MY470 - Week 8: The R Language

Friedrich Geiecke

16 November 2020

Outline

1. Introduction
2. Fundamentals and data structures
3. Control flow
4. Functions
5. Reading in data and plotting
6. Data science workflows with R today

Outline

1. **Introduction**
2. Fundamentals and data structures
3. Control flow
4. Functions
5. Reading in data and plotting
6. Data science workflows with R today

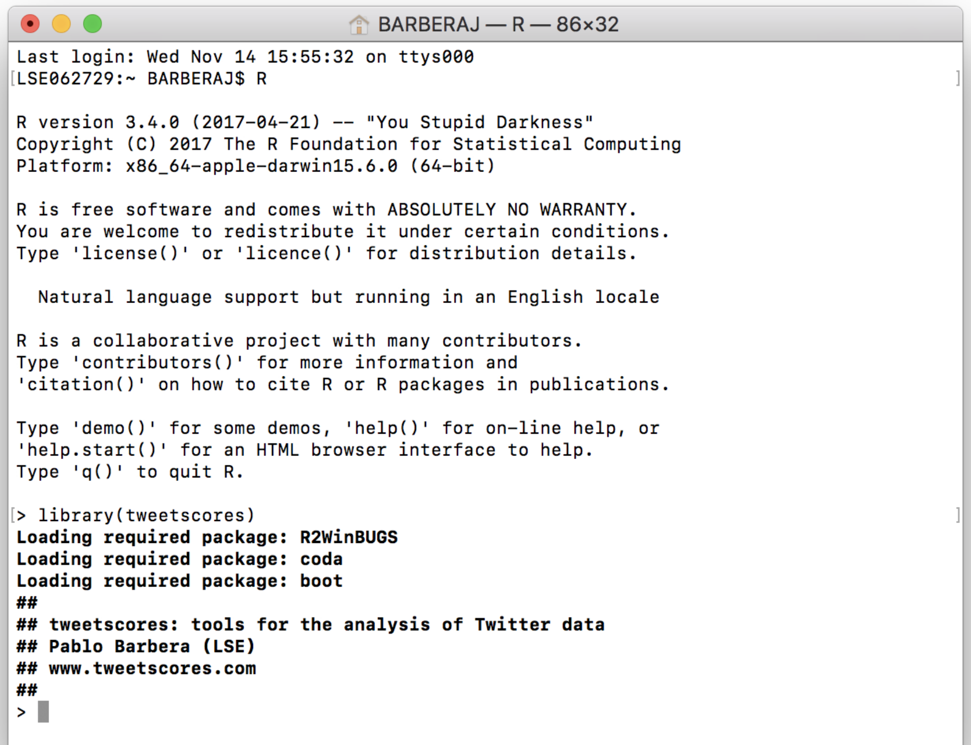
R in a nutshell

R is a versatile, open source programming language that is useful for statistics, data science, and beyond. Inspired by the programming language S

- Open source software under GPL
- Superior (if not just comparable) to commercial alternatives. R has over 10,000 user contributed packages (CRAN) and many more elsewhere
- Available on all platforms
- Not just for statistics, but also general purpose programming, graphics, network analysis, machine learning, web scraping…
- Object oriented, but at its core a functional language
- Can be run interactively or in batch mode

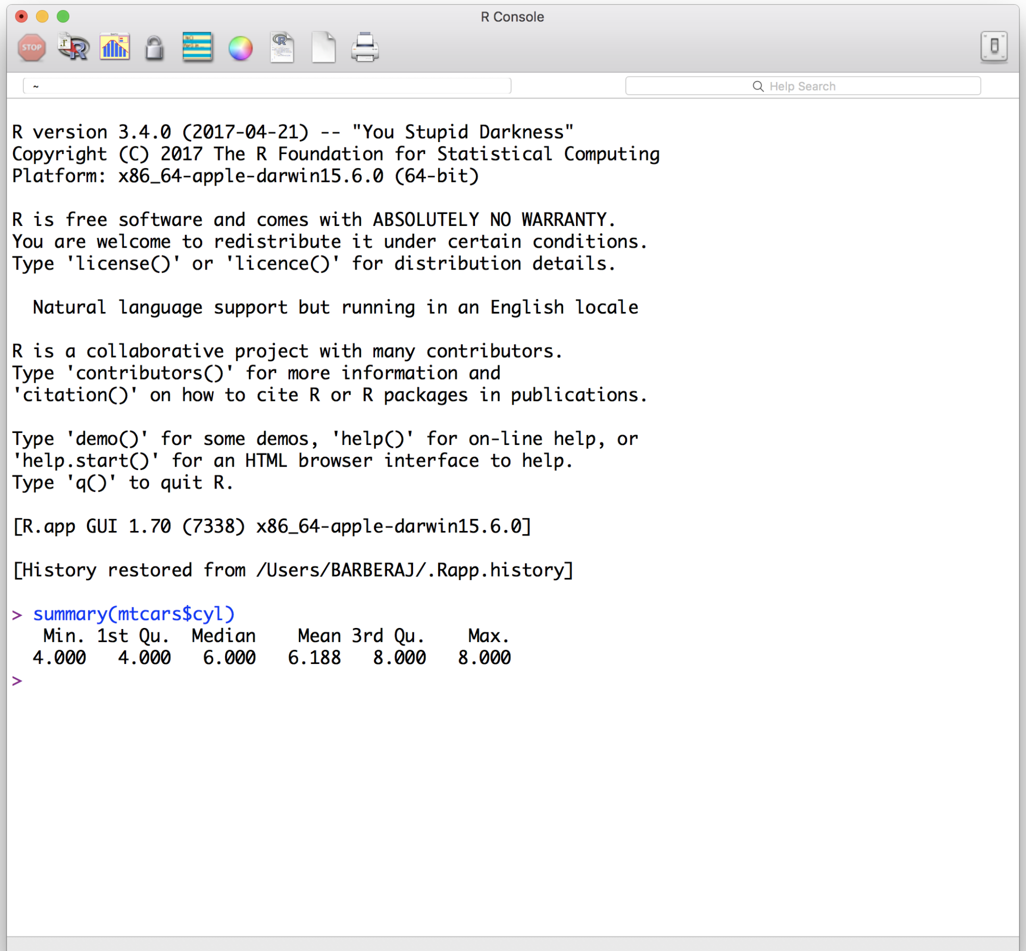
Running R interactively: Terminal

- For example, type `R` into terminal. The window that appears is called the R console. Any command you type into this prompt is interpreted by the R kernel



