UNIVERSITY OF BIRMINGHAM

R for Scientists

LM Regulatory Science and Toxicology for the 21st Century

Dr. Ralf Weber (r.j.weber@bham.ac.uk)
Ossama Edbali (o.edbali@bham.ac.uk)

UNIVERSITY OF BIRMINGHAM

Attendance code: 52258880

UNIVERSITY OF BIRMINGHAM

Chapter 1: Introduction to R and RStudio

Why R?

UNIVERSITY OF BIRMINGHAM

Data analysis is a fundamental step in research, particularly in hypothesis-generating studies. One of the most critical aspects of data analysis is ensuring **reproducibility**.

Over the years, numerous tools have been developed to enhance the analysis of datasets, including commercial software like PowerBI and SPSS. However, these tools often come with licensing costs.

In contrast, R—a powerful and versatile open-source programming language—is specifically designed for **data analysis**, **statistics**, and **visualisation**. With an extensive ecosystem of packages and libraries, R allows users to perform complex statistical analyses, manipulate data efficiently, and create high-quality, customisable visualisations.

Why RStudio?

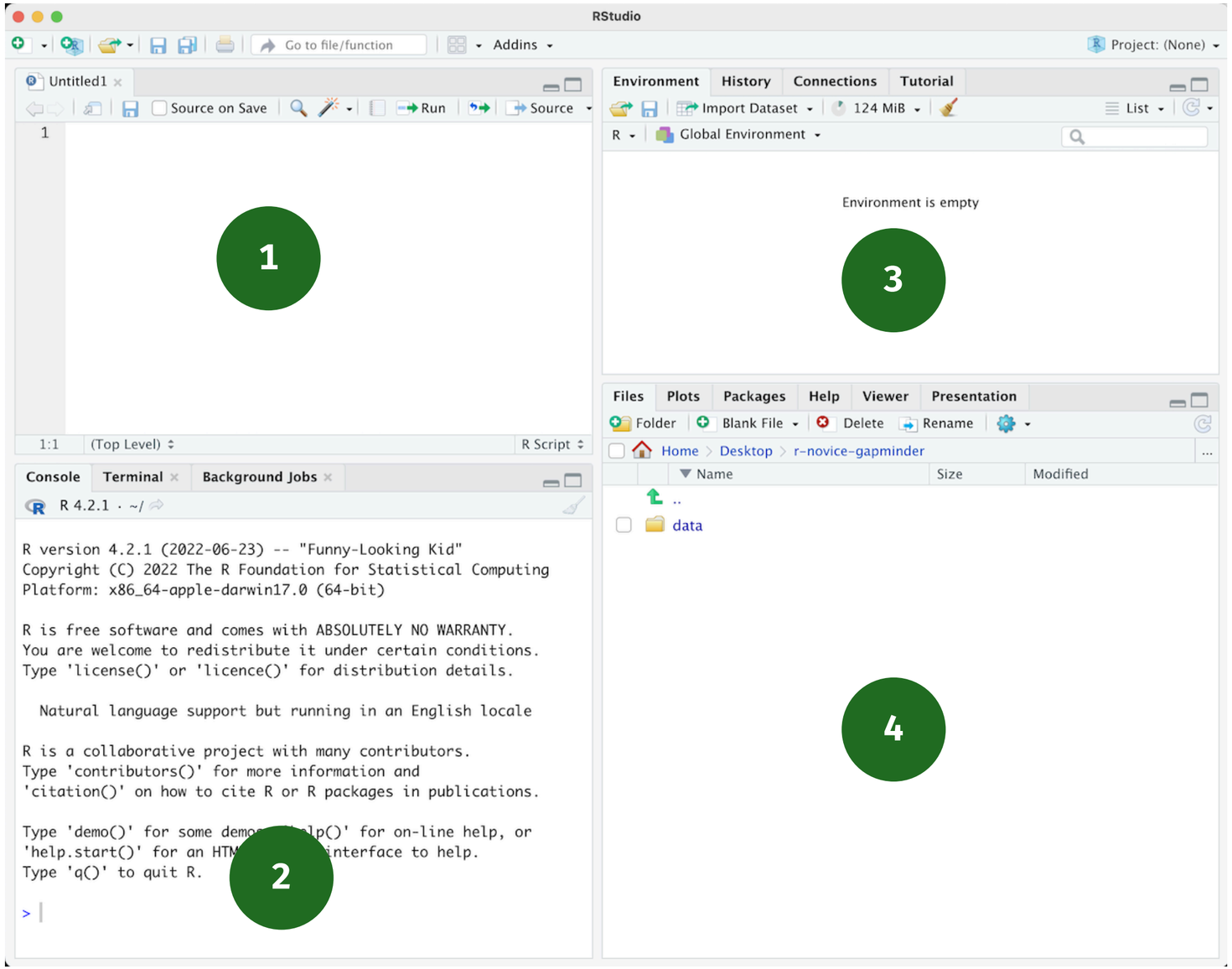
UNIVERSITY OF BIRMINGHAM

RStudio is an **integrated development environment** (IDE) specifically designed for the R programming language. It provides a user-friendly interface and tools that make working with R more efficient and productive.

Think of R as the engine of a car, and RStudio as the dashboard and controls that help you drive the car. You don’t need RStudio to use R, but it makes the whole process much smoother, especially for those new to programming.

Let's explore RStudio

UNIVERSITY OF BIRMINGHAM



1. R files are displayed here. Here you can edit, run the whole file or some lines.

2. R console where you can submit R commands. This is usually used for testing and checking commands.

3. R global environment where you can see all the variables and functions for the current session.

4. Explore files, plots, help pages, and packages.

Introduction to R

UNIVERSITY OF BIRMINGHAM

Open RStudio and in the Console window input the following commands:

Use R as calculator

- `1 + 100`
- `3 + 5 * 2`
- `(3 + 5) *2`

Comparing things

- `1 == 1`
- `2 != 1`
- `31 > 24`

Vectors

- `x <- 1:5`
- `x <- c(1, 2, 3)`
- `x <- 1:3`
  `x * 2`
  → `2 4 6`

Mathematical functions

- `sin(1)`
- `log(1)`
- `exp(0.5)`

Variables

- `x <- 10`
  `log(x)`
- `x <- 10`
  `x <- x + 3`

The R environment

UNIVERSITY OF BIRMINGHAM

An R environment is a collection of objects created during an R session.
The default environment is the global environment (see top-right panel in RStudio).

To interact with the global environment, there are several functions:

Function	Description
`ls()`	Lists all the objects in the environment
`rm(x)`	Removes the object named x
`rm(list = ls())`	Removes **all** objects in the environment

R packages

UNIVERSITY OF BIRMINGHAM

R packages are a collection of related functions and data in a well-defined format.
They are the fundamental units of reproducible R code.

Examples of packages are:

ggplot2
Package for plotting

gapminder
A data package with a data excerpt from the Gapminder project

dplyr
Package for data manipulation

UNIVERSITY OF BIRMINGHAM

Chapter 2: Project Management with RStudio

Project management with RStudio

UNIVERSITY OF BIRMINGHAM

As researchers we deal with data, results, and workflows/processes to produce such results. This requires careful planning, organisation, and documentation to ensure that our work is accurate, reproducible, and efficient.
In this context, RStudio allows you to create a self-contained and reproducible projects.

Best practices for project organisation

- Treat data as read only
- Data cleaning
- Treat generated output as disposable
- Separate function definition and application
- Modularisation
- Version control your project

UNIVERSITY OF BIRMINGHAM

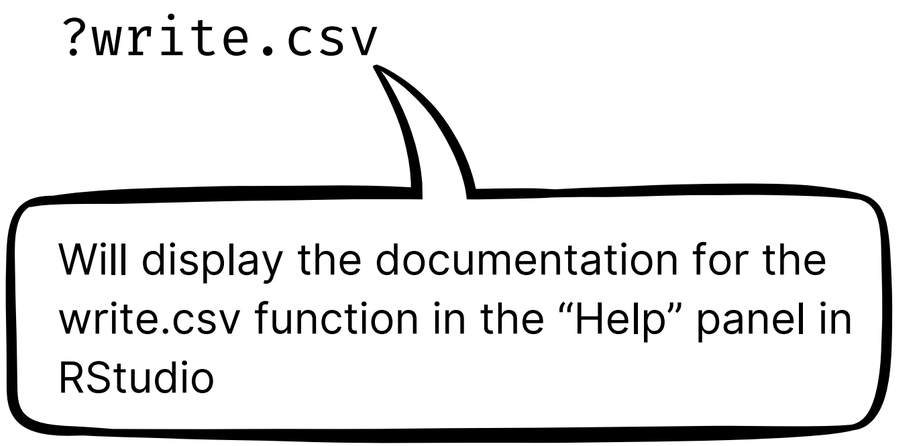
Chapter 3: Seeking help

Seeking help

UNIVERSITY OF BIRMINGHAM

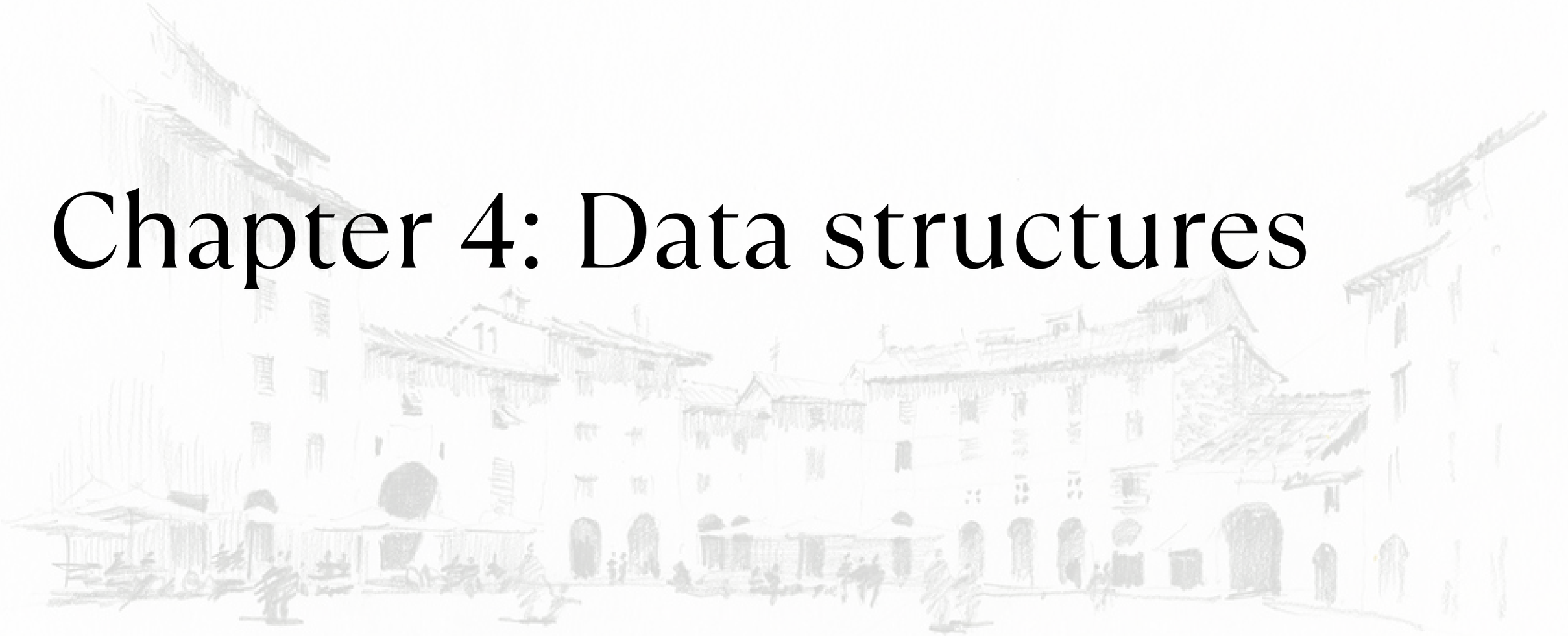
Often, when installing a new package we would like to explore how it is used. We can do so through different ways:

- Explore the vignettes (e.g., ggplot2 vignettes)
- Reference manual (e.g., ggplot2 manual)
- In RStudio by prefixing the package or function name with a "?"



