

# MY470 Computer Programming: The R Language

Main contributors: Friedrich Geiecke, Pablo Barbera, Ken Benoit

Week 8 Lecture



# Outline

1. Introduction
2. Fundamentals and data structures
3. Control flow
4. Functions
5. Reading data and plotting
6. Resources

# R in a Nutshell

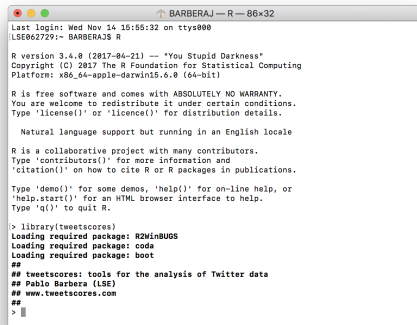
- ▶ Open source programming language under GPL
- ▶ Inspired by the programming language S
- ▶ Useful for statistics and data science, graphics, network analysis, machine learning, web scraping, general purpose programming. . .
- ▶ Superior (if not just comparable) to commercial alternatives. R has over 10,000 user contributed packages (CRAN) and many more elsewhere
- ▶ Available on all platforms
- ▶ Object oriented, but at its core a functional language
- ▶ Interpreted — can be run interactively or as scripts

# Installing R and RStudio

- ▶ R 4.2.2 – Install from <https://www.r-project.org/>
- ▶ RStudio – Install afterwards from <https://www.rstudio.com/products/rstudio/download/>

# Running R Interactively: Terminal

- ▶ 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.



```
Last login: Wed Nov 14 15:55:32 on ttys000
LSE062729:~ BARBERAJ$ R

R version 3.4.0 (2017-04-21) -- "You Stupid Darkness"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

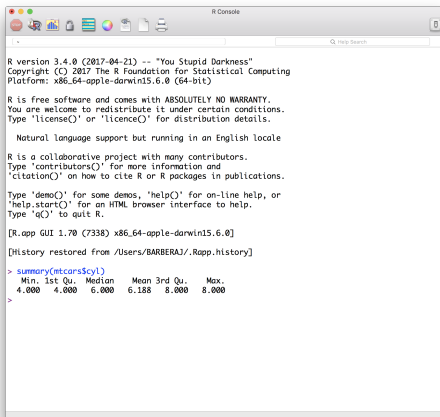
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> library(tweetscores)
Loading required package: R2WinBUGS
Loading required package: coda
Loading required package: boot
##
## tweetscores: tools for the analysis of Twitter data
## Pablo Barbera (LSE)
## www.tweetscores.com
##
> |
```

# Running R Interactively: R GUI Console

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



```
R version 3.4.0 (2017-04-21) -- "You Stupid Darkness"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

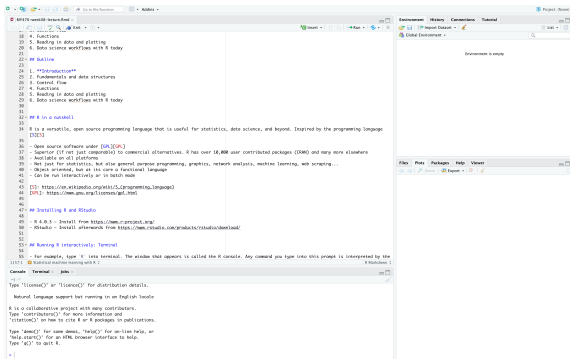
[R.app GUI 1.70 (7338) x86_64-apple-darwin15.6.0]

[History restored from /Users/BARBERAJ/.Rapp.history]

> summary(mtcars$cyl)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.000  4.000   6.000   6.188   8.000   8.000
>
```

# Running R Interactively: RStudio (Preferred)

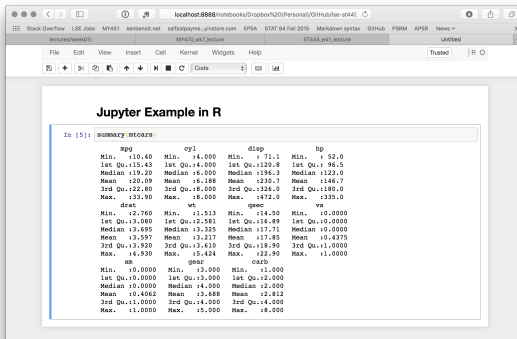
- IDE (Integrated Development Environment) that makes many things easier