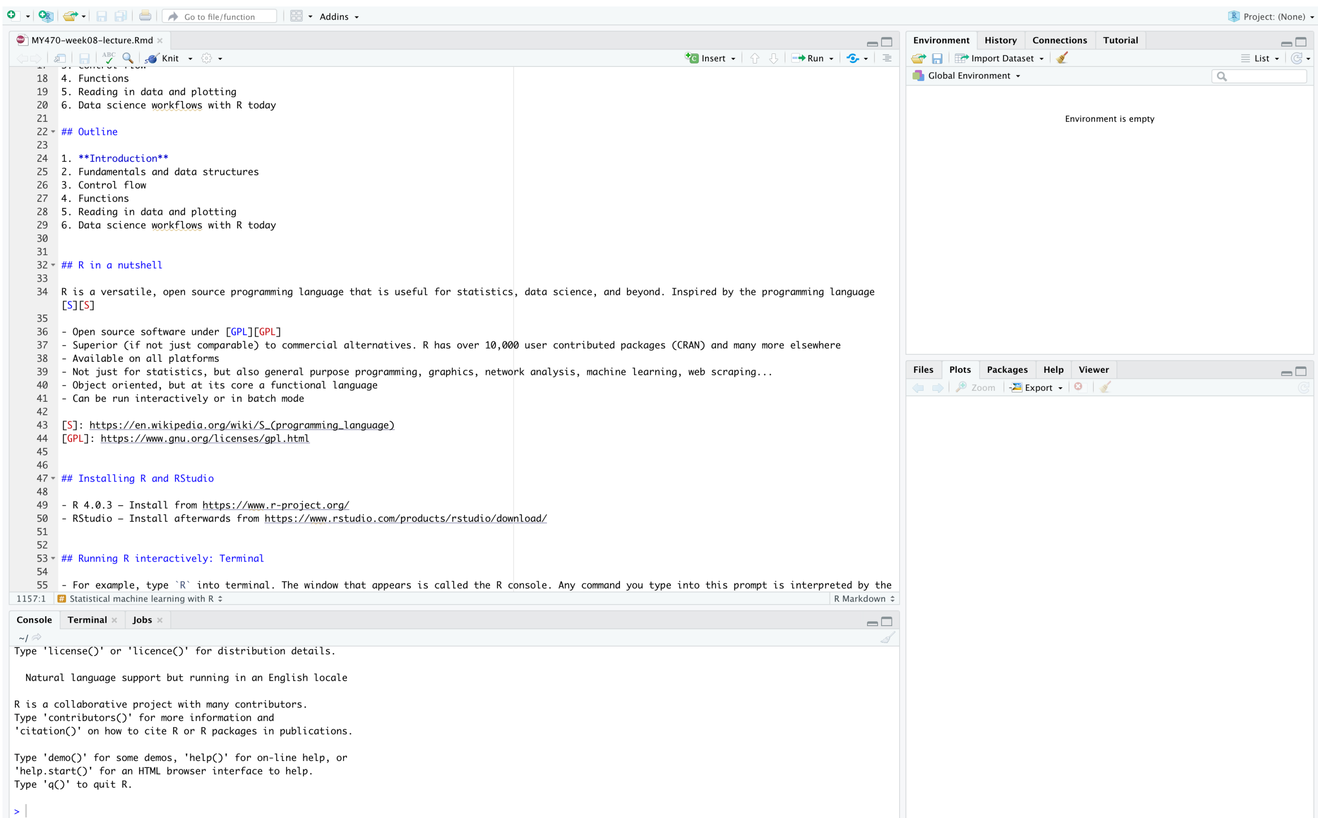
Running R interactively: R GUI console

- The plain R programme after installation (also has a text editor window)



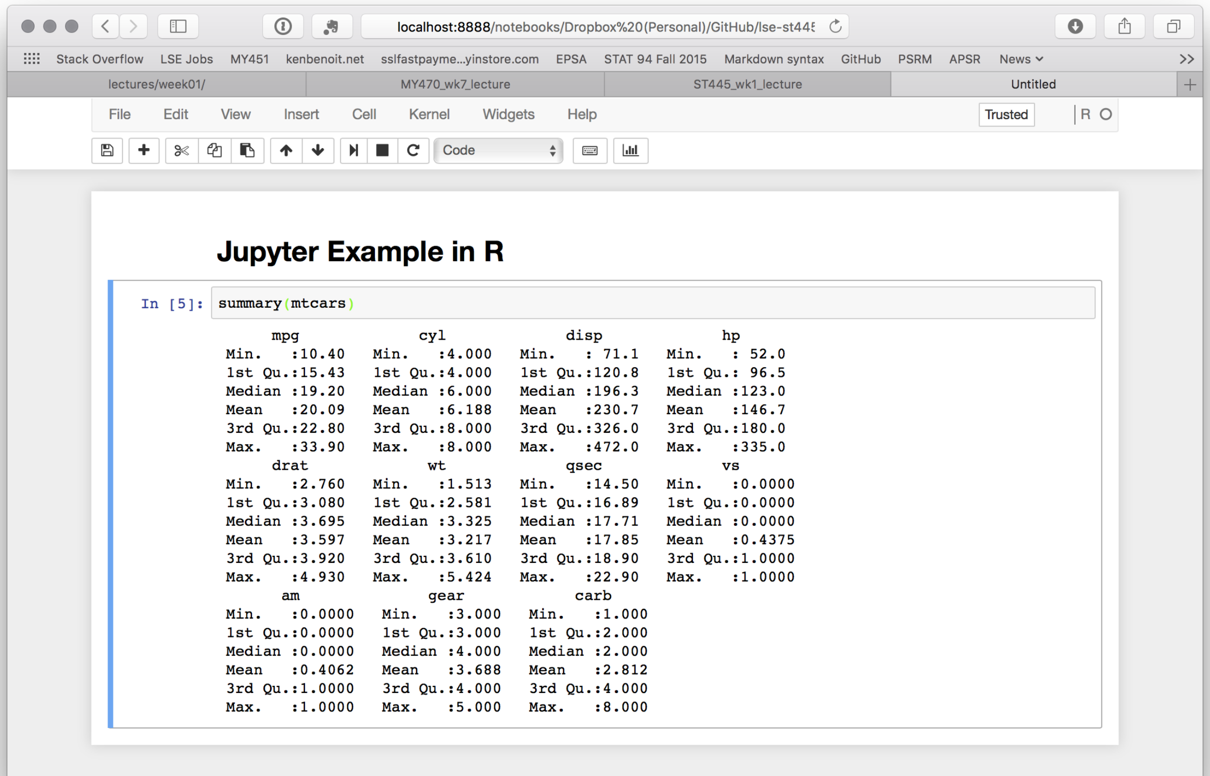
Running R interactively: RStudio (used in this course)