UNIVERSITY OF BIRMINGHAM

Chapter 4: Data structures



Overview

UNIVERSITY OF BIRMINGHAM

A data structure is a specialized format for organizing, processing, retrieving and storing data.

Let's prepare a toy example using a ***data frame***:

```
cats <- data.frame(
    coat = c("calico", "black", "tabby"),
    weight = c(2.1, 5.0, 3.2),
    likes_catnip = c(1, 0, 1))
```

coat	weight	likes_catnip
calico	2.1	1
black	5.0	0
tabby	3.2	1

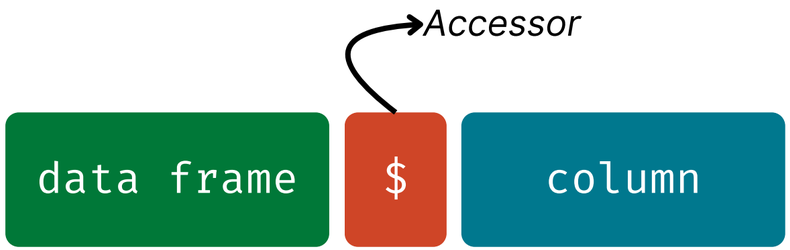
```
write.csv(x = cats, file = "data/feline-data.csv", row.names = FALSE)
```

More on data frames in **slide 28**!

Data types

UNIVERSITY OF BIRMINGHAM

Let’s inspect the dataset by pulling the individual columns:



“From the data frame give me this column”

```
cats$weight
```

```
2.1 5.0 3.2
```

```
typeof(cats$weight)
```

```
“double”
```

```
cats$coat
```

```
“calico” “black” “tabby”
```

```
typeof(cats$coat)
```

```
“double”
```

Data types

UNIVERSITY OF BIRMINGHAM

Data type	Examples
double	2.1. 3.4. -5.75
integer	3 1 7 -4
complex	3 + 8i
logical	TRUE FALSE
character	“hello”

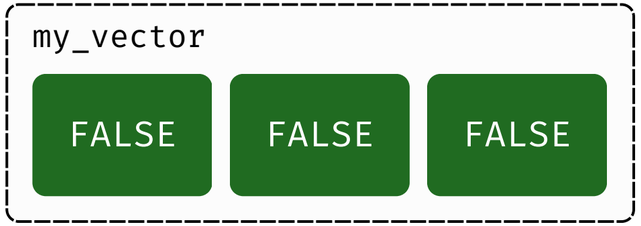
Vectors

UNIVERSITY OF BIRMINGHAM

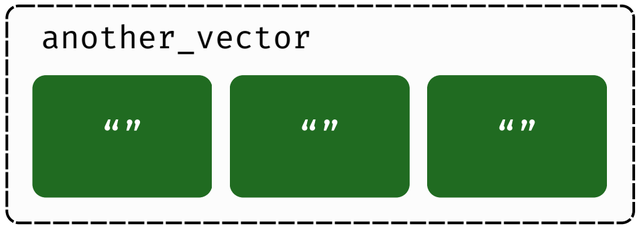
A vector in R is essentially an **ordered list of elements**, with the special condition that *everything in the vector must be the same basic data type*.

To create a vector you can use the "vector" function:

```
my_vector <- vector(length = 3)
```

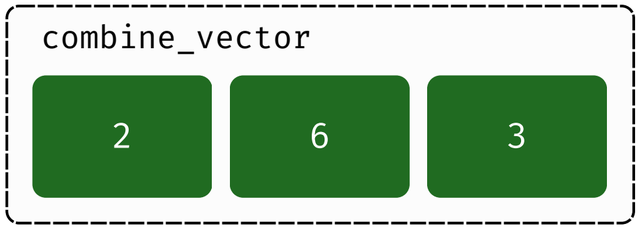



```
another_vector <- vector(mode = "character", length = 3)
```



Or combining elements:

```
combine_vector <- c(2, 6, 3)
```

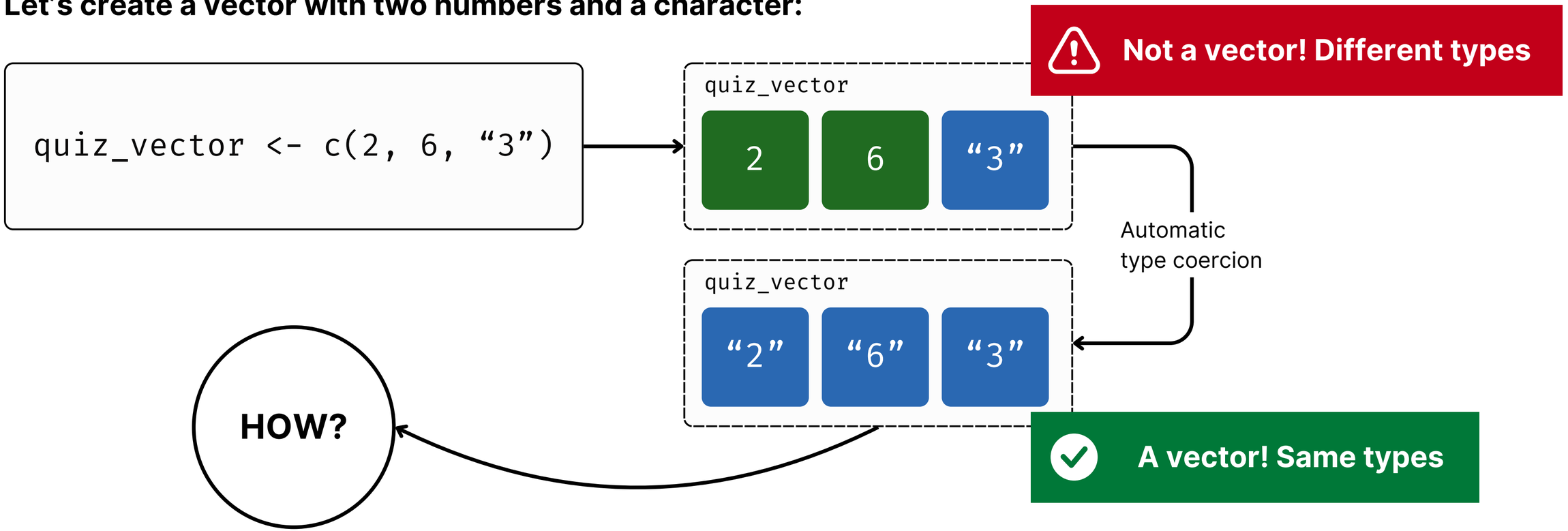


Type coercion

UNIVERSITY OF BIRMINGHAM

Type coercion refers to the *automatic conversion of values from one data type to another*.

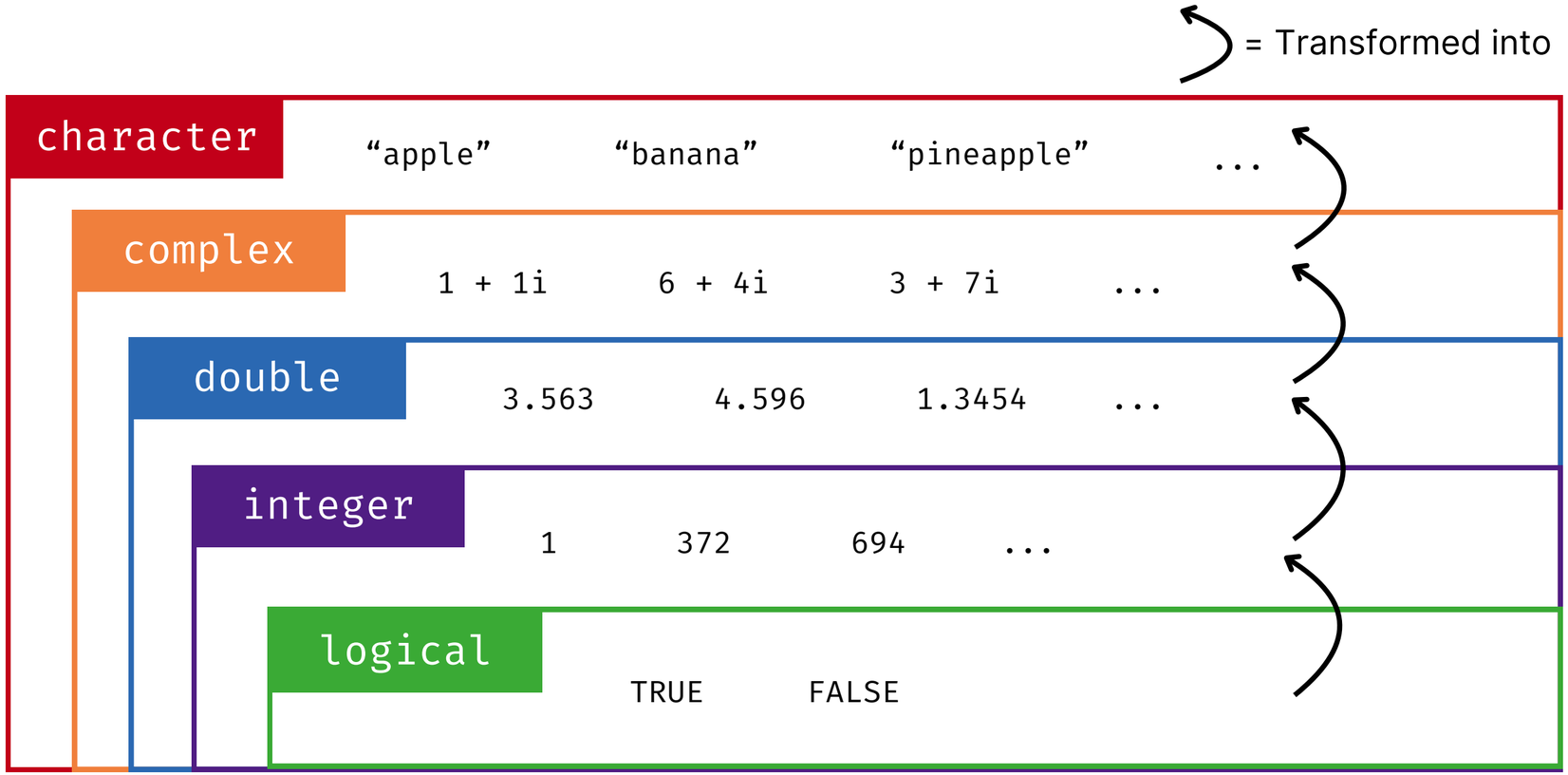
Let's create a vector with two numbers and a character:



Type hierarchy

UNIVERSITY OF BIRMINGHAM

Type coercion refers to the *automatic conversion of values from one data type to another*.



