`NOTES.md` - `R` lesson

These notes are for the tutor on day one of the two-day `R` Software Carpentry course, taught 5th December 2017 at the University of St Andrews.

THINGS TO REMEMBER

To clear a console environment in `R`:

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

Start the slides

TITLE

- Programming in `R`: Part One

ETHERPAD

- Please use the Etherpad for the course

LEARNING OBJECTIVES

- We're being **QUITE AMBITIOUS**, and covering a few things, but we **HAVE TWO SESSIONS**, so should be OK
- We'll be covering the fundamentals of `R` and `RStudio` - how to **GET AROUND `R` AND WHAT `RStudio` CAN DO**
- You'll come away with fundamental knowledge applicable to **PROGRAMMING IN ANY LANGUAGE**
- We'll be learning some of the best practices for writing and organising your code (**FOLLOWING ON FROM GIT**) and for reproducible computing
- Finally, we'll be going through how to use these skills to be **MORE EFFECTIVE WITH DATA ANALYSIS** in `R`

SECTION 01: Introduction to `R` and `RStudio`

SLIDE: Learning Objectives

- We're going to talk in general terms about **WHAT `R` and `RSTUDIO` are**.
- We're also going to discuss **WHAT MAKES THEM DIFFERENT FROM SOME OTHER TOOLS** you might already use for similar tasks, like `Excel`

SLIDE: What is `R`?

- `R` is a **PROGRAMMING LANGUAGE**, and the **SOFTWARE** that runs programs written in that language.
- `R` is **AVAILABLE ON ALL PLATFORMS**
- This can sometimes be confusing - **IF AT ANY POINT I AM UNCLEAR, PLEASE ASK!**

- **HAS ANYONE IN THE ROOM USED `R` BEFORE? - GREEN STICKY**

  - If anyone has used `R`, please could you be available to help one of your neighbours who has not, if they have any questions. Look around you - if there's someone nearby with a green sticky, say 'hi'!

- **WHY USE OR TEACH `R`?**

  - `R` is **FREE**, and very **WIDELY-USED** across a range of disciplines.
  - There are **MANY USEFUL PACKAGES** or data analysis and statistics
  - It has excellent **GRAPHICS CAPABILITIES**
  - There is an international, friendly **COMMUNITY** across a range of disciplines, so it's easy to find local and online support

---

SLIDE: But I already know `Excel`

- I'm **NOT HERE TO CRITICISE `EXCEL`**. `Excel` is brilliant at what it's meant to do. It's **POWERFUL** and **INTUITIVE**.
- But **`R` HAS MANY ADVANTAGES FOR REPRODUCIBLE AND COMPLEX ANALYSIS**

- A key thing for reproducibility is that it **SEPARATES DATA FROM ANALYSIS**.

  - In `R`, when you do anything to the original data, that original data remains unmodified (unless you overwrite the file).
  - **POINT OF GOOD PRACTICE: RAW DATA SHOULD BE READ-ONLY**
  - In `Excel` however it's easy to overwrite data with copy-and-paste (and many bad things have happened as a result) - see Mike Croucher's talk for examples.

- Because **YOUR ANALYSIS IN `R` IS A PROGRAM**, every step is written down explicitly, and is transparent and understandable by someone else.

  - In `Excel`, there is no clear record of where you moved your mouse, or what you copied and pasted, and it's not immediately obvious how your formulas work.

- `R` code is **EASY TO SHARE AND ADAPT**, and to apply again to a different or a modified input dataset. It's easy to publish the analyses via online resources, such as GitHub.

- `R` code can also be **RUN ON EXTREMELY LARGE DATASETS**, quickly. That's much harder in `Excel`.

SLIDE: What is `RStudio`?

- `RStudio` is an **INTEGRATED DEVELOPMENT ENVIRONMENT** - which is to say it's a very powerful tool for writing and using `R` and programs in the `R` language.
    - It's available on **ALL PLATFORMS**
    - It's available as a webserver, too.