- RStudio is an IDE (Integrated Development Environment) that makes many things easier



Running R interactively: Jupyter

- You need to change the kernel to R
- R Markdown is the more natural option in R (great integration with RStudio)



Running R in batch mode

You can also run one or more R scripts in batch mode, e.g.

```
R CMD BATCH script_1.R script_2.R
```

Installing R and RStudio

- R 4.0.3 – Install from https://www.r-project.org/
- RStudio – Install afterwards from https://www.rstudio.com/products/rstudio/download/

Installing and managing packages in R

- R has over 10,000 user contributed packages on CRAN (The Comprehensive R Archive Network) and many more elsewhere
- CRAN "is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R.", see CRAN
- `install.packages("package-name")` will download a package from one of the CRAN mirrors. Similar to `pip install` and `conda install`, however, run from R directly
- If you have not set a preferred CRAN mirror in your `options()`, then a menu will pop up asking you to choose a location
- At the beginning of your script you load packages with `library("package-name")`. Similarly to `import`
- Use `old.packages()` to list all your locally installed packages that are now out of date. `update.packages()` will update all packages interactively. This can take a while if you haven't done it recently. To update everything without any user intervention, use the `ask = FALSE` argument

```
update.packages(ask = FALSE)
```

Outline

1. Introduction
2. **Fundamentals and data structures**
3. Control flow
4. Functions
5. Reading in data and plotting
6. Data science workflows with R today

2. Fundamentals and data structures

- 2.1 Basic operations in R
- 2.2 Objects in R
- 2.3 Key data structures
    - 2.3.1 Atomic vectors
    - 2.3.2 Lists
    - 2.3.3 Matrices and data frames

2.1 Basic operations in R

Basic operations

```
1 + 1
```

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

```
5 - 3
```

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

```
6 / 2
```

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

```
4 * 4
```

```
## [1] 16
```

```
3 ^ 3
```

```
## [1] 27
```

Basic mathematical functions

```
log(<number>)
exp(<number>)
sqrt(<number>)
mean(<numbers>)
sum(<numbers>)
```

Basic logical operators

- `<` : less than
- `>` : greater than
- `==` : equal to (note, not `=`)
- `>=` : greater than or equal to
- `<=` : less than or equal to
- `!=` : not equal to
- `&` : and
- `|` : or
- `%in%`: in

2.2 Objects in R

Object assignment

- R assignment operator `<-`
- Assigns values on the right to objects on the left. Mostly similar to `=` but subtle differences. Use `<-` in R
- The `<-` notation also emphasises that `=` is not a mathematical equal sign when using it for assignments in programming: x = x + 1 ?

```
my_object <- 10
print(my_object)
```

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

```
my_other_object <- 4
print(my_object - my_other_object)
```

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

Mutable/immutable objects and memory allocations

- A way to think about mutable vs immutable object is whether they are copied to a new address in memory when modified or kept in their old one
- Unlike in Python, most objects in R are copied when modified (with some important exceptions) so can be called immutable in that sense
- An exception would for example be a vector that has only been assigned to a single name, it can be modified in place
- It can pay off to study these topics for performance of code regardless of the language. A very good summary for R, which is also the basis of this slide and the next two, can be found here: https://adv-r.hadley.nz/names-values.html

Mutable/immutable objects and memory allocations

```
library(pryr)
x <- c(3, 6, 9)
x
```

```
## [1] 3 6 9
```

```
y <- x
address(x)
```

```
## [1] "0x7fa321219de8"
```

```
address(y)
```

```
## [1] "0x7fa321219de8"
```

Mutable/immutable objects and memory allocations

```
x[2] <- 4
x
```

```
## [1] 3 4 9
```

```
address(x)
```

```
## [1] "0x7fa321c52458"
```

```
address(y)
```

```
## [1] "0x7fa321219de8"
```

- Can check this in Python as well via `id()`

```
# Mutable            # Immutable
a = [1,2,3]          b = 42
id(a)                id(b)
a[2] = 4             b += 1
id(a)                id(b)
```

Querying object attributes

```
y <- 10          # For example a numeric vector of length 1
class(y)         # Class of the object
```

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

```
typeof(y)        # R's internal type for storage
```

```
## [1] "double"
```

```
length(y)        # Length
```

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

```
attributes(y) # Metadata (matrices e.g. store their dimensions)
```

```
## NULL
```

```
names(y)         # Names
```

```
## NULL
```

```
dim(y)           # Dimensions
```

Viewing objects in your global environment and removing them

- List objects in your current environment

```
ls()
```

- Remove objects from your current environment

```
x <- 5
ls()
```

```
## [1] "my_object"       "my_other_object" "x"               "y"
```

```
rm(x)
ls()
```

```
## [1] "my_object"       "my_other_object" "y"
```

Remove all objects from your current environment

```
rm(list = ls())
```

- Notice that we have nested one function inside another here

Object oriented programming (OOP) in R

- "Everything that exists in R is an object" (John Chambers), but *object oriented* ***programming*** *(OOP)* is much less important in the daily use of R than functional programming (more on functions and functional programming later)
- Users mostly solve problems by decomposing them into functions rather than summarising them in classes
- OOP is also more challenging in R as there are multiple OOP systems called S3, R6, S4, etc.
- If you would like to learn about object oriented programming in R (e.g. to write packages), see the link