Input vector	Coerced vector
c(FALSE, 1)	c(0, 1)
c(1, 3.563)	c(1.000, 3.563)
c(4.56, 6+4i)	c(4.56+0i, 6.00+4i)
c(3,“banana”)	c(“3”,“banana”)

Basic vector functions

UNIVERSITY OF BIRMINGHAM

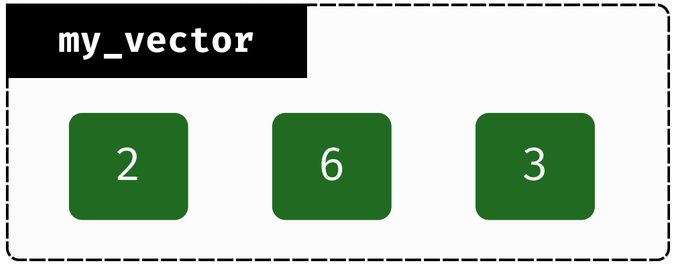
`c(5,2,1)`	`5 2 1`	Join elements into a vector
`1:5`	`1 2 3 4 5`	Create a series of numbers
`seq(5)`		
`vec <- 20:25` **`head(vec, n=2)`**	`20 21`	Get the first *n* elements of a vector
`tail(vec, n=4)`	`22 23 24 25`	Get the last *n* elements of a vector
`length(vec)`	`6`	Get the number of elements
`typeof(vec)`	`"integer”`	Get the data type of the vector
`vec[3]`	`22`	Get the nth element of a vector

Lists

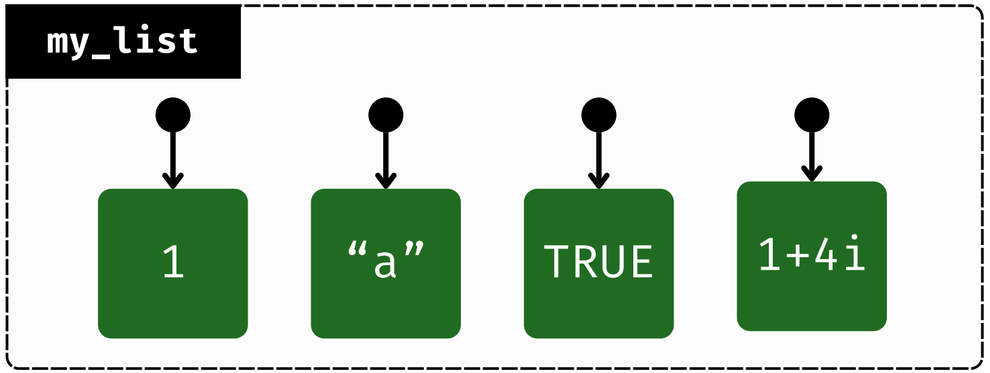
UNIVERSITY OF BIRMINGHAM

A list is a collection of elements. Unlike vectors, lists **can have different data types**.

```
my_vector <- c(2, 6, 3)
```




```
my_list <- list(1, "a", TRUE, 1+4i)
```



How to access values in a vector

```
my_vector[2]
```

→ 6

How to access values in a list

```
my_list[[2]]
```

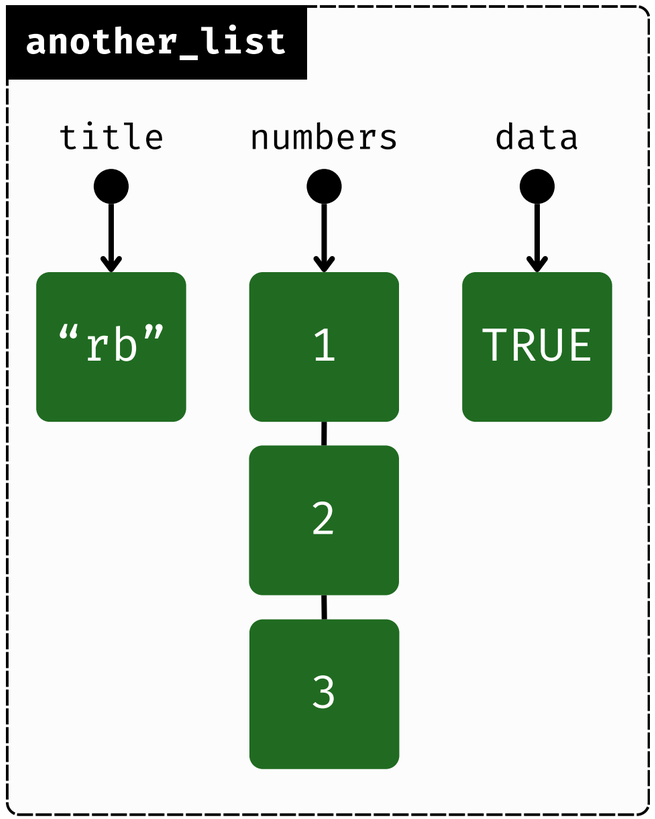
→ “a”

Lists

UNIVERSITY OF BIRMINGHAM

A list can have named entries:

```
another_list <- list(title = "rb", numbers = 1:3, data = TRUE)
```



Output in R

```
$title
[1] "rb"

$numbers
 [1]  1  2  3

$data
[1] TRUE
```

How to access values

```
another_list[[”title”]]
```

or

```
another_list$title
```

Names

UNIVERSITY OF BIRMINGHAM

Vectors and lists can have names attached to values.
Names give meaning to values.

```
person <- list(name = "Marco", age = 35, year = 3)
```

```
pizza <- c(tomato = 3, mozzarella = 2, dough = 5)
```

How to access values

```
person[["age"]]
```

```
person$age
```

```
pizza['dough']
```

Names

UNIVERSITY OF BIRMINGHAM

Accessing names

```
names(person)
```

→ `"name"` `"age"` `"year"`

```
names(person)[2]
```

→ `"age"`

Changing names

```
names(person)[1] <- "first_name"

names(person)
```

→ `"first_name"` `"age"` `"year"`

Data frames

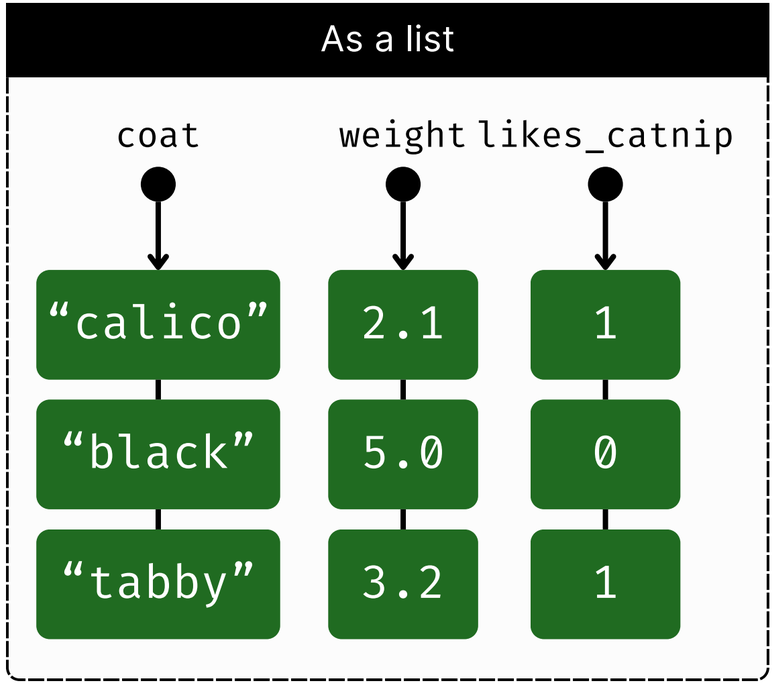
UNIVERSITY OF BIRMINGHAM

Let's explore the **cats** data frame created in slide X.

```
typeof(cats)
```
→ `"list"`

coat	weight	likes_catnip
calico	2.1	1
black	5.0	0
tabby	3.2	1

Under the hood, data frames are lists! More specifically a data frame is a list of vectors of the same length.



Matrices

UNIVERSITY OF BIRMINGHAM

- A matrix is a two dimensional dataset with columns and rows.
- Unlike a data frame, the data type of matrix elements should be the same.

```
matrix_example <- matrix(0, ncol=4, nrow=3)
```

0	0	0	0
0	0	0	0
0	0	0	0

Get dimensions

```
dim(matrix_example)
```
→
```
3  4
```

```
nrow(matrix_example)
```
→
```
3
```

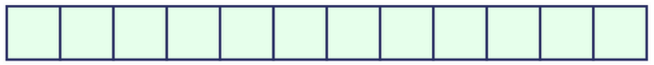
```
ncol(matrix_example)
```
→
```
4
```

Data structures recap

UNIVERSITY OF BIRMINGHAM

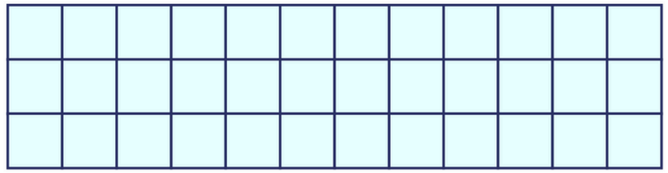
Data types: double, integer, character, logical

Data structures



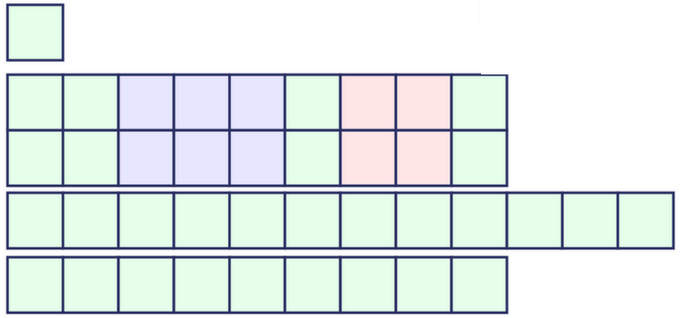
Vector

Same type, one dimensional



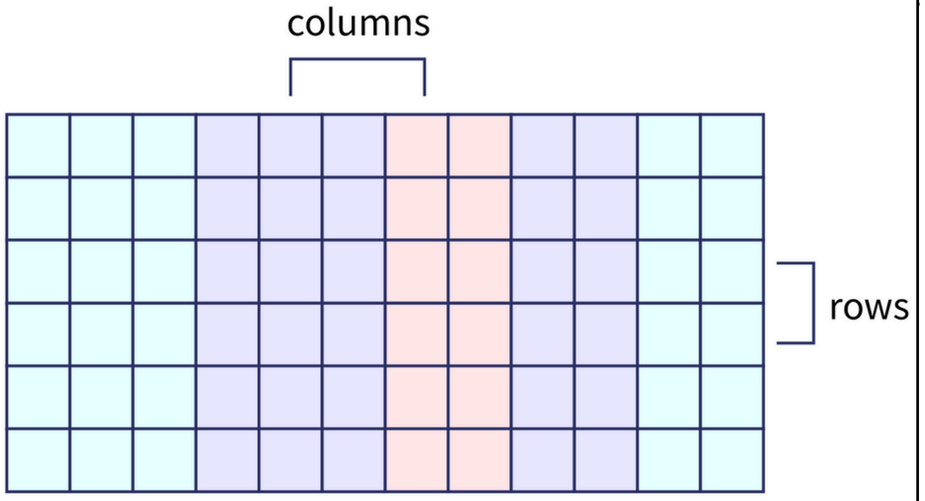
Matrix

Same type, multi dimensional



List

Multi type, multi dimensional



Data frame

Same type per column, two dimensional

UNIVERSITY OF BIRMINGHAM

Chapter 5: Exploring data frames

Adding columns and rows

UNIVERSITY OF BIRMINGHAM

Let's consider the **cats** data frame created earlier and add a column:

```
age <- c(2, 3, 5)
cats <- cbind(cats, age)
```