- On the left is a screenshot **WHILE I WAS WRITING THIS PRESENTATION IN RSTUDIO** on a Mac

- On the right is the Windows version, with an **EXAMPLE ANALYSIS AND VISUALISATION**

- You can use it to **INTERACT WITH `R` DIRECTLY TO EXPERIMENT WITH DATA**

- It has a **CODE/SCRIPT EDITOR FOR WRITING PROGRAMS**

- It has tools for **VISUALISING DATA**

- It has built-in **GIT INTEGRATION FOR MANAGING YOUR PROJECTS**

SECTION 02: Getting To Know `RStudio`

SLIDE: Learning Objectives

- We need to get some **FAMILIARITY WITH OUR WORKING ENVIRONMENT**: the `RStudio` IDE
- Develop some **BEST PRACTICES FOR PROJECT MANAGEMENT** in general, and in `RStudio`
- We'll **BUILD UPON THE `GIT` LESSON** because `RStudio` integrates naturally with `git` to keep all your analysis and code under version control.
- We'll introduce `R` syntax, and **SEE HOW `R` REPRESENTS DATA** and how to **PROGRAM IN `R`**
- If we have time we'll deal with **MANAGING SOME OF THE MANY USEFUL PACKAGES** available in `R`

SLIDE: `RStudio` overview - Interactive Demo

- **START `RStudio`** (click icon/go into start menu and select RStudio/etc.)
    - **CHECK EVERYONE CAN START `RSTUDIO`**

Red sticky for a question or issue　　Green sticky if complete

- **REMIND PEOPLE THEY CAN USE RED/GREEN STICKIES AT ANY TIME**
- You should see **THREE PANELS**
  - Interactive `R` console: **you can type here and get instant feedback**
  - Environment/History window
  - Files/Plots/Packages/Help/Viewer: **interacting with files on the computer, and viewing help and some output**
- We're going to use `R` in the interactive console to get used to some of the features of the language, and `RStudio`. **DEMO CODE: ASK PEOPLE TO TYPE ALONG**
  - **THE RIGHT ANGLED BRACKET IS A PROMPT: `R` EXPECTS INPUT**
  - **TYPE THE CALCULATION, THEN PRESS RETURN**

```
> 1 + 100
[1] 101
> 30 / 3
[1] 10
```

- **RESULT IS INDICATED WITH A NUMBER `[1]`** this indicates the line with output in it
- If you type an **INCOMPLETE COMMAND**, `R` will wait for you to complete it
  - **DEMO CODE**

```
> 1 +
+
```

- The **PROMPT CHANGES TO `+` WHEN `R` EXPECTS MORE INPUT**
- You can either complete the line, or use `Esc` (`Ctrl-C`) to exit

```
> 1 +
+ 6
[1] 7
> 1 +
+

>
```

- `R` obeys the usual **PRECEDENCE OPERATIONS** (`(`, `**`/`^`, `/`, `*`, `+`, `-`)
  - **DEMO CODE: NOTE SPACES AROUND OPERATORS**

```
> 3 + 5 * 2
[1] 13
> (3 + 5) * 2
[1] 16
> 3 + 5 * 2 ^ 2
[1] 23
> 3 + 5 * (2 ^ 2)
[1] 23
```

- **ARROW KEYS RECOVER OLD COMMANDS**
- **THE `HISTORY` TAB SHOWS ALL COMMANDS USED**
- `R` will report in **SCIENTIFIC NOTATION**
    - **CHECK THAT EVERYONE KNOWS WHAT SCIENTIFIC NOTATION IS**

Red sticky for a question or issue

Green sticky if complete

```
> 2 / 1000
[1] 0.002
> 2 / 10000
[1] 2e-04
> 5e3
[1] 5000
```

- `R` has many **STANDARD MATHEMATICAL FUNCTIONS**
- **FUNCTION SYNTAX**
    - type the function name
    - open parentheses
    - type input value
    - close parentheses
    - press return
    - **DEMO CODE**

```
> sin(1)
[1] 0.841471
> log(1)
[1] 0
> log10(10)
[1] 1
> log(10)
[1] 2.302585
```

- How do we learn more about a function, or the difference between `log()` and `log10()` ?