Reference: https://adv-r.hadley.nz/oo.html

2.3 Key data structures

Key data structures

Common data structures in R

Stores homogenous elements	Stores heterogenous elements
Atomic vector	List
Matrix	Data frame
Array	

http://adv-r.had.co.nz/Data-structures.html

More extensive lists e.g. here: https://cran.r-project.org/doc/manuals/r-release/R-lang.html

Comparison R and Python

Python class	Closest R class
bool	logical
int	numeric: integer
float	numeric: double
list	unnamed list
tuple	-
str	character
set	-
frozenset	-
dict	named list (named vector also has key-value structure but can only store one type)

2.3.1 (Atomic) vectors

Vectors in R

- Atomic vectors (this subsection)
- Lists (sometimes called recursive vectors)

Atomic vectors

- An (atomic) vector is a collection of entities which all share the same type
- Vectors are the most common and basic data structure in R

Six types (excluding the `raw` class for this lecture)

- character
- integer
- double
- logical
- complex

Integer and double vectors are called numeric vectors

https://r4ds.had.co.nz/vectors.html

Types

Example	Type
“a”, “swc”	character
2L (Must add a `L` at end to denote integer)	numeric: integer
2, 15.5	numeric: double
`TRUE`, `FALSE`	logical
1+4i	complex

Note: Complex numbers are hardly used in statistical analysis, but if you are interested in the topic, this video offers a great introduction

Special values

- Integers have one special value: NA
- Doubles have four: NA, NaN, Inf and -Inf

`Inf` is infinity. You can have either positive or negative infinity

```
1 / 0
```

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

`NaN` means Not a number. It is an undefined value

```
0 / 0
```

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

Examples

Use the `c()` function to concatenate observations into a vector

- Character vector

```
char_vec <- c("hello", "world")
print(char_vec)
```

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

- Numeric (integer) vector

```
num_int_vec <- c(5L, 3L, 2L) # note: c(5,3,2) would be stored as double
print(num_int_vec)
```

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

Examples

- Numeric (double) vector

```
num_double_vec <- c(5, 4, 100, 7.65)
print(num_double_vec)
```

```
## [1]   5.00   4.00 100.00   7.65
```

- Logical vector

```
logical_vec <- c(TRUE, FALSE, TRUE)
print(logical_vec)
```

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

Examples

- Complex vector

```
complex_vec <- c(18+2i , 5+4i)
print(complex_vec)
```

```
## [1] 18+2i  5+4i
```

- General note: The following two objects are identical in R, scalars are vectors of length one here!

```
identical(1.41, c(1.41))
```

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

Empty vectors

- You can create an empty vector with `vector()` (by default the mode is `logical`, but you can define different modes as shown in the examples below)
- It is more common to use direct constructors such as `character()`, `numeric()`, etc.

```
vector()
```

```
## logical(0)
```

```
vector(mode = "character", length = 10) # with a length and type
```

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

```
character(5) # character vector of length 5, also see numeric(5) and logical(5)
```

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

Adding elements to vectors

```r
z <- c("my470", "is")
z <- c(z, "fantastic")
z
```

```
## [1] "my470"     "is"        "fantastic"
```

You can also create vectors with sequences of numbers

```
series <- 1:10
series
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
series <- seq(1, 10, by = 0.1)
series
```

```
##  [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3  2.4
## [16]  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9
## [31]  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1  5.2  5.3  5.4
## [46]  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9
## [61]  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.0  8.1  8.2  8.3  8.4
## [76]  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3  9.4  9.5  9.6  9.7  9.8  9.9
## [91] 10.0
```

```
class(series)
```

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

General notes on indexing elements in R

- Many languages index from 0
- R indices start at 1
- R indices include the last element
- So `myvector[1:3]` selects elements 1, 2, and 3 in R
- In Python, `mylist[0:3]` selects elements 0, 1, and 2

Excursus: Implications for indexing characters in R

In Python:

```
letters = "abcdefghijklmnopqrztuv"
print letters[0:4]
```

- In R, however, even a single string is a character vector of length one
- Hence, we cannot index individual character elements of a string in R like this

```
firstletters <- "abcdefg"
firstletters[1:3]
```

```
## [1] "abcdefg" NA        NA
```

Excursus: Implications for indexing characters in strings

- Instead, use letters or words as individual elements in a vector
- For example, `letters` is a pre-defined character vector of length 26 in R

```
length(letters)
```

```
## [1] 26
```

```
letters[1:5]
```

```
## [1] "a" "b" "c" "d" "e"
```

- To determine length of a string, e.g. use `nchar`

```
length("London")
```

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

```
nchar("London")
```

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

Vector subsetting in R

- To subset a `vector`, use square parenthesis to index the elements you would like via `object[index]`.
- Numerical subsetting

```
num_double_vec[3]
```

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

```
num_double_vec[1:2]
```

```
## [1] 5 4
```

Vector subsetting in R

- Subsetting with names

```
x <- c(1,2,4)
names(x) <- c("element1", "element2", "element3")
x["element1"]
```

```
## element1
##        1
```

Caveat: Although this looks somewhat like a Python dictionary, recall that vectors can only store single types

Vector subsetting in R

- Logical subsetting

```
char_vec <- c("hello", "world")
char_vec
```

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

```
logical_vec <- c(TRUE, FALSE)
logical_vec
```

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

```
char_vec[logical_vec]
```

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

Vector operations

- In R, mathematical operations on vectors typically occur **element-wise** (unless you e.g. specify a dot-product with `%*%`)

```
fib <- c(1, 1, 2, 3, 5, 8, 13, 21)
fib[1:7] + fib[2:8]
```

```
## [1]  2  3  5  8 13 21 34
```

Vector operations

- It is also possible to perform logical operations on vectors

```
fib <- c(1, 1, 2, 3, 5, 8, 13, 21)
fib_greater_five <- fib > 5
print(fib_greater_five)
```