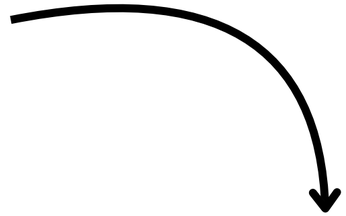
cbind(

coat	weight	likes_catnip
calico	2.1	1
black	5.0	0
tabby	3.2	1

,

age
2
3
5

)



coat	weight	likes_catnip	age
calico	2.1	1	2
black	5.0	0	3
tabby	3.2	1	5

Adding columns and rows

UNIVERSITY OF BIRMINGHAM

Let's add a row:

```
new_cat <- list("short", 3.3, TRUE, 9)
cats <- rbind(cats, new_cat)
```

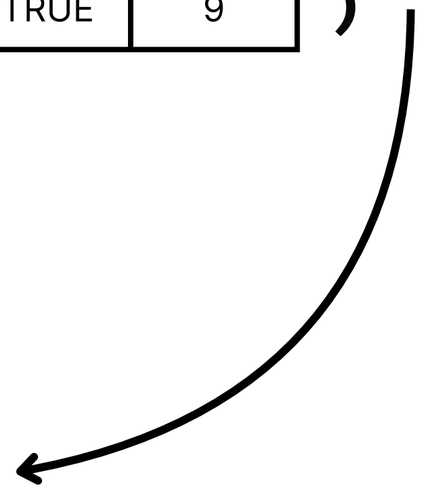
rbind(

coat	weight	likes_catnip	age
calico	2.1	1	2
black	5.0	0	3
tabby	3.2	1	5

,

short	3.3	TRUE	9

)



coat	weight	likes_catnip	age
calico	2.1	1	2
black	5.0	0	3
tabby	3.2	1	5
short	3.3	1	9

Removing columns and rows

UNIVERSITY OF BIRMINGHAM

To remove a column:

```
cats <- cats[, -4]
```

To remove a row:

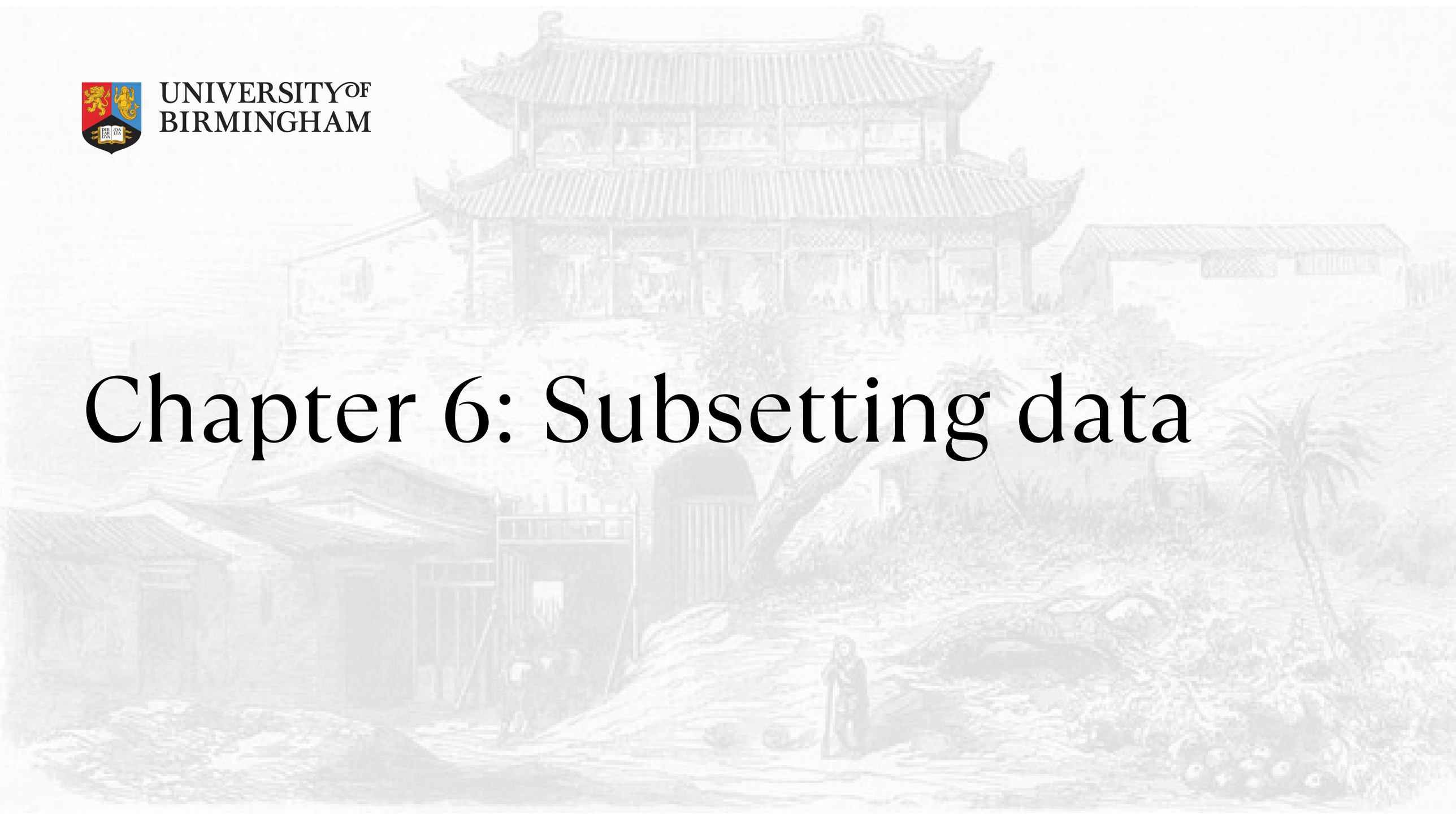
```
cats <- cats[-2, ]
```

Refers to the row index
If positive will select **only** that row
If negative will remove that row

```
cats[x, y]
```

Refers to the **column** index
If positive will select **only** that column
If negative will remove that column

More in the next chapter!



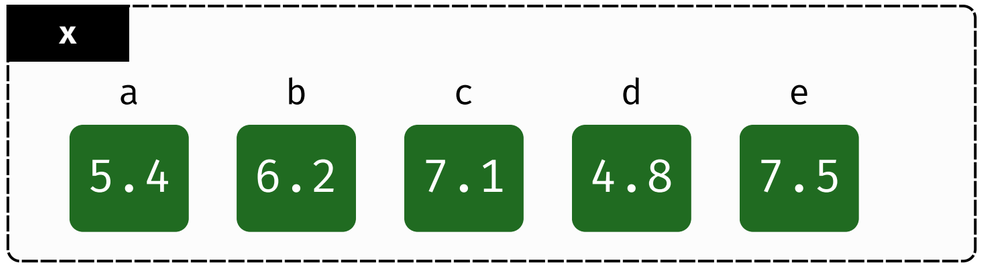
UNIVERSITY OF BIRMINGHAM

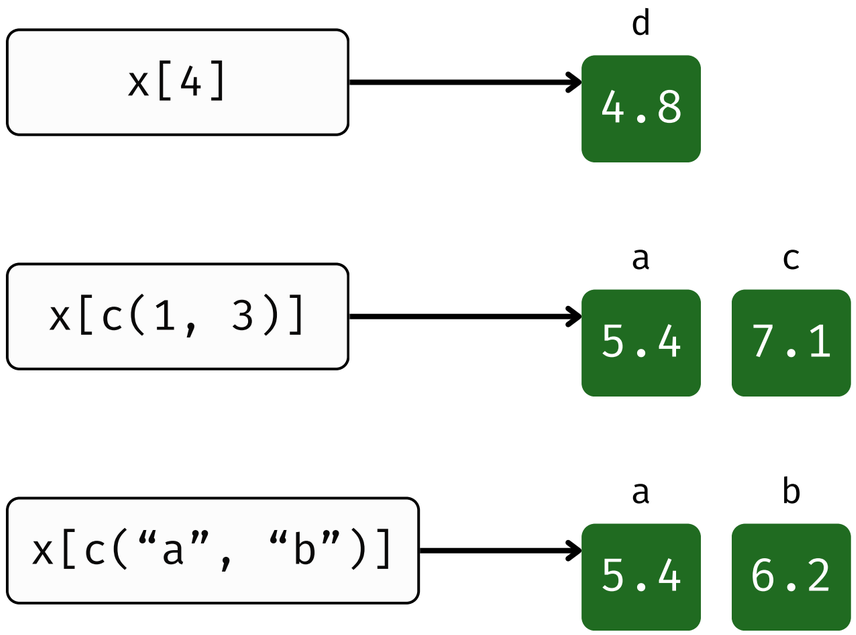
Chapter 6: Subsetting data

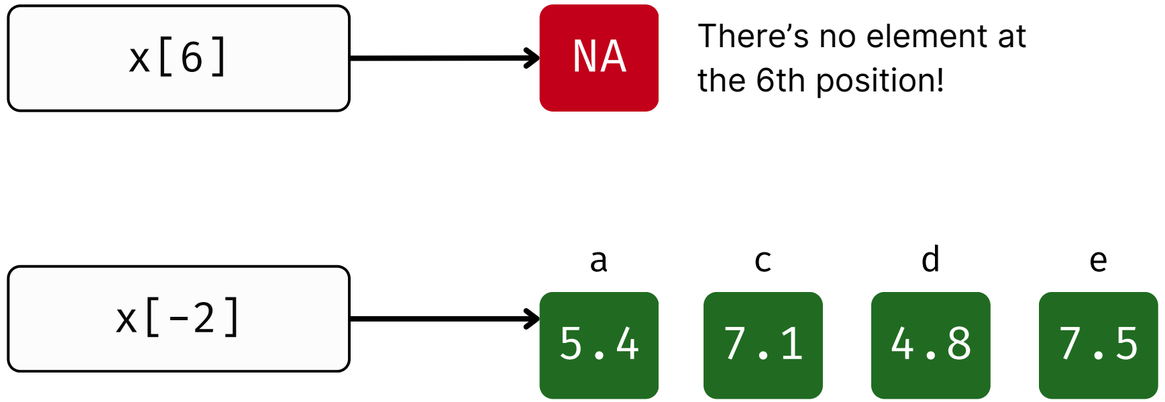
Accessing elements using indices and names