- **USE `R` BUILT-IN HELP**
  - Type `?` then the function name
  - Scroll to the bottom of the page to find example code

```
> ?log
```

- This brings up help in the **HELP WINDOW**
- You can also use the **SEARCH BOX** at the top of the help window (try `sin()`)
- If you're not sure about spelling, the editor has **AUTOCOMPLETION** which will suggest all possible endings for something you type (try `log`) - **USE TAB TO SEE AUTOCOMPLETIONS**

- We can do **COMPARISONS** in `R`
  - Comparisons return `TRUE` or `FALSE`. **DEMO CODE**
  - **NOTE:** when comparing numbers, it's better to use `all.equal()` (*machine numeric tolerance*) **ASK IF THERE'S ANYONE FROM MATHS/PHYSICS**

```
> 1 == 1
[1] TRUE
> 1 != 2
[1] TRUE
> 1 < 2
[1] TRUE
> 1 <= 1
[1] TRUE
> 1 > 0
[1] TRUE
> 1 >= -9
[1] TRUE
> all.equal(1.0, 1.0)
[1] TRUE
> all.equal(1.0, 1.1)
[1] "Mean relative difference: 0.1"
> all.equal(pi-1e-7, pi)
[1] "Mean relative difference: 3.183099e-08"
> all.equal(pi-1e-8, pi)
[1] TRUE
> pi-1e-8 == pi
[1] FALSE
```

SLIDE: Challenge 01

Red sticky for a question or issue

Green sticky if complete

SLIDE: Variables

- **VARIABLES** are critical to programming in general, and also to working in `R`
- Variables are like **NAMED BOXES**
  - Like a box, they **HOLD THINGS**
  - When we make reference to a box's name, we **MEAN THE THING THE BOX CONTAINS**
- Here, the box is called `Name`, and it contains the word `Samia`
- When we refer to the box, we call it `Name`, and we might ask questions like:
  - "What is the length of `Name`?", meaning "What is the length of the word in the box called `Name`?" (answer: 5)

SLIDE: Variables - Interactive Demo

- This is a very important concept, so we're going to go through some practical examples, to reinforce it
  - In `R`, variables are assigned with the **ASSIGNMENT OPERATOR** `<-`
  - We will assign the value `1/40` to the variable `x`
  - **DEMO CODE**

```
> x <- 1 / 40
```

- At first, nothing seems to happen, but we can see that the variable `x` now exists, and contains the value `0.025` - a **DECIMAL APPROXMATION** of the fraction `1/40`

```
> x
[1] 0.025
```

- **CLICK ON THE ENVIRONMENT WINDOW**
  - You should see that `x` is defined, there
- The *Environment* window in `RStudio` tells you the name and content of every variable currently active in your `R` session.
- This **VARIABLE CAN BE USED ANYWHERE THAT EXPECTS A NUMBER**
  - such as an argument to a function

```
> log(x)
[1] -3.688879
> sin(x)
[1] 0.0249974
> x + x
[1] 0.05
> 2 * x
[1] 0.05
> x ^ 2
[1] 0.000625
```

- We can **REASSIGN VALUES TO VARIABLES**
  - **MONITOR THE VALUE OF `x` IN THE ENVIRONMENT WINDOW**
  - We can also assign a variable to itself, to modify the variable

```
> x <- 100
> x <- x + 5
```

- We can assign **ANY KIND OF VALUE** to a variable
  - Including **STRINGS** - i.e. sets of characters

```
> name <- "Samia"
> name
[1] "Samia"
```

- But **BE CAREFUL** - `R` is not always intuitive
  - Strings in particular may not work the way you expect

```
> length(name)
[1] 1
> nchar(name)
[1] 5
```