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

Recycling

- R usually operates on vectors of the same length
- If it encounters two vectors of different lengths in a binary operation, it *replicates* (recycles) the smaller vector until it is of the same length as the longer vector
- Afterwards it does the operation
- Related to “broadcasting” in Python/numpy
- Often helpful, but can lead to very hard to find bugs

Recycling

- If the recycled smaller vector has to be "chopped off" to make it the length of the longer vector, you will get a warning, but it will still return a result

```
x <- c(1, 2, 3)
y <- c(5, 10)
x * y
```

```
## Warning in x * y: longer object length is not a multiple of shorter object
## length
```

```
## [1]  5 20 15
```

Recycling

- Other examples or recycling

```
x <- 1:20
```

```
x * c(1, 0)    # turns the even numbers to 0
```

```
##  [1]  1  0  3  0  5  0  7  0  9  0 11  0 13  0 15  0 17  0 19  0
```

```
x * c(0, 0, 1) # turns non-multiples of 3 to 0
```

```
## Warning in x * c(0, 0, 1): longer object length is not a multiple of shorter
## object length
```

```
##  [1]  0  0  3  0  0  6  0  0  9  0  0 12  0  0 15  0  0 18  0  0
```

```
x < ((1:4) ^ 2)  # recycling c(1, 4, 9, 16) for logical operation
```

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

Factors: Vectors with labels

- A factor is a special kind of vector
- It is similar to a character vector, but here each unique element is also associated with a numerical value which e.g. allows to better process categorical data
- A factor vector can only contain predefined values

```
factor_vec <- as.factor(c("a", "b","c","a","b","c"))
factor_vec
```

```
## [1] a b c a b c
## Levels: a b c
```

```
as.numeric(factor_vec) # how it is processed in the background
```

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

- Note: Statistical models can require categorical variables to be stored as factors

2.3.2 Lists

Lists

- Lists are the other type of vector in R, they are sometimes referred to as recursive vectors as lists can also contain lists themselves
- In general, however, atomic vectors are commonly called `vectors` in R and lists are called `lists`
- A `list` is a collection of any set of object types
- Closest to the dictionary’s key-value structure in Python if elements in the list are named

Lists

A `list` is a collection of any set of object types

```
my_list <- list(something = num_double_vec,
                another_thing = matrix(data = 1:9, nrow = 3, ncol = 3),
                something_else = "my470")
my_list
```

```
## $something
## [1]   5.00   4.00 100.00   7.65
##
## $another_thing
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $something_else
## [1] "my470"
```

How to index list elements in R

- Using [

```
my_list["something_else"]
```

```
## $something_else
## [1] "my470"
```

```
my_list[3]
```

```
## $something_else
## [1] "my470"
```

```
class(my_list[3])
```

```
## [1] "list"
```

How to index list elements in R

- Using [[

```
my_list[["something"]]
```

```
## [1]   5.00   4.00 100.00   7.65
```

```
my_list[[1]]
```

```
## [1]   5.00   4.00 100.00   7.65
```

```
class(my_list[[1]])
```

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

How to index list elements in R

- **Using $**

```
my_list$another_thing
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

(does not allow multiple elements to be indexed in one command)

2.3.3 Matrices and data frames

Matrices

- Next, we will discuss tabular data in more detail
- A `matrix` arranges data from a vector into a tabular form, **all elements have to be of the same type**
- `Arrays` have more than 2 dimensions

```
my_matrix <- matrix(data = 1:100, nrow = 10, ncol = 10)
my_matrix
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   43   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   56   66   76   86    96
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   18   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
```

Data frames

A `data.frame`, in contrast, is a matrix-like R object in which the **columns can be of different types**

```
my_data_frame <- data.frame(numbers = num_double_vec,
                            words = char_vec,
                            logical = logical_vec)
my_data_frame
```

```
##   numbers words logical
## 1    5.00 hello    TRUE
## 2    4.00 world   FALSE
## 3  100.00 hello    TRUE
## 4    7.65 world   FALSE
```

Beware: `stringsAsFactors` might be TRUE by default

```
logical_vec <- c(TRUE, FALSE, FALSE, TRUE, FALSE)
char_vec <- c("my457", "my459", "my470", "my472", "my474")
my_data_frame <- data.frame(numbers = c(5, 4, 2, 100, 7.65),
                            courses = char_vec,
                            logical = logical_vec,
                            stringsAsFactors = TRUE)
str(my_data_frame)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ numbers: num  5 4 2 100 7.65
##  $ courses: Factor w/ 5 levels "my457","my459",..: 1 2 3 4 5
##  $ logical: logi  TRUE FALSE FALSE TRUE FALSE
```

How to correct this

```
my_data_frame <- data.frame(numbers = c(5, 4, 2, 100, 7.65),
                            courses = char_vec,
                            logical = logical_vec,
                            stringsAsFactors = FALSE)
str(my_data_frame)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ numbers: num  5 4 2 100 7.65
##  $ courses: chr  "my457" "my459" "my470" "my472" ...
##  $ logical: logi  TRUE FALSE FALSE TRUE FALSE
```

Matrix and data frame subsetting

- You can subset a `matrix` or `data.frame` with integers referring to rows and columns

```
my_matrix[2, 2]
```

```
## [1] 12
```

```
my_matrix[2:3, 2:3]
```

```
##      [,1] [,2]
## [1,]   12   22
## [2,]   13   23
```

```
my_data_frame[,1]
```

```
## [1]   5.00   4.00   2.00 100.00   7.65
```

Matrix and data frame subsetting

- You can also access rows and columns with names

```
# Adding some column names to the matrix
colnames(my_matrix) = letters[1:10]

# Works for matrices and data frames
my_matrix[,"e"]
```

```
##  [1] 41 42 43 44 45 46 47 48 49 50
```

```
my_data_frame[,"numbers"]
```

```
## [1]   5.00   4.00   2.00 100.00   7.65
```

```
my_data_frame[,c("numbers", "courses")]
```

```
##   numbers courses
## 1    5.00   my457
## 2    4.00   my459
## 3    2.00   my470
## 4  100.00   my472
## 5    7.65   my474
```