UNIVERSITY OF BIRMINGHAM

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
```



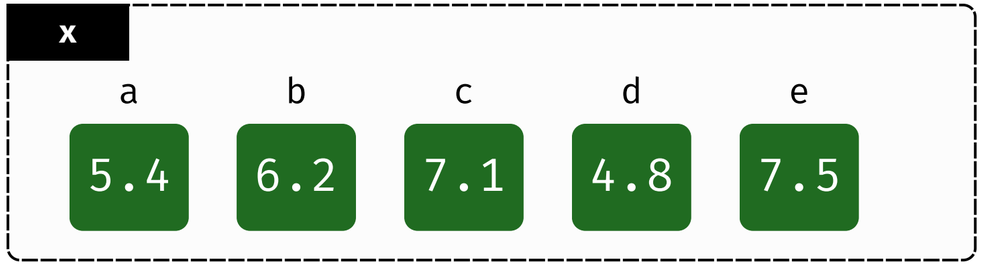




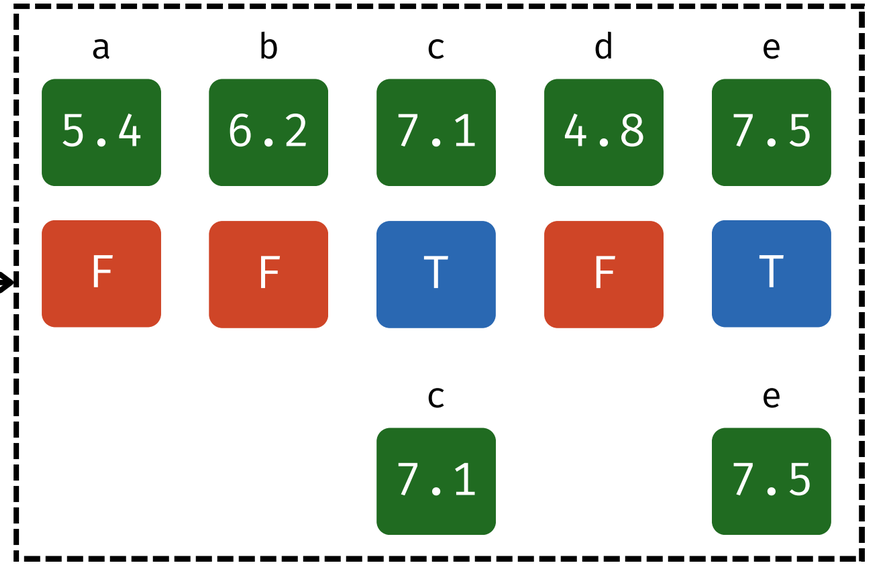
Subsetting through logical operations

UNIVERSITY OF BIRMINGHAM

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
```



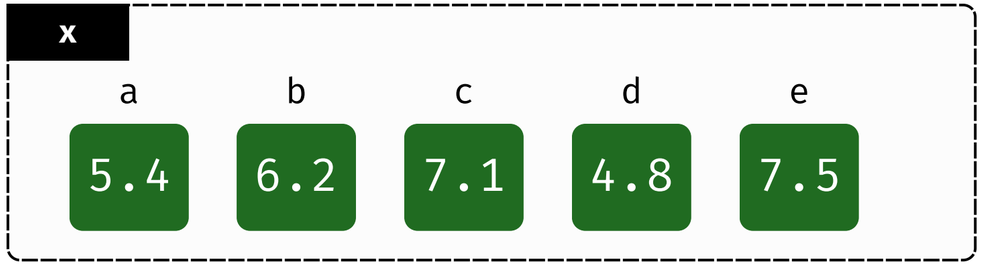

```
x[c(FALSE, FALSE, TRUE, FALSE, TRUE)]
```



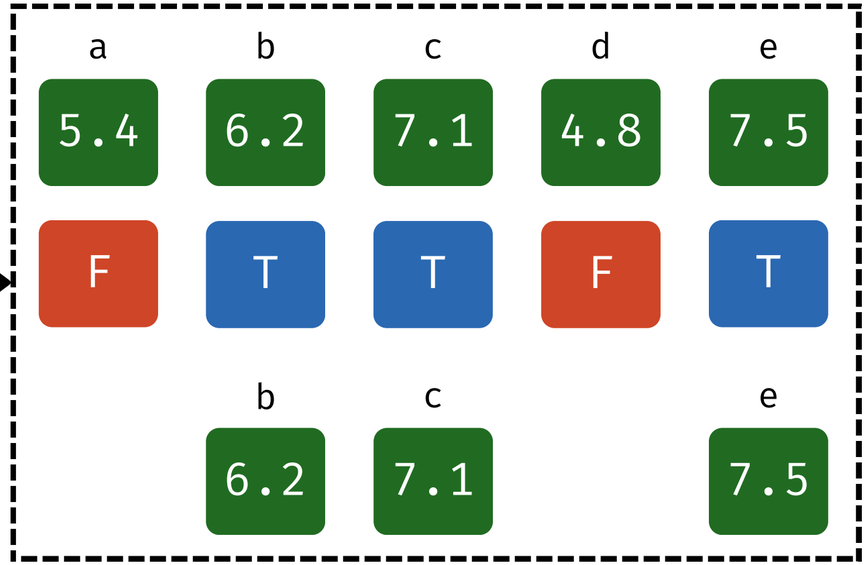
Subsetting through logical operations

UNIVERSITY OF BIRMINGHAM

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
```



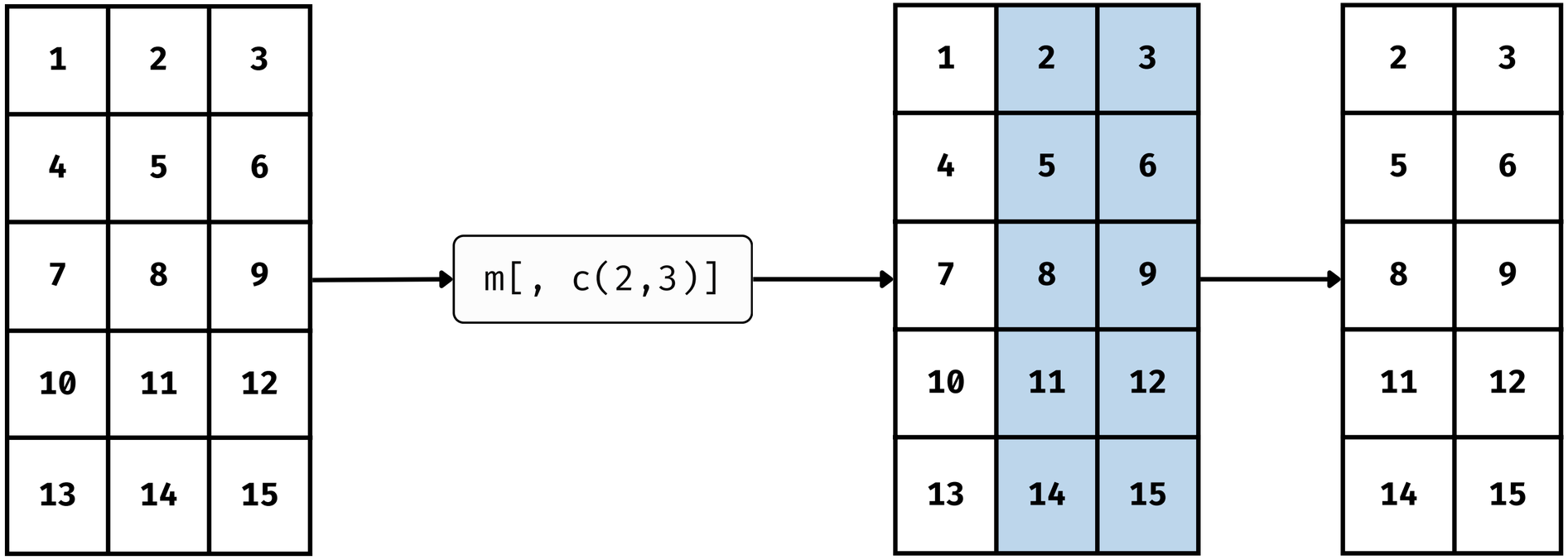