# Running R Interactively: Jupyter

- ▶ First, install R in Anaconda Navigator
- ▶ Change the kernel to R
- ▶ May cause problems if R already installed
- ▶ R Markdown is the more natural option in R (great integration with RStudio)



The screenshot shows a Jupyter Notebook interface with a web browser at the top displaying the URL `localhost:8888/notebooks/Dropbox%20(Personal)/Github/jse-ep441`. The notebook has a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, cell execution, and zooming. The main content area is titled "Jupyter Example in R" and contains a code cell with the following R code and output:

```
In [5]: summary(mtcars)
```

mpg		cyl	disp	hp			
Min.	10.40	Min.	14.000	Min.	1 71.1	Min.	1 52.0
1st Qu.	15.43	1st Qu.	14.000	1st Qu.	130.8	1st Qu.	1 96.5
Median	19.20	Median	16.000	Median	196.3	Median	1 23.0
Mean	20.09	Mean	16.188	Mean	230.7	Mean	1 146.7
3rd Qu.	22.80	3rd Qu.	18.000	3rd Qu.	234.0	3rd Qu.	1 180.0
Max.	33.90	Max.	18.000	Max.	472.0	Max.	1 335.0

drat		wt	qsec	vs			
Min.	12.760	Min.	11.513	Min.	14.50	Min.	1 0.0000
1st Qu.	13.080	1st Qu.	12.581	1st Qu.	14.89	1st Qu.	1 0.0000
Median	13.695	Median	13.325	Median	17.71	Median	1 0.0000
Mean	13.597	Mean	13.217	Mean	17.85	Mean	1 0.4375
3rd Qu.	13.920	3rd Qu.	13.610	3rd Qu.	18.90	3rd Qu.	1 1.0000
Max.	14.930	Max.	15.424	Max.	22.90	Max.	1 1.0000

am		gear	carb		
Min.	10.0000	Min.	13.000	Min.	1 1.000
1st Qu.	10.0000	1st Qu.	13.000	1st Qu.	1 2.000
Median	10.0000	Median	4.000	Median	1 2.000
Mean	10.4062	Mean	3.688	Mean	1 2.812
3rd Qu.	11.0000	3rd Qu.	14.000	3rd Qu.	1 4.000
Max.	11.0000	Max.	15.000	Max.	1 8.000

# 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
- ▶ To install a package, run directly in R  
`install.packages("packagename")` (similar to `pip install` and `conda install`, which you run from the terminal, however)
- ▶ To load a package, include `library("packagename")` at the beginning of your script (similar to `import`)

# Fundamentals and Data Structures

- ▶ Basic operations in R
- ▶ Objects in R
- ▶ Key data structures
  - ▶ Atomic vectors
  - ▶ Lists
  - ▶ Matrices and data frames

# Operators and Mathematical Functions

Python	R
+	+
-	-
/	/
*	*
**	^

```
log(<number>)
```

```
exp(<number>)
```

```
sqrt(<number>)
```

```
mean(<numbers>)
```

```
sum(<numbers>)
```

# Logical Operators

Python	R
<	<
>=	>=
==	==
!=	!=
and	&
or	
in	%in%

## Object Assignment

- ▶ The assignment operator in R is `<-`
- ▶ 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, e.g., `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 Allocation

- ▶ 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

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

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

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

```
## [1] "0x7fc0b45ca8b8"
```

```
address(y)
```

```
## [1] "0x7fc0b45ca8b8"
```



# Mutable/Immutable Objects

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

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

```
address(x)
```

```
## [1] "0x7fc0b53d78f8"
```

```
address(y)
```

```
## [1] "0x7fc0b45ca8b8"
```

► Can check this in Python as well with `id()`:

# <i>Mutable</i>	# <i>Immutable</i>
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
```

# Viewing Objects in Your Global Environment and Removing Them

- ▶ List objects in your current environment

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

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

- ▶ Remove objects from your current environment

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

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