```
# Works only with data frame columns
my_data_frame$numbers
```

```
## [1]   5.00   4.00   2.00 100.00   7.65
```

Matrix and data frame subsetting

- Dropping rows or columns can be done using the `-` operator and integers (in combination with the `c` function if multiple rows are dropped)

```
my_matrix[-4, -5]
```

```
##        a  b  c  d  f  g  h  i   j
##  [1,]  1 11 21 31 51 61 71 81  91
##  [2,]  2 12 22 32 52 62 72 82  92
##  [3,]  3 13 23 33 53 63 73 83  93
##  [4,]  5 15 25 35 55 65 75 85  95
##  [5,]  6 16 26 36 56 66 76 86  96
##  [6,]  7 17 27 37 57 67 77 87  97
##  [7,]  8 18 28 38 58 68 78 88  98
##  [8,]  9 19 29 39 59 69 79 89  99
##  [9,] 10 20 30 40 60 70 80 90 100
```

```
my_matrix[-c(2:8), -c(2:8)]
```

```
##       a  i   j
## [1,]  1 81  91
## [2,]  9 89  99
## [3,] 10 90 100
```

- `2:8` creates a vector of the integers 2, … , 8 and the `-` operator negates these. We wrap the vector in the `c` function so that `-` applies to each element, and not just the first

Outline

1. Introduction
2. Fundamentals and data structures
3. **Control flow**
4. Functions
5. Reading in data and plotting
6. Data science workflows with R today

3.1 Conditionals

If-else statements

- Using the logical operations discussed before, R has the usual if, if-else, and else conditions
- Contrary to Python, indentation is optional (but advised for readability), and brackets separate parts of the statements

If-else statements

```r
x <- 3
if (x > 4) {
  print(24)
} else {
  print(17)
  }
```

```
## [1] 17
```

With an additional else if part

```r
x <- 2
y <- 3
if (x < y) {
    print(24)
  } else if (x > y) {
    print(18)
  } else {
    print(17)
  }
```

```
## [1] 24
```

3.2 Loops

For loops

- Like conditionals, the different parts of loops are distinguished via brackets rather than mandatory indentation

```
for (i in 1:10) {
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

For loops can also iterate over character vectors

```
character_vector <- c("hello", "world", "in", "a", "for", "loop")
for (text in character_vector) {
    print(text)
}
```

```
## [1] "hello"
## [1] "world"
## [1] "in"
## [1] "a"
## [1] "for"
## [1] "loop"
```

While loops

```
x <- 1
while (x < 11) {
    print(x)
    x <- x + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

Improving efficiency

- Using vectorised operations where possible instead of loops can immensely speed up your code

```
# For example:
x <- 1:10000
y <- 1:10000
z <- numeric(10000)
for (i in 1:10000) {

  z[i] <- x[i]*y[i]

}
# vs
z <- x*y


# Or:
x <- 1:10000
y <- 1:10000
z <- 0
for (i in 1:10000) {

  z <- z + x[i]*y[i]

}
# vs:
z <- x%*%y
```

- Same considerations apply to vectorised operations in `numpy`
- For an in-depth discussion of measuring and improving performance in R: https://adv-r.hadley.nz/perf-measure.html and https://adv-r.hadley.nz/perf-improve.html

Outline

1. Introduction
2. Fundamentals and data structures
3. Control flow
4. **Functions**
5. Reading in data and plotting
6. Data science workflows with R today

Functions

- Functions all have the same basic structure

```
function_name(argument_one, argument_two, ...)
```

Where

- `function_name` is the name of the function
- `argument_one` is the first argument passed to the function
- `argument_two` is the second argument passed to the function
- and so on

Using function arguments

- When a function argument is not assigned a *default value*, then it is mandatory to be specified by the user
- Default arguments can be overridden if supplied
- It is not necessary to specify the names of the arguments, although it is best to do so except for the first or possibly second argument

Function example

- Let's consider the `mean()` function. This function takes two (main) arguments whereas the second one is set to `FALSE` by default

```
mean(x, na.rm = FALSE)
```

- Where `x` is a numeric vector, and `na.rm` is a logical value that indicates whether we'd like to remove missing values (`NA`).

```
vec <- c(1, 2, 3, NA, 5)
mean(x = vec, na.rm = TRUE)
```

```
## [1] 2.75
```

User defined functions

- Just like in Python and other programming languages, it is key to create own functions for a modular programme
- We can define a function as follows

```
my_addition_function <- function(a = 10, b) {
    return(a + b)
}
```

```
my_addition_function(a = 5, b = 50)
```

```
## [1] 55
```

```
my_addition_function(3, 4)
```

```
## [1] 7
```

```
my_addition_function(b = 100)
```

```
## [1] 110
```

Variables in functions have local scope

```
my_demo_function <- function(a) {
    a <- a * 2
    return(a)
}

a <- 1
my_demo_function(a = 20)
```

```
## [1] 40
```

```
a
```

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

Pipe operator

- Very often used in R code today
- The pipe operator %>% simply indicates that the previous object is used as the first argument in the subsequent function

```
library(tidyverse) # pipe operators are originally from the `magrittr` package by Stefan Milton Bache
x <- c(1,2,3,4,15)
# Equivalent:
mean(x)
```

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

```
x %>% mean()
```

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

```
# Useful for chains of computations
x <- c("1", "2")
x %>%
  as.numeric() %>%
  mean() %>%
  sqrt()
```

```
## [1] 1.224745
```

```
# Easier to read than the equivalent nested functions
sqrt(mean(as.numeric(x)))
```

Loops revisited: Apply functions

- Another very frequently used approach in R that reflects its focus on functions is to replace loops with apply
- It applies a function to every column, row, element of a vector, list, etc.
- Apply in Python for example in `pandas` with `df.apply()` and `df.map()`
- The following function avoids having to write a loop over all columns and determining the maximum value in each of them

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
apply(X = x, MARGIN = 2, FUN = max)
```

```
## [1] 3 6 9
```

- Other options
  - `sapply` to apply function to every element of a vector
  - `lapply` to apply function to every element of list

Functional programming and R

- R has plenty of object orientation and classes, but at its core it is more of a functional programming language