```
x[x > 6]
```



Matrix subsetting

UNIVERSITY OF BIRMINGHAM

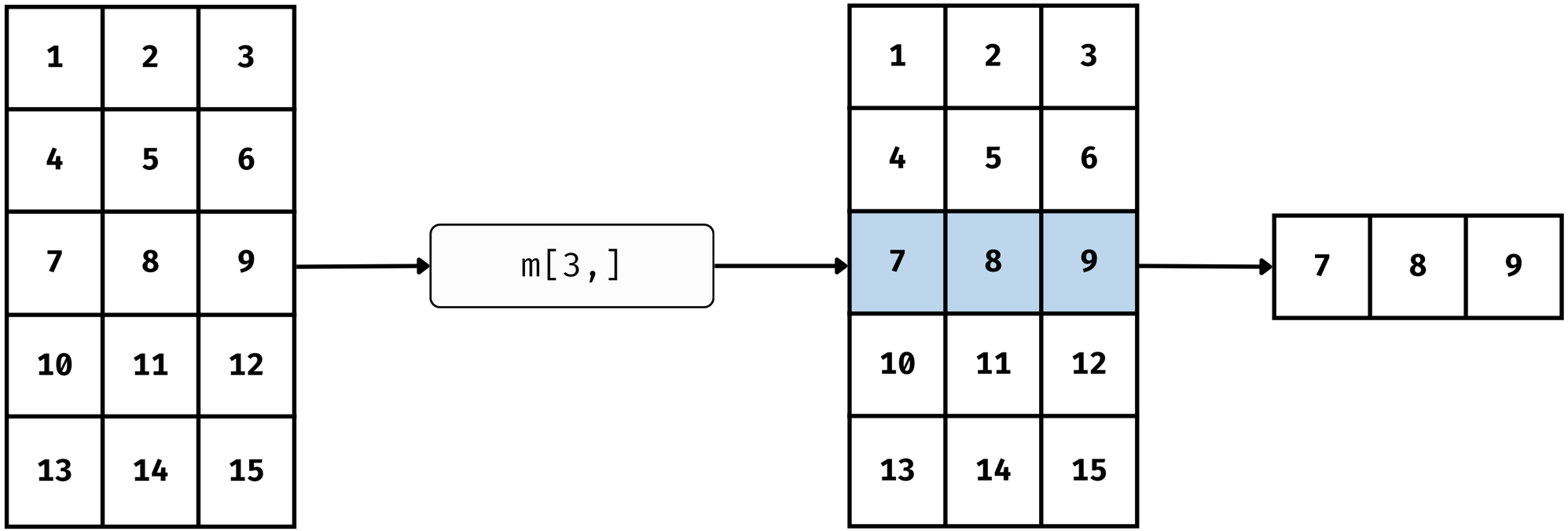
```
m <- matrix(1:15, nrow = 5, ncol = 3, byrow = TRUE)
```



Matrix subsetting

UNIVERSITY OF BIRMINGHAM

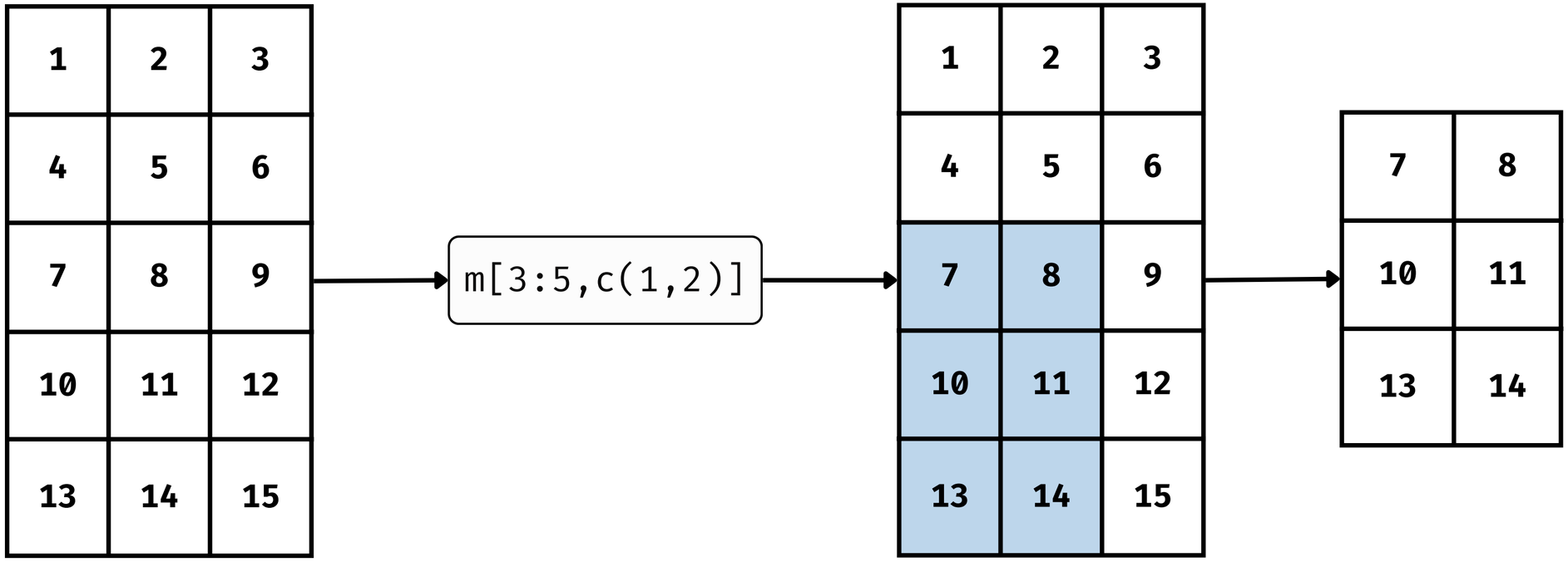
```
m <- matrix(1:15, nrow = 5, ncol = 3, byrow = TRUE)
```



Matrix subsetting

UNIVERSITY OF BIRMINGHAM

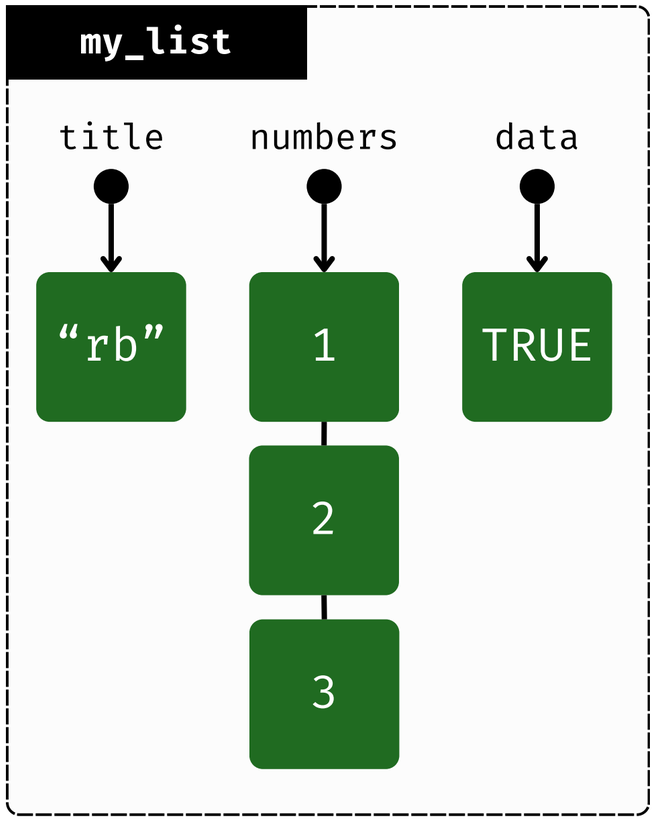
```
m <- matrix(1:15, nrow = 5, ncol = 3, byrow = TRUE)
```



List subsetting

UNIVERSITY OF BIRMINGHAM

```
my_list <- list(title = "rb", numbers = 1:3, data = TRUE)
```



Output in R

```
$title
[1] "rb"

$numbers
 [1]  1  2  3

$data
[1] TRUE
```

How to access values

```
another_list[[”title”]]
```

or

```
another_list$title
```

Subsetting summary

UNIVERSITY OF BIRMINGHAM

Data type	Subsetting options	Description
Vector	`x[4]`	Selects the 4th element in the vector
	`x["age"]`	Selects the element with the name "age"
Matrix	`x[1,]`	Selects the first row
	`x[,3]`	Selects the 3rd column
	`x[1:3,c(3,4)]`	Select the first 3 rows and the 3rd and fourth column
List	`x[["age"]]`	Selects the element with name "age"
	`x$age`	Selects the element with name "age" (alternative)
	`x[1]`	Selects the first element
Data frame	`x[["gdp"]]`	Selects the column "gdp"
	`x$gdp`	Selects the column "gdp" (alternative)
	`x[1:3,"gdp"]`	Selects the first 3 rows and the column "gdp"

UNIVERSITY OF BIRMINGHAM

Chapter 7: Control flow

Control flow

UNIVERSITY OF BIRMINGHAM

Control flow refers to the order in which statements and instructions are executed in a program.

Conditional statements

Conditional statements allow you to run different statements based on conditions.

```
a <- 8

if (a >= 10) {
    print("a is greater than 10")
} else {
    print("a is less than 10")
}
```

Loops

Loops are used to repeat a block of code multiple times until a certain condition is met.

```
number_sequence <- 1:10

for (i in number_sequence) {
    print(i)
}
```

UNIVERSITY OF BIRMINGHAM

Chapter 8: Creating Publication-Quality Graphics with ggplot

Grammar of graphics

UNIVERSITY OF BIRMINGHAM

ggplot is an easy-to-use plotting package for R.
Through the use of a **grammar** the user can **declaratively** create plots.
Such grammar is comprised of 3 main concepts:

Datasets

The data that you, the user, provide.

Mapping aesthetics

What connect the data to the graphics.

Layers

Determine what kinds of plot are shown (scatterplot, histogram, etc.).

Plotting example

UNIVERSITY OF BIRMINGHAM

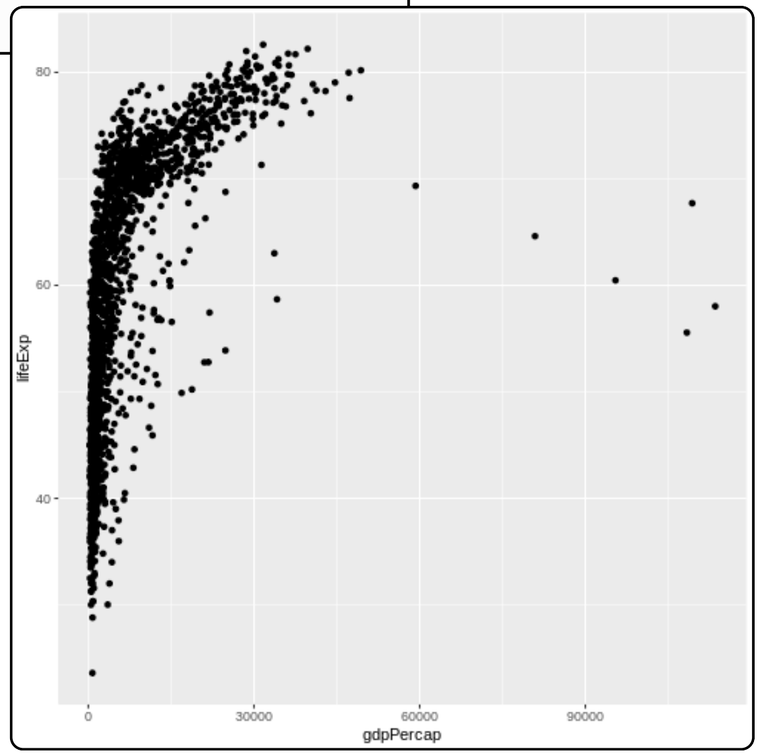
```
library(ggplot2)

ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
    geom_point()
```

Layers

Dataset section

Mapping aesthetic



UNIVERSITY OF BIRMINGHAM

Chapter 9: Vectorization

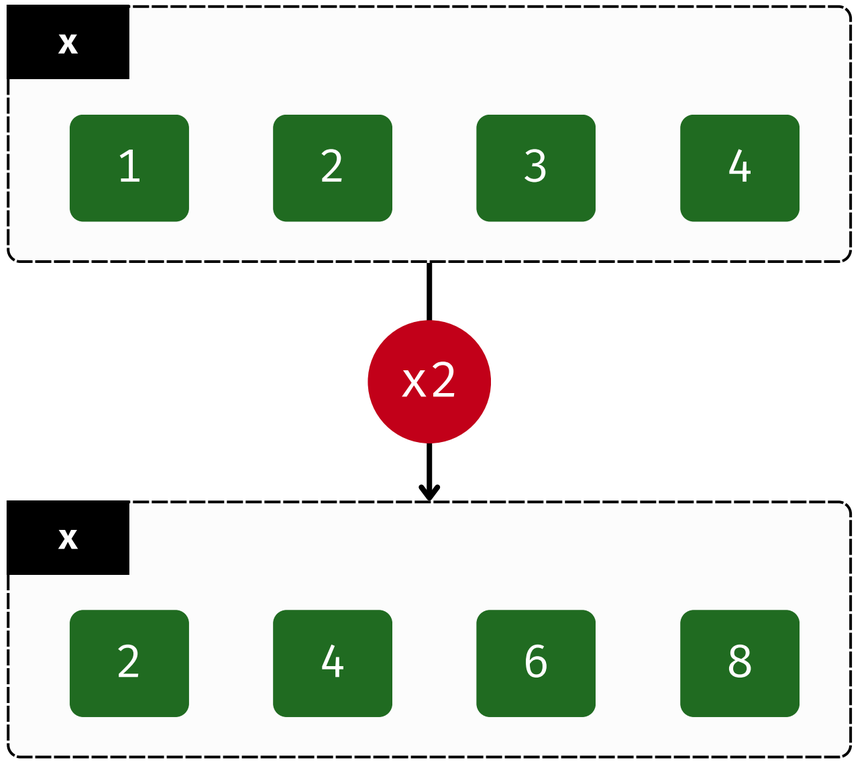
Vectorization: the concept

UNIVERSITY OF BIRMINGHAM

Most functions in R are vectorized, meaning that they operate on vectors.