```
rm(list = ls())    # Remove all
```

# Object Oriented Programming (OOP) in R

- ▶ “Everything that exists in R is an object” (John Chambers)
- ▶ However, **object oriented programming (OOP)** is much less important in the daily use of R than functional programming
- ▶ Functional programming treats computation as the evaluation of mathematical functions avoiding mutable data
- ▶ 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 [here](#)

# Common Data Structures in R

Dimension	Stores homogenous elements	Stores heterogenous elements
1D	Atomic vector	<b>List</b>
2D	Matrix	<b>Data frame</b>
nD	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 of Python and R

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

# Vectors in R

- ▶ Atomic vectors
- ▶ 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 `raw` and `complex` for this lecture)

- ▶ `character`
- ▶ `integer`
- ▶ `double`
- ▶ `logical`

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

# 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_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)
```

# Everything in R Is a Vector!

- ▶ 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 mode
```

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

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

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

## Add Elements to Vectors

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

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

## 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  
## [16] 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4  
## [31] 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9  
## [46] 5.5 5.6 5.7 5.8 5.9 6.0 6.1 6.2 6.3 6.4  
## [61] 7.0 7.1 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9  
## [76] 8.5 8.6 8.7 8.8 8.9 9.0 9.1 9.2 9.3 9.4  
## [91] 10.0
```

```
class(series)
```

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

# Index Elements

- ▶ **R indices start at 1**
- ▶ **R slices include the last element**
- ▶ So `myvector[1:3]` selects 1st, 2nd, and 3rd elements in R
- ▶ In Python, it should be `mylist[0:3]`



# Implications for Indexing Characters

- ▶ 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[1:4]
```

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

## Implications for Indexing Characters

- ▶ Instead, use specialized functions

```
substr(firstletters, 1, 3)
```

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

- ▶ 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
```

- ▶ 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 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
## length
```

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

```
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
```



## Factors: Vectors with Labels

- ▶ A factor is a special kind of vector
- ▶ It is similar to a character vector, but each unique element is also associated with a numerical value which 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

# 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,  
                                       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)

# 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
```



## 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    4.00 100.00    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 4.00 100.00 7.65
```

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

```
## numbers words
```

```
## 1 5.00 hello
```

```
## 2 4.00 world
```

## 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)]
```

# 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

# 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

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

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

## With an Additional else if Part

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

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

## For-Loops

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

```
for (i in 1:4) {  
  print(i)  
}
```

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

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

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

# While-Loops

```
x <- 1
while (x < 5) {
  print(x)
  x <- x + 1
}
```

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



## Improving Efficiency

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

```
# For example:
x <- 1:1000
y <- 1:1000
z <- numeric(1000)
for (i in 1:1000) {
  z[i] <- x[i]*y[i]
}

# vs:
z <- x*y

# Or:
z <- 0
for (i in 1:1000) {
  z <- z + x[i]*y[i]
}

# vs:
```

# Calling Functions

```
function_name(parameter_one, parameter_two, ...)
```

- ▶ When a function parameter is not assigned a default value, then it is mandatory to be specified by the user
- ▶ Default arguments can be overridden if supplied
- ▶ For example, consider the `mean()` function: `mean(x, na.rm = FALSE)`
- ▶ This function takes two (main) arguments
  - ▶ `x` is a numeric vector
  - ▶ `na.rm` is a logical value that indicates whether we'd like to remove missing values (NA). `na.rm` is set to `FALSE` by default

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

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

## Defining Functions

- ▶ Just like in Python and other programming languages, it is key to create own functions for a modular programme

```
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
```

- ▶ Notice that in R, default arguments must be placed before non-default arguments

## 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
```

# The 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 tidyverse  
x <- c(1,2,3,4,15)  
mean(x)
```

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

```
# Same but with the pipe operator  
x %>% mean()
```

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

# The Pipe Operator

- 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)))
```

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

## Loops Revisited: Apply Functions

- ▶ Another very frequently used approach in R is to replace loops with `apply`
- ▶ It applies a function to every column, row, element of a vector, list, etc.
- ▶ `Apply` exists in Python too, for example `map()` or `df.apply()` in pandas
- ▶ 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)
```