Two stylised features of functional programming

1. First-class functions, i.e. functions that behave like any other data structure. In R, this means that you can do many of the things with a function that you can do with a vector: You can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

2. "Pure" functions: The output only depends on the inputs, i.e. if you call it again with the same inputs, you get the same outputs. The function also has no side-effects, like changing the value of a global variable, writing to disk, or displaying to the screen. So e.g. `y <- 4; my_function <- function(x) {return(y + x)}` is not a pure function

Source: https://adv-r.hadley.nz/fp.html

Functional programming and R

- Of course not all functions in R always return the same output with the same inputs, e.g. runif() depends on the pseudo-random number seed, and write.csv() also writes output to disk
- Furthermore, Python also has features of both object oriented and functional programming
- Yet, the number of pure functions is arguably higher in R than in some other programming languages

Python and R

- Python, following more the OOP approach, has many functions/methods and attributes attached to objects (recall week 5 on classes)
- For examples, consider R vs `pandas` in Python and say we have some data contained in a data frame object called “df”
- `colnames(df)` vs `df.columns`
- `nrow(df)` vs `df.shape[0]`
- `apply(X = df, MARGIN = 2, FUN = max)` vs `df.apply(func=max, axis=0)`
- Note that methods/functions which are associated with objects frequently change attributes/data of their objects. In pandas, e.g. `df.rename(columns={"old_name": "some_new_name"}, inplace=True)` changes the data frame’s associated column names and is thereby impure
- Similarly, the `fit()` method in `some_regression_object.fit(X,y)` in `scikit-learn` runs the regression with some data `X,y` and updates the coefficients stored in the `some_regression_object`. In contrast, the `lm()` function in R takes the data as input and returns the estimated linear regression object as output

Outline

1. Introduction
2. Fundamentals and data structures
3. Control flow
4. Functions
5. **Reading in data and plotting**
6. Data science workflows with R today

Reading in and writing out .csv files

```
my_data <- read.csv(file = "my_file.csv")
```

- `my_data` is an R data.frame object
- `my_file.csv` is a .csv file with your data
- Might need to use the `stringsAsFactors = FALSE` argument
- In order for R to load `my_file.csv`, it will have to be saved in your current working directory
    - Use `getwd()` to check your current working directory
    - Use `setwd()` to change your current working directory
- Otherwise define the full path to the file

```
write.csv(my_data, "my_file.csv")
```

An alternative: Creating some (pseudo-)random data

```
set.seed(123)
n <- 1000
x <- rnorm(n)
z <- runif(n)
g <- sample(letters[1:6], n, replace = T)
beta <- 0.5
beta2 <- 0.3
beta3 <- -0.4
alpha <- 0.3
eps <- rnorm(n, sd = 1)
y <- alpha + beta * x + beta2 * z + beta3 * (x * z) + eps
my_data <- data.frame(x = x, y = y, z = z, g = g)
```

Plots

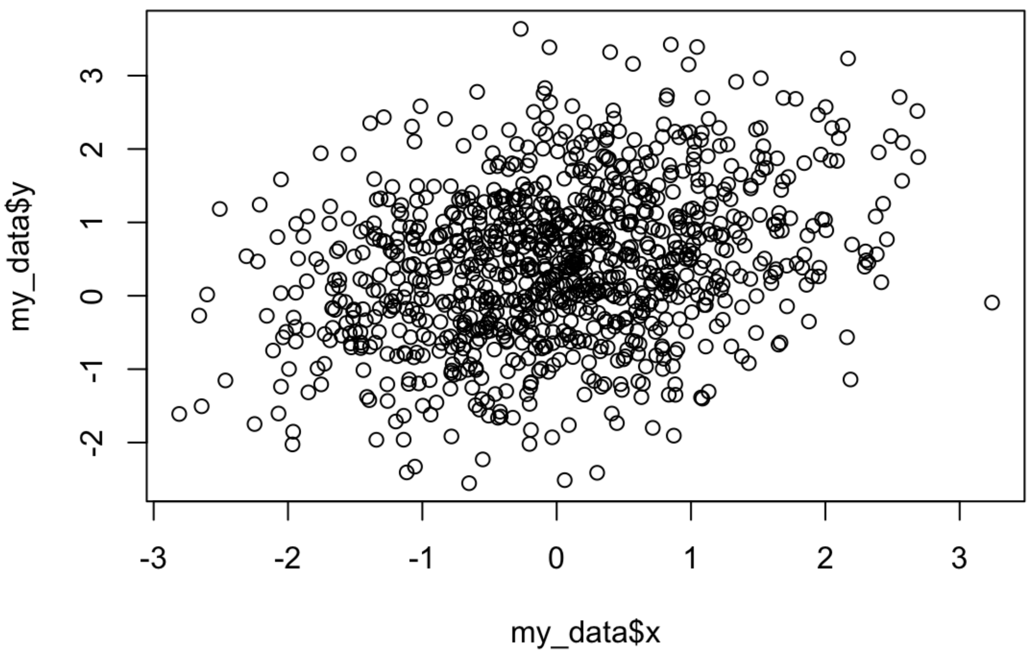
- Plots are one of the strengths of R
- There are two main frameworks for plotting

  1. Base R graphics
  2. **ggplot2**

Base R plots

- The basic plotting syntax is very simple
- `plot(x_var, y_var)` will give you a scatter plot

```
plot(my_data$x, my_data$y)
```