```
x <- 1:4

x * 2
```



Vectorization: examples

UNIVERSITY OF BIRMINGHAM

```
x <- 1:4
```

Function/operation	Output
`log(x)`	`0.0000000 0.6931472 1.0986123 1.3862944`
`x > 2`	`FALSE FALSE TRUE TRUE`
`x * -1`	`-1 -2 -3 -4`
`as.character(x)`	“1” “2” “3” “4”

UNIVERSITY OF BIRMINGHAM

Chapter 10: Functions

Functions: the concept

UNIVERSITY OF BIRMINGHAM

1 Let's say we want to calculate the total sum of squares:

$$\text{TSS} = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

n = number of observations
y_i = value in a sample
$\bar{y}$ = mean value of a sample

2 In R, given a vector we can calculate it as follows:

```
x <- 1:10
x_mean <- mean(x)
tss <- sum((x - x_mean)^2)
```

3 Given another vector we need to repeat the same code:

```
y <- 30:40
y_mean <- mean(y)
tss <- sum((y - y_mean)^2)
```

Now imagine doing this for many more vectors, so much repetition. Solution: **functions**!

Anatomy of a function

UNIVERSITY OF BIRMINGHAM

Function keyword
Designates the start of the function

Name of the function

Parameters (inputs)

```
calculate_tss <- function(x) {
    x_mean <- mean(x)
    tss <- sum((x - x_mean)^2)
    return(tss)
}
```

Function body
Includes the actions you want to perform

Output

Run a function

UNIVERSITY OF BIRMINGHAM

```
y <- c(1, 2, 3)

tss_y <- calculate_tss(y)
```

```
z <- c(6, 7, 8)

tss_z <- calculate_tss(y)
```

Combining functions

UNIVERSITY OF BIRMINGHAM

```
square <- function(x) {
    return(x^2)
}
```

```
add_one <- function(x) {
    return(x + 1)
}
```

```
x <- 3
y <- add_one(x)
z <- square(y)
```

=

```
z <- square(add_one(x))
```

The order of function application goes from the inner one to the outer one.
A more intuitive approach of combining functions will be covered in **Chapter 12**.

UNIVERSITY OF BIRMINGHAM

Chapter 11: Saving data

Saving plots

UNIVERSITY OF BIRMINGHAM

Saving a single plot

```
p <- ggplot(data = gapminder, aes(x = year, y = lifeExp, colour = country)) +
    geom_line() +
    theme(legend.position = "none")

ggsave("Life_Exp_vs_time.pdf", p)
```

Saving multiple plots

```
pdf("Life_Exp_vs_time.pdf", width=12, height=4)

ggplot(data = gapminder, aes(x = year, y = lifeExp, colour = country)) +
    geom_line() +
    theme(legend.position = "none")

ggplot(data = gapminder, aes(x = year, y = lifeExp, colour = country)) +
 geom_line() +
 theme(legend.position = "none")

dev.off()
```

Create PDF file

One plot for each page

Closes the connection to the PDF file

Saving data

UNIVERSITY OF BIRMINGHAM

```
uk_data <- gapminder[gapminder$country == "United Kingdom",]

write.table(uk_data,
 file="uk-data.csv",
 sep=","
)
```

```
"country","continent","year","lifeExp","pop","gdpPercap"
"1","United Kingdom","Europe",1952,69.18,50430000,9979.508487
"2","United Kingdom","Europe",1957,70.42,51430000,11283.17795
"3","United Kingdom","Europe",1962,70.76,53292000,12477.17707
"4","United Kingdom","Europe",1967,71.36,54959000,14142.85089
"5","United Kingdom","Europe",1972,72.01,56079000,15895.11641
...
```

UNIVERSITY OF BIRMINGHAM

Chapter 12: Data frame manipulation with dplyr

Day-to-day data manipulation

UNIVERSITY OF BIRMINGHAM

As researchers we need to manipulate raw data in different ways:

- Filtering
- Missing value imputation
- Transformation
- Summarising: mean, median, ...
- ...

Let's consider the **gapminder** dataset from R.
To get the average GDP per capita for each continent we can do the following:

```
mean(gapminder$gdpPercap[gapminder$continent == "Africa"])
```

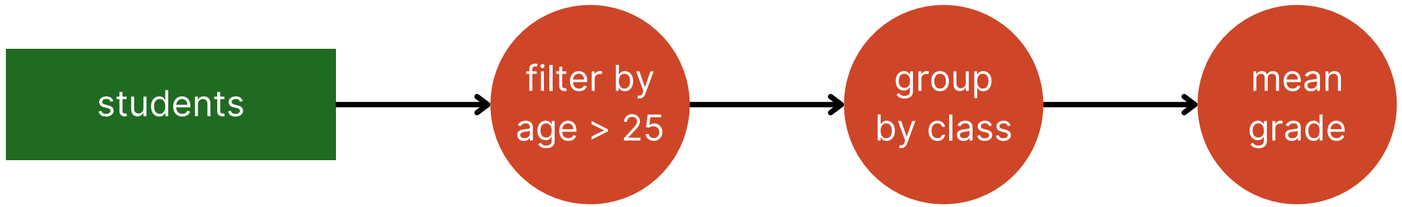
As you can see it's longwinded and needs repetition for each continent.
A better way would be to use a package specifically made
to manipulate data in an easy and intuitive manner, **dplyr**!

```
install.packages("dplyr")
library(dplyr)
```

dplyr: the concept

UNIVERSITY OF BIRMINGHAM

dplyr allows you to easily manipulate data through the concept of **piping** which is a way to combine functions to progressively transform data, just like water pipes carry water across physical spaces.



We will cover the most used functions in dplyr:

- select
- filter
- group_by
- summarise
- mutate

Selecting columns

UNIVERSITY OF BIRMINGHAM

To select a subset of the columns or remove certain columns use the select function:

```
year_country_gdp <- select(gapminder, year, country, gdpPercap)
```

This will only select the **year**, **country**, and **gdpPercap** columns from the **gapminder** dataset.

```
gapminder_modified <- select(gapminder, -continent)
```

This will remove the **continent** column whilst keeping the rest.

Using the piping operator

UNIVERSITY OF BIRMINGHAM

Considering the operations in the previous slide, we can use a more readable syntax through the piping operator %>%. The utility of this syntax will be evident when we combine multiple functions.

```
year_country_gdp <- gapminder %>% select(year, country, gdpPercap)
```

Think of the %>% operator as an arrow, moving data from one state to another!

Filtering rows

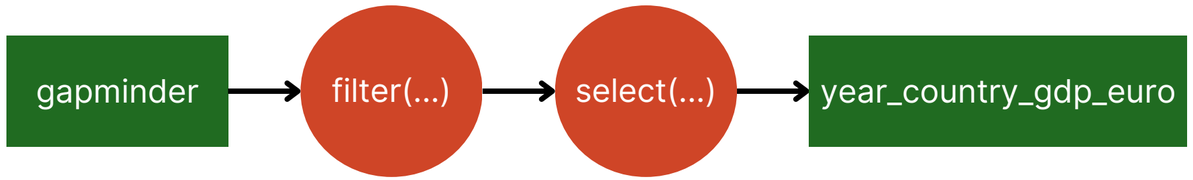
UNIVERSITY OF BIRMINGHAM

To filter rows based on a certain condition we can use the **filter** function:

```
gapminder_europe <- gapminder %>% filter(continent == "Europe")
```

This will keep rows that have Europe as the continent.

```
year_country_gdp_euro <- gapminder %>%
    filter(continent == "Europe") %>%
    select(year, country, gdpPercap)
```



We can combine filtering rows and selecting columns as above.

Grouping data

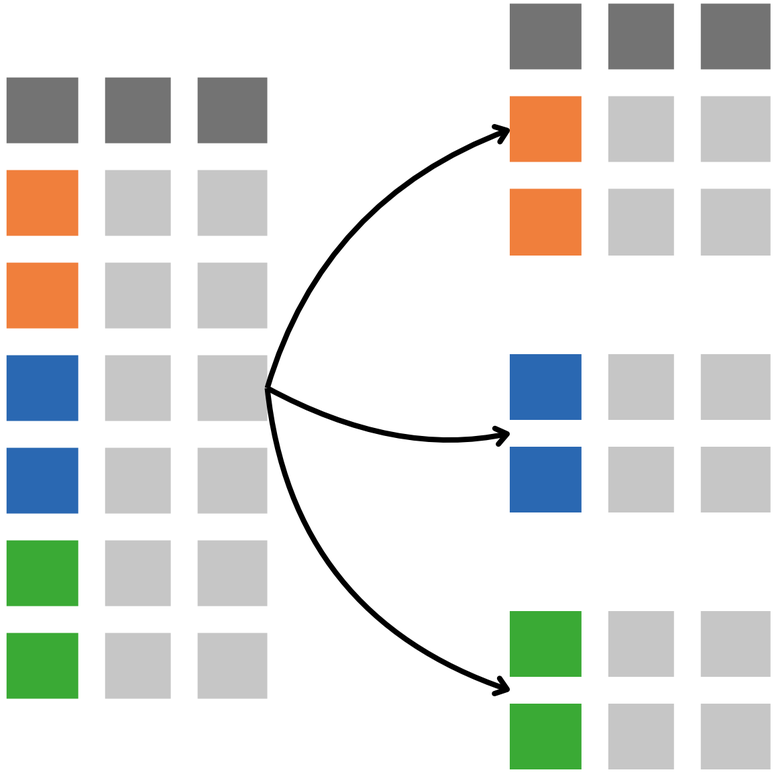
UNIVERSITY OF BIRMINGHAM

Often we want to group data (e.g., by organism) to perform certain operations (e.g., median intensity).

This will group the dataset by continent:

```
gapminder_grouped <- gapminder %>%
    group_by(continent)
```

A grouped data frame can be thought of as a **list of data frames**, where each data frame corresponds to the rows grouped by a certain column (continent in this case).



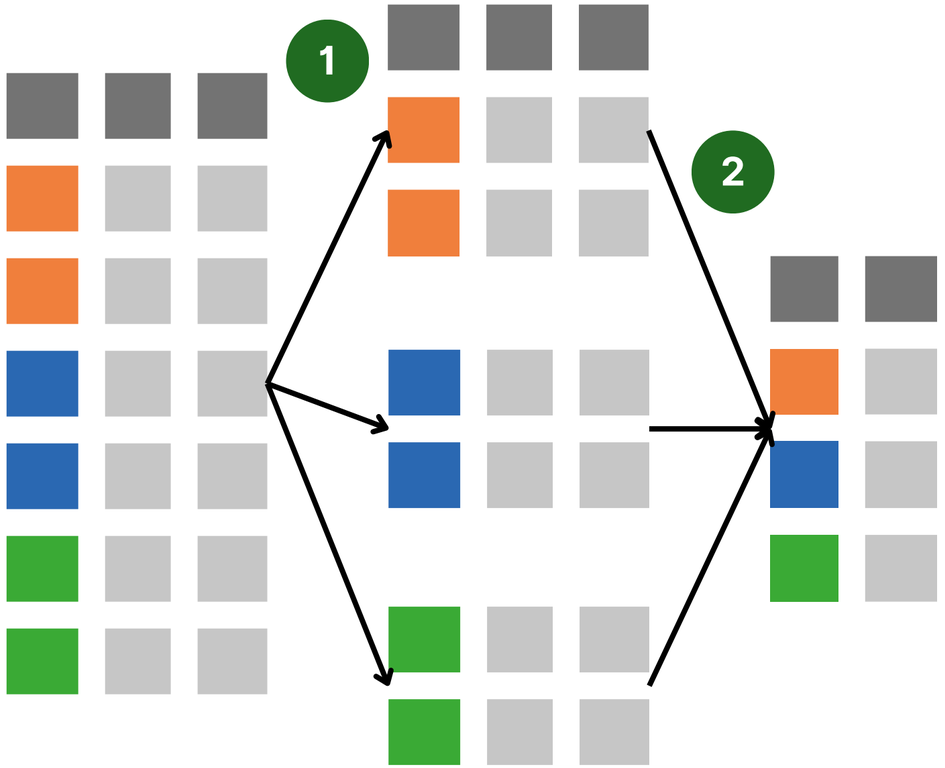
Summarising data

UNIVERSITY OF BIRMINGHAM

Grouping data isn't useful unless we do something with such groups. Usually we would like to calculate summary statistics, such as the mean GDP per capita for each continent:

```
gdp_by_continents <- gapminder %>%
  group_by(continent) %>%
  summarize(mean_gdpPercap = mean(gdpPercap))
```

continent	mean_gdpPercap
Africa	2193.755
Americas	7136.110
Asia	7902.150
Europe	14469.476
Oceania	18621.609

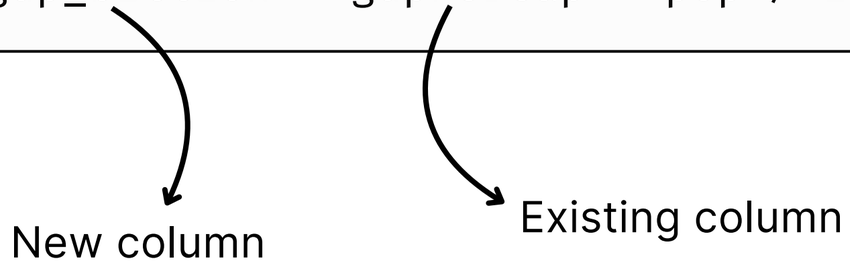


Adding/modifying data

UNIVERSITY OF BIRMINGHAM

To create a new column from existing columns we can use the **mutate** function:

```
gdp_by_continents <- gapminder %>%
    mutate(gdp_billion = gdpPercap * pop / 10^9)
```



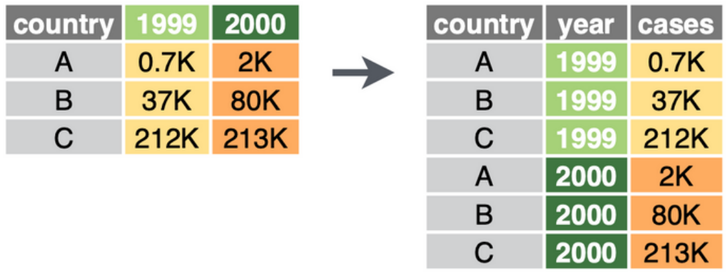
UNIVERSITY OF BIRMINGHAM

Chapter 13: Data frame manipulation with tidyr

Data tidying

UNIVERSITY OF BIRMINGHAM

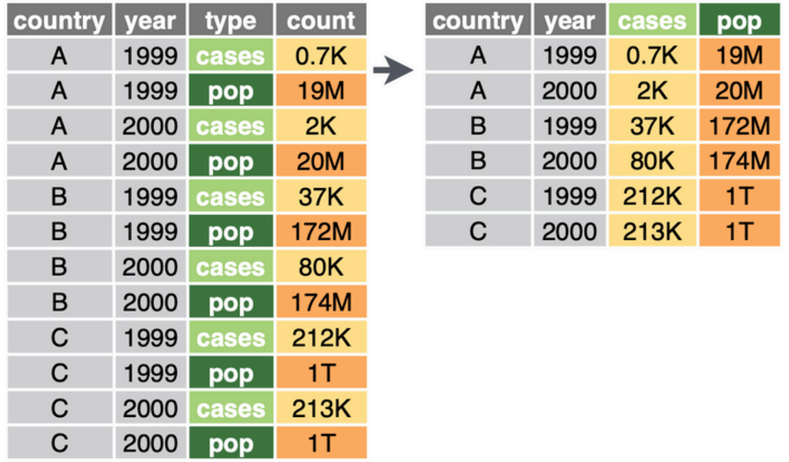
Often the data we receive is not in the format suitable for the tools we use. In this regard, **tidyr** is a package that allows us to tidy and clean the data. The package contains many functions for tidying data, however we will focus on ***pivoting***, i.e. converting between **long** and **wide** forms of data.



country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

Wide to long format
From tidyr cheatsheet



country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

Long to wide format
From tidyr cheatsheet

- **Wide format**: Each variable has its own column. This format is often easier for humans to read.
- **Long format**: Each row represents a single observation, and variables are stored in a key-value pair format. This format is often easier for analysis and visualization.

Example

UNIVERSITY OF BIRMINGHAM

Firstly install tidyr if you haven't already done so: `install.packages("tidyr")`

Then load the package in the current session: `library(tidyr)`

Next, let's create the following data frame (wide format):

```
student_grades <- data.frame(
    student = c("Luigi", "Elisabetta", "Sebastiano"),
    math = c(90, 85, 92),
    italian = c(88, 89, 95),
    history = c(78, 82, 80)
)
```

student	math	italian	history
Luigi	90	88	78
Elisabetta	85	89	82
Sebastiano	92	95	80

Example - wide to long data

UNIVERSITY OF BIRMINGHAM

The pivot_longer() function is used to convert wide data into long format:

```
student_grades_long <- student_grades %>%
  pivot_longer(
    cols = c(math, italian, history),
    names_to = "subject",
    values_to = "score"
  )
```

- Columns to pivot (all except 'student')
- Name of the new column for variable names
- Name of the new column for values

student	math	italian	history
Luigi	90	88	78
Elisabetta	85	89	82
Sebastiano	92	95	80

pivot_longer()

student	subject	score
Luigi	math	90
Luigi	italian	88
Luigi	history	78
Elisabetta	math	85
Elisabetta	italian	89
Elisabetta	history	82
Sebastiano	math	92
Sebastiano	italian	95
Sebastiano	history	80

Example - long to wide data

UNIVERSITY OF BIRMINGHAM

The pivot_wider() function is used to convert long data into wide format:

```
student_grades_wide <- student_grades_long %>%
  pivot_wider(
    names_from = subject,
    values_from = score
  )
```

→ Column to get new variable names from

→ Column to get values from

student	subject	score
Luigi	math	90
Luigi	italian	88
Luigi	history	78
Elisabetta	math	85
Elisabetta	italian	89
Elisabetta	history	82
Sebastiano	math	92
Sebastiano	italian	95
Sebastiano	history	80

pivot_wider()

student	math	italian	history
Luigi	90	88	78
Elisabetta	85	89	82
Sebastiano	92	95	80

UNIVERSITY OF BIRMINGHAM

Chapter 14: Producing reports with knitr

Create reports

UNIVERSITY OF BIRMINGHAM

As researchers we often need to create reports to summarise our findings to stakeholders. Usually a Word document or a PowerPoint presentation is the go-to approach. However, for reproducibility purposes and to keep the analysis close to the reports being generated, we can use R Markdown and knitr to produce quality reports.

R Markdown is a markup language for creating formatted text with both text and code in the same document.

knitr is an engine for dynamic report generation with R.

Components of an R Markdown document

UNIVERSITY OF BIRMINGHAM

```
---
title: "Gapminder dataset"
author: "Alessandro Rossi"
date: "25/01/2025"
output: html_document
---

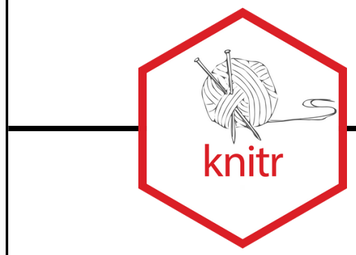
## A title

- A list item (1)
- A list item (2)

**Bold** and _italic_ words.

```{r}
library(dplyr)
library(gapminder)
library(ggplot2)

ggplot(
  data = gapminder,
  mapping = aes(x = gdpPercap, y = lifeExp)
) +
  geom_point()
```
```



Gapminder dataset

Alessandro Rossi

25/01/2025

A title

- A list item (1)
- A list item (2)

Bold and *italic* words.

```
library(dplyr)
```

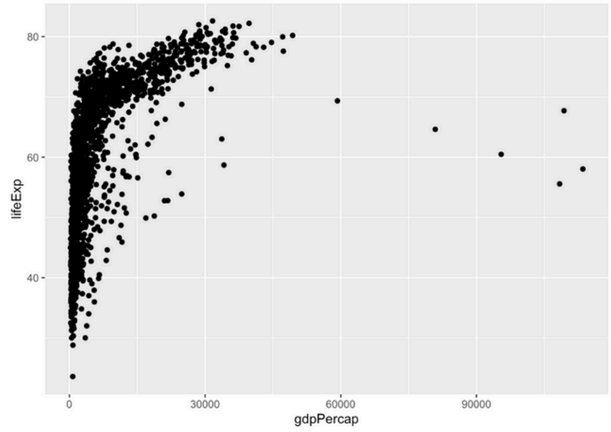
```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(gapminder)
library(ggplot2)

ggplot(data = gapminder, mapping = aes(x = gdpPercap, y = lifeExp)) +
  geom_point()
```



UNIVERSITY OF BIRMINGHAM

Chapter 15: Writing good software

Writing good software

UNIVERSITY OF BIRMINGHAM

Writing good software for research involves a combination of clear design principles, disciplined coding practices, and a focus on maintainability, readability, and reproducibility.

Structure your project folder

A good structure allows you and collaborators to better find data and organise files.

Make code readable

Yourelf and other people need to be able to easily read your code.

Document your code

Include comments that tell you *why* you're solving a problem, and *what* problem that is.

Keep your code modular

Split your code based on functionalities.

Test your code

Test your code thoroughly to make sure it does what it's supposed to do.

Don't repeat yourself

If you find yourself repeating parts of code then collect them in a function and reuse it.

UNIVERSITY OF BIRMINGHAM

Additional resources

Additional resources

UNIVERSITY OF BIRMINGHAM

- R cheat sheet (collection of the most important features in two pages)
- dplyr cheat sheet
- tidyr cheat sheet
- R Codecademy course
- 30 day roadmap to learn R
- Introduction to statistics in R