SLIDE: Functions

- You've now used some **functions**. These are like "canned scripts"
- Functions have **THREE MAIN PURPOSES**
  - They **ENCAPSULATE COMPLEX TASKS** - you don't need to worry about how a sine is calculated, just that you give the function a value, and it tells you what the sine is.
  - They **MAKE CODE MORE READABLE** - by dividing code into logical operations, represented by short names, the code is easier to read
  - They also **MAKE CODE MORE REUSABLE** - you don't need to write the routine for finding a square root every time you want one, you just need to call the `sqrt()` function

- **ALL THE FUNCTIONS YOU'VE SEEN ARE BUILT-IN**, the so-called `base` functions
- **OTHER FUNCTIONS FOR SPECIFIC TASKS CAN BE BROUGHT IN, THROUGH `libraries`**
- Functions usually **TAKE ARGUMENTS** (input), e.g. `sqrt(4)` - the `4` is an argument
- Functions often **RETURN** values (output), e.g. `sqrt(4)` **returns** the value `2`

SLIDE: Getting Help for Functions

- **DEMO IN CONSOLE**

```
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
> ?sqrt
> help(sqrt)
> ??sqrt
> help.search("sqrt")
> help.search("categorical")
> vignette(two-table)
Error in vignette(two - table) : object 'two' not found
> vignette("two-table")
```

SLIDE: Removing Variables

- To remove **ONE OR MORE SPECIFIED VARIABLES**, use `rm()`
  - `ls()` **IS A FUNCTION THAT LISTS VARIABLES** (like the Environment tab)
  - **DEMO CODE**

```
> x <- 1
> y <- 2
> z <- 3
> ls()
[1] "x" "y" "z"
> rm(x)
> ls()
[1] "y" "z"
> rm(y, z)
> ls()
character(0)
```

SLIDE: Challenge 02

Solution:

`mass <- 47.5` This will give a value of 47.5 for the variable mass

`age <- 122` This will give a value of 122 for the variable age

`mass <- mass * 2.3` This will multiply the existing value of 47.5 by 2.3 to give a new value of 109.25 to the variable mass.

`age <- age - 20` This will subtract 20 from the existing value of 122 to give a new value of 102 to the variable age.

Red sticky for a question or issue

Green sticky if complete

SLIDE: Good Variable Names

- Good variable names **SHOULD HELP YOU DESCRIBE WHAT'S GOING ON**
    - Helpful **FOR YOU AND FOR OTHERS**
- It's better to **avoid using names that already exist**
- Use a **CONSISTENT NAMING STYLE**
- Rules for **VARIABLE NAMES DIFFER BETWEEN LANGUAGES**
- **IN `R`**
    - You can only use **LETTERS, NUMBERS, UNDERSCORES, AND PERIODS**
    - You **can't start variable names with a number**
    - You **can't use whitespace in a variable name**

SLIDE: Good Project Management Practices

- There is **NO SINGLE GOOD WAY TO ORGANISE A PROJECT**
    - It's important to find something that **WORKS FOR YOU**
    - But it's **ALSO IMPORTANT THAT IT WORKS FOR COLLABORATORS**
    - Some **PRINCIPLES MAKE THINGS EASIER FOR EVERYONE**
- Using a **SINGLE WORKING DIRECTORY PER PROJECT/ANALYSIS**
    - **NAME IT AFTER THE PROJECT**
    - **EASY TO PACKAGE UP** the whole directory and move it around, or share it **- OR PUBLISH**

    ALONGSIDE YOUR PAPER
    - **NO NEED TO HUNT AROUND THE WHOLE DISK** to find relevant or important files.
    - You can use **RELATIVE PATHS** that will always work, so long as you work within the project directory

- Treat your **RAW DATA AS READ-ONLY**
    - Establishes **PROVENANCE** and **ENABLES REANALYSIS**
    - Keep in **SEPARATE SUBFOLDER**
- **CLEAN THE DATA PROGRAMMATICALLY** (part of the analysis chain)
    - Remove/fill in null values, etc. - whatever is appropriate
    - **KEEP CLEANED DATA SEPARATE FROM RAW** - like food hygiene
- **GENERATED OUTPUT IS DISPOSABLE**
    - This means **ANYTHING GENERATED AUTOMATICALLY BY YOUR CODE/ANALYSIS**
    - If a file can be generated by scripts/code in your project, no need to put it under version control