# 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()` 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

## R vs. Python

- ▶ Python, following more the OOP approach, has many methods and attributes attached to objects (recall week 5 on classes)
- ▶ For example, consider R vs. pandas in Python. Let's assume 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)`

## Reading and Writing .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")
```

## Creating (Pseudo-)Random Data

```
set.seed(123)      # set random seed to get replicable results
n <- 1000
x <- rnorm(n)       # draw 1000 points from the normal distribution
z <- runif(n)       # draw 1000 points from the uniform distribution
g <- sample(letters[1:6], n, replace = T) # sample with replacement
# Set some parameters, including noise
beta1 <- 0.5
beta2 <- 0.3
beta3 <- -0.4
alpha <- 0.3
eps <- rnorm(n, sd = 1)
# Generate data that follows a linear trend
y <- alpha + beta1 * x + beta2 * z + beta3 * (x * z) + eps
# Save data in a data frame
my_data <- data.frame(x = x, y = y, z = z, g = g)
```

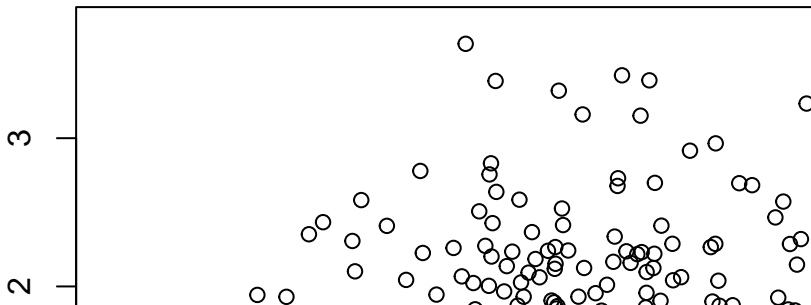
# Plots

- ▶ 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, y)` will give you a scatter plot

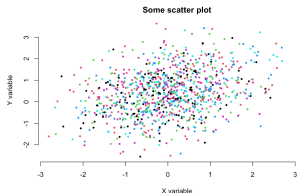
```
plot(my_data$x, my_data$y)
```



# Improving Base R Plots

- ▶ The plot function takes a number of arguments (?plot for a full list)

```
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
```



## Resources



# Data Science with R

- ▶ MY470 is a course about programming, so we covered fundamentals of the R language in this lecture
- ▶ 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

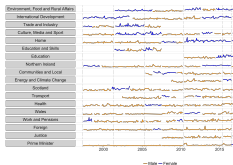
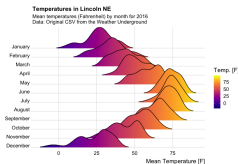
- ▶ Programming in R programming
  - ▶ *Advanced R* by Hadley Wickham: <https://adv-r.hadley.nz/>
- ▶ Applied data science in R
  - ▶ *R for Data Science* by Hadley Wickham and Garrett Grolemund: <https://r4ds.had.co.nz/>

# Data Processing with R

- ▶ tidyverse: Collection of packages such as tidyr, dplyr, ggplot2, etc.
  - ▶ More information: <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 *R for Data Science*: <https://r4ds.had.co.nz/>
- ▶ data.table: Particularly fast package to process very large datasets

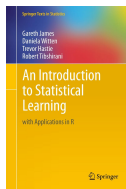
# Visualisation with R: ggplot2

- ▶ ggplot2 is a flexible tool for visualisation
- ▶ Book by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen: <https://ggplot2-book.org>
- ▶ Great website with ggplot2 sample code for many different types of plots:  
<https://www.r-graph-gallery.com/ggplot2-package.html>



# 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/>



- ▶ LSE course **MY474 Applied Machine Learning for Social Science**

# Text Analysis with R

- ▶ The quanteda package
  - ▶ Quickstart: <https://quanteda.io/articles/quickstart.html>
  - ▶ Tutorials: <https://tutorials.quanteda.io/>
- ▶ LSE course **MY459 Quantitative Text Analysis**

# Network Analysis

- ▶ Packages such as igraph
- ▶ Book by Douglas A. Luke:  
<https://www.springer.com/de/book/9783319238821>
- ▶ LSE course **MY461 Social Network Analysis**