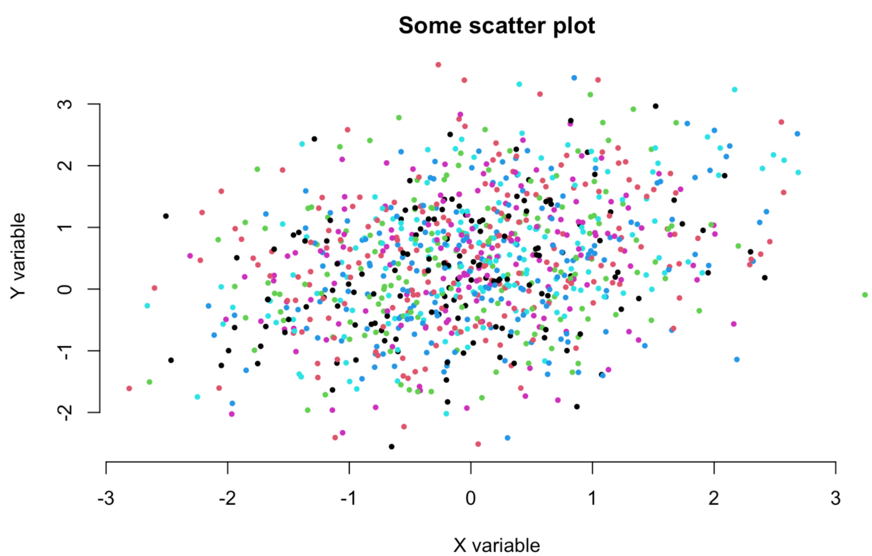


- This can be substantially improved even with base R plots

Base R plots

- The plot function takes a number of arguments (`?plot` for a full list). The fewer you specify, the simpler your plot (as shown above:

```
plot(x = my_data$x, y = my_data$y,
     xlab = "X variable",             # x axis label
     ylab = "Y variable",             # y axis label
     main = "Some scatter plot", # main title
     pch = 19,                        # solid points
     cex = 0.5,                       # smaller points
     bty = "n",                       # remove surrounding box
     col = as.factor(my_data$g)       # colour by grouping variable
)
```



Outline

1. Introduction
2. Fundamentals and data structures
3. Control flow
4. Functions
5. Reading in data and plotting
6. **Data science workflows with R today**

Data science in R

- MY470 is a course about programming, so we covered fundamentals of the R language in this lecture
- At the same time, this provided the necessary knowledge for you to use a range of tools in subsequent work
- The following gives an outlook and many links to resources that you can use

Excellent books on the R language

- R programming
  - Advanced R by Hadley Wickham: https://adv-r.hadley.nz/

- More applied focus on data science in R
  - R for Data Science by Hadley Wickham and Garrett Grolemund: https://r4ds.had.co.nz/

Data processing with R

- `tidyverse`: Collection of packages such as `tidyr`, `dyplr`, `gplot2`, etc.
- More information about the tidyverse: https://www.tidyverse.org/

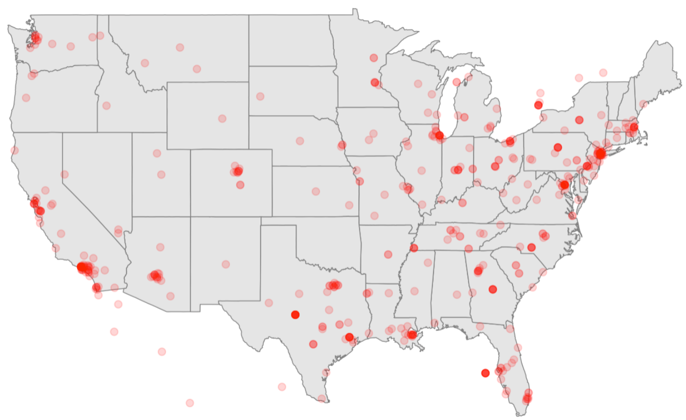
-> “The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.”

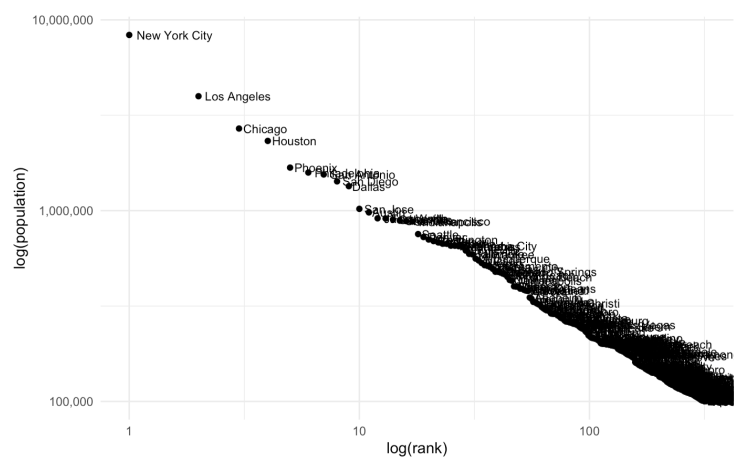
- Summary of how to use the `tidyverse` packages in the R for Data Science book https://r4ds.had.co.nz/

- `data.table`: Particularly fast package to process very large datasets

Visualisation with R: ggplot2

- `ggplot2` is a very flexible tool for visualisation
- Book by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen: https://ggplot2-book.org/



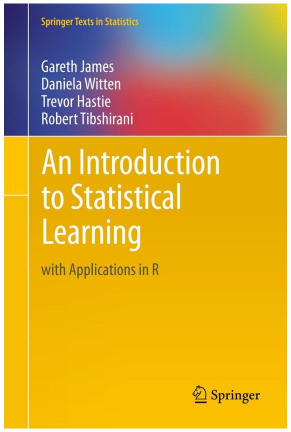


Web scraping and accessing APIs with R

- Scraping static content with `rvest`
- Scraping dynamic content and webforms with browser automation through `RSelenium`
- Accessing generic APIs with `httr`
- Many wrappers for specific APIs exist, e.g. `rtweet` for Twitter
- Side note: Great list of APIs that could e.g. be useful for dissertations, essays, etc. https://github.com/public-apis/public-apis

Statistical machine learning with R

- Range of packages from LASSO regressions (`glmnet`) to random forest (`randomForest`) or support vector machines (`e1071`)
- Excellent book that describes most key concepts in statistical machine learning: http://faculty.marshall.usc.edu/gareth-james/ISL/



- Particularly helpful: Has R appendices with code implementing all methods

Textual analysis with R

- For example, the `quanteda` package offers a wide range of functionalities in textual analysis
- Quickstart: https://quanteda.io/articles/quickstart.html
- Tutorials: https://tutorials.quanteda.io/

Network analysis

- Book by Douglas A. Luke: https://www.springer.com/de/book/9783319238821
- Packages such as `igraph`