---

SLIDE: Example Directory Structure

- This is **ONE WAY TO STRUCTURE YOUR WORKING DIRECTORY**
    - It's a good starting point, but something else might be more appropriate for your own work
- **`WORKING DIR/`** is the *root* directory of the project.
    - Everything related to the project (subdirectories of data, scripts and figures; `git` files; configuration files; notes to yourself; whatever)
- **`data/`** is a subdirectory for storing data
    - raw data only, or raw and intermediate data? **YOUR DECISION**
    - `data/raw`, `data/intermediate` - **USE SUBFOLDERS WHEN SENSIBLE**
- **`data_output/`** could be a place to write the analysis output (`.csv` files etc.)
- **`documents/`** is a place where notes, drafts, and explanatory text could be stored
- **`fig_output/`** could be a place to write graphical output of the analysis (keep separate from tables)
- **`scripts`** might be where you would choose to keep executable code that automates your analysis
- The important thing is that the structure is **SELF-EXPLANATORY WHERE POSSIBLE**

SLIDE: Project Management in `RStudio`

- `RStudio` **TRIES TO BE HELPFUL** and provides the 'Project' concept
    - Keeps **ALL PROJECT FILES IN A SINGLE DIRECTORY**
    - **INTEGRATES WITH `GIT`**
    - Enables switching between projects within `RStudio`
    - Keeps project histories
- **INTERACTIVE DEMO**
- **CREATE PROJECT**
- Click `File` -> `New Project`
    - Options for how we want to create a project: brand new in a new working directory; turn an existing directory into a project; or checkout a project from `GitHub` or some other repository
- Click `New Directory`
    - Options for various things we can do in `RStudio`. Here we want `New Project`
- Click `New Project`
    - We are asked for a directory name. **ENTER `swc-r-lesson`**
    - We are asked for a parent directory. **PUT YOURS ON THE DESKTOP; STUDENTS CAN CHOOSE ANYWHERE SENSIBLE**
    - **SELECT `Create a git repository`** - this will create and initialise a `git` repository, just for this project
- Click `Create Project`
- **YOU SHOULD SEE AN EMPTY-ISH `RSTUDIO` WINDOW**
- **INSPECT PROJECT ENVIRONMENT**
- First, **NOTE THE WINDOWS**: editor; environment; files
- **EDITOR** is empty
- **ENVIRONMENT** is empty
- **FILES** shows
    - **CURRENT WORKING DIRECTORY** (see breadcrumb trail)
    - **TWO FILES**: `*.Rproj` - information about your project; `.gitignore` - your project's `.gitignore` file (remember the `git` lesson?)

- **CLICK ON `GIT` TAB**
  - We see that the files are not yet added to the project (status is `?`)
  - **CHECK BOXES TO STAGE FILES**
  - **CLICK ON `DIFF` TO SEE CHANGES** (note colours, lines, etc.)
  - **CLICK ON `COMMIT` TO COMMIT CHANGES**
  - **WRITE COMMIT MESSAGE** (add .gitignore and R project files to repo)
  - **CLICK `COMMIT`** (explain message)
- **CREATE DIRECTORIES IN PROJECT**
- **Create directoris called `scripts` and `data`**
  - Click on `New Folder`
  - Enter directory name (`scripts`)
  - Note that the directory now exists in the `Files` tab (but not in the `git` tab, as the directory is empty)
  - Do the same for `data/`
- **NOTE THAT WE WILL NOW POPULATE THE DIRECTORY**

SLIDE: Working in RStudio

- `RStudio` offers **SEVERAL WAYS TO WRITE CODE**
  - We'll not see all of them today
  - You've seen **DIRECT INTERACTION IN THE CONSOLE** (entