factorial <- function (input\_value) {  
 result <- 1  
 for (i in 1:input\_value) {  
 cat("current:", result, " | i:", i, "\n")  
 result <- result \* i  
 }  
 result  
}  
factorial(3)

## current: 1 | i: 1   
## current: 1 | i: 2   
## current: 2 | i: 3

## [1] 6

## Scope

The **scope of a variable** is the part of code in which the variable is `visible'' In R, variables have a \*\*hierarchical\*\* scope: - a variable defined in a script can be used referred to from within a definition of a function in the same script - a variable defined within a definition of a function will \*\*not\*\* be referable from outside the definition - scope does \*\*not\*\* apply toifor loop constructs ## Example In the case below -x\_valueis \*\*global\*\* to the functiontimes\_x-new\_valueandinput\_valueare \*\*local\*\* to the functiontimes\_x- referring tonew\_valueorinput\_valuefrom outside the definition oftimes\_x` would result in an error

x\_value <- 10  
times\_x <- function (input\_value) {  
 new\_value <- input\_value \* x\_value  
 new\_value  
}  
times\_x(2)

## [1] 20

## Summary

Functions

* Defining functions
* Scope of a variable

**Next**: Practical session

* Conditional statements
* Loops
  + While
  + For
* Functions
  + Loading functions from scripts
* Debugging

# Data frames

## Recap

**Prev**: R programming

* 111 Lecture: Data types
* 112 Lecture: Control structures
* 113 Lecture: Functions
* 114 Practical session

**Now**: Data Frames

* Data Frames
* Tibbles

## Lists and named lists

**List**

* can contain elements of different types
* whereas elements of vectors are all of the same type
* in **named lists**, each element has a name
* elements can be selected using the operator $

employee <- list(employee\_name = "Stef", start\_year = 2015)  
employee[[1]]

## [1] "Stef"

employee$employee\_name

## [1] "Stef"

## Data Frames

A **data frame** is equivalent to a *named list* where all elements are *vectors of the same length*.

employees <- data.frame(  
 EmployeeName = c("Maria", "Pete", "Sarah"),  
 Age = c(47, 34, 32),  
 Role = c("Professor", "Researcher", "Researcher"))  
employees

## EmployeeName Age Role  
## 1 Maria 47 Professor  
## 2 Pete 34 Researcher  
## 3 Sarah 32 Researcher

Data frames are the most common way to represent tabular data in R. Matrices and lists can be converted to data frames.

## Selection

Selection is similar to vectors and lists.

employees[1, 1] # value selection

## [1] Maria  
## Levels: Maria Pete Sarah

employees[1, ] # row selection

## EmployeeName Age Role  
## 1 Maria 47 Professor

employees[, 1] # column selection

## [1] Maria Pete Sarah  
## Levels: Maria Pete Sarah

## Selection

Selection is similar to vectors and lists.

employees$EmployeeName # column selection, as for named lists

## [1] Maria Pete Sarah  
## Levels: Maria Pete Sarah

employees$EmployeeName[1]

## [1] Maria  
## Levels: Maria Pete Sarah

## Table manipulation

* Values can be assigned to cells
* using any selection method
* and the assignment operator <-
* New columns can be defined
* assigning a vector to a new name

employees$Age[3] <- 33   
employees$Place <- c("Leicester", "Leicester","Leicester")   
employees

## EmployeeName Age Role Place  
## 1 Maria 47 Professor Leicester  
## 2 Pete 34 Researcher Leicester  
## 3 Sarah 33 Researcher Leicester

## Column processing

Operations can be performed on columns as they where vectors

10 - c(1, 2, 3)

## [1] 9 8 7

# Use Sys.Date to retrieve the current year  
current\_year <- as.integer(format(Sys.Date(), "%Y"))  
# Calculate employee year of birth  
employees$Year\_of\_birth <- current\_year - employees$Age  
employees

## EmployeeName Age Role Place Year\_of\_birth  
## 1 Maria 47 Professor Leicester 1973  
## 2 Pete 34 Researcher Leicester 1986  
## 3 Sarah 33 Researcher Leicester 1987

## tibble

A [tibble](https://tibble.tidyverse.org/) is a modern reimagining of the data.frame within tidyverse

* they do less
  + don’t change column names or types
  + don’t do partial matching
* complain more
  + e.g. when referring to a column that does not exist

That forces you to confront problems earlier, typically leading to cleaner, more expressive code.

## Summary

Data Frames

* Data Frames
* Tibbles

**Next**: Data selection and filtering

* dplyr
* dplyr::select
* dplyr::filter

# Selection and filtering

all R code parts are removed, because of an unsolved error message

## Recap

**Prev**: Data Frames

* Data Frames
* Tibbles

**Now**: Data selection and filtering

* dplyr
* dplyr::select
* dplyr::filter

## dplyr

The dplyr (pronounced *dee-ply-er*) library is part of tidyverse and it offers a grammar for data manipulation

* select: select specific columns
* filter: select specific rows
* arrange: arrange rows in a particular order
* summarise: calculate aggregated values (e.g., mean, max, etc)
* group\_by: group data based on common column values
* mutate: add columns
* join: merge tables (tibbles or data.frames)

## Example dataset

## Selecting table columns

## dplyr::select

select can be used to specify which columns to retain

## dplyr::select

... or whichones to drop, using - in front of the column name

## Logical filtering

Conditional statements can be used to filter a vector

* i.e. to retain only certain values
* where the specified value is TRUE

## Conditional filtering

As a conditional expression results in a logic vector...

... conditional expressions can be used for filtering

## Filtering data frames

The same approach can be applied to **data frames** and **tibbles**

## dplyr::filter

## Select and filter

## Summary

Data selection and filtering

* dplyr
* dplyr::select
* dplyr::filter

# Read and write data

all R code parts are removed, because of an unsolved error message

## Summary

Tidy-up your data

* Wide and long data
* Re-shape data
* Handle missing values

**Next**: Read and write data

* file formats
* read
* write

## Text file formats

A series of formats based on plain-text files

For instance

* comma-separated values files .csv
* semi-colon-separated values files .csv
* tab-separated values files .tsv
* other formats using custom delimiters
* fix-width files .fwf

## Comma Separated Values

The file 2011\_OAC\_supgrp\_Leicester.csv contains

* one row for each [Output Area (OA)](https://www.ons.gov.uk/methodology/geography/ukgeographies/censusgeography) in Leicester
* [Lower-Super Output Area (LSOA)](https://www.ons.gov.uk/methodology/geography/ukgeographies/censusgeography) containing the OA
* code and name of the supergroup assigned to the OA by the [2011 Output Area Classification](http://geogale.github.io/2011OAC/)
* total population of the OA

Extract showing only the first few rows

OA11CD,LSOA11CD,supgrpcode,supgrpname,Total\_Population  
E00069517,E01013785,6,Suburbanites,313  
E00069514,E01013784,2,Cosmopolitans,323  
E00169516,E01013713,4,Multicultural Metropolitans,341  
E00169048,E01032862,4,Multicultural Metropolitans,345

## readr

The [readr](https://readr.tidyverse.org/) (pronounced *read-er*) library is part of [tidyverse](https://www.tidyverse.org/)

Provides functions to read and write text files

* readr::read\_csv: comma-separated files .csv
* readr::read\_csv2: semi-colon-separated files .csv
* readr::read\_tsv: tab-separated files .tsv
* readr::read\_fwf: fix-width files .fwf
* readr::read\_delim: files using a custom delimiter

and their *write* counterpart, such as

* readr::write\_csv: comma-separated files .csv

## readr::read\_csv

The readr::read\_csv function of the [readr](https://readr.tidyverse.org/index.html) library reads a *csv* file from the path provided as the first argument

## Read options

Read functions provide options about how to interpret a file contents

* For instance, readr::read\_csv
* col\_names:
  + TRUE or FALSE whether top row is column names
  + or a vector of column names
* col\_types:
  + a cols() specification or a string
* skip: lines to skip before reading data
* n\_max: max number of record to read

## Column specifications

* col\_logical() or l as logic values
* col\_integer() or i as integer
* col\_double() or d as numeric (double)
* col\_character() or c as character
* col\_factor(levels, ordered) or f as factor
* col\_date(format = "") or D as data type
* col\_time(format = "") or t as time type
* col\_datetime(format = "") or T as datetime
* col\_number() or n as numeric (dropping marks)
* col\_skip() or \_ or - don’t import
* col\_guess() or ? use best type based on the input

## readr::read\_csv

Using readr::read\_csv as in the previous example with no further options will generate the following warning

## readr::read\_csv

## readr::read\_csv

## readr::write\_csv

The function write\_csv can be used to save a dataset to csv

Example:

1. **read** the 2011 OAC dataset
2. **select** a few columns
3. **filter** only those OA in the supergroup *Suburbanites* (code 6)
4. **write** the results to a file named *2011\_OAC\_supgrp\_Leicester\_supgrp6.csv*

## readr::write\_tsv

## Other data imports

[Tidyverse](https://www.tidyverse.org/) also imports [other packages for reading data](https://www.tidyverse.org/packages/#import)

* Tabular formats
* [readxl](https://readxl.tidyverse.org/) for Excel (.xls and .xlsx)
* [haven](https://haven.tidyverse.org/) for SPSS, Stata, and SAS data.
* Databases
* [DBI](https://cran.r-project.org/web/packages/DBI/) for relational databases
* NoSQL
* [jsonlite](https://cran.r-project.org/web/packages/jsonlite/) for JSON
* [xml2](https://cran.r-project.org/web/packages/xml2/) for XML
* Web
* [httr](https://cran.r-project.org/web/packages/httr/) for web APIs

## Summary

Read and write data

* file formats
* read
* write

**Next**: Practical session

* Read and write data
* Tidy data
* Join operations

# Data visualisation

all R code parts are removed, because of an unsolved error message

## Recap

**Prev**: Reproducibility

* 221 Reproducibility
* 222 R and Markdown
* 223 Git
* 224 Practical session

**Now**: Data visualisation

* Grammar of graphics
* ggplot2

## Visual variables

A **visual variable** is an aspect of a **mark** that can be controlled to change its appearance.

Visual variables include:

* Size
* Shape
* Orientation
* Colour (hue)
* Colour value (brightness)
* Texture
* Position (2 dimensions)

## Grammar of graphics

Grammars provide rules for languages

*"The grammar of graphics takes us beyond a limited set of charts (words) to an almost unlimited world of graphical forms (statements)"* (Wilkinson, 2005)

Statistical graphic specifications are expressed in six statements:

1. **Data** manipulation
2. **Variable** transformations (e.g., rank),
3. **Scale** transformations (e.g., log),
4. **Coordinate system** transformations (e.g., polar),
5. **Element**: mark (e.g., points) and visual variables (e.g., color)
6. **Guides** (axes, legends, etc.).

## ggplot2

The ggplot2 library offers a series of functions for creating graphics **declaratively**, based on the Grammar of Graphics.

To create a graph in ggplot2:

* provide the data
* specify elements
  + which visual variables (aes)
  + which marks (e.g., geom\_point)
* apply transformations
* guides

## Boxplots

* x categorical variable
* y variable to plot
* geom\_boxplot

## Jittered points

* x categorical variable
* y variable to plot
* geom\_jitter

## Violin plot

* x categorical variable
* y variable to plot
* geom\_violin

## Violin plot

## Lines

* x e.g., a temporal variable
* y variable to plot
* geom\_line

## Lines

## Scatterplots

* x and y variable to plot
* geom\_point

## Overlapping points

* x and y variable to plot
* geom\_count counts overlapping points and maps the count to size

## Overlapping points

## Bin counts

* x and y variable to plot
* geom\_bin2d

## Bin counts

## Summary

Data visualisation

* Grammar of graphics
* ggplot2

**Next**: Descriptive statistics

* stat.desc
* dplyr::across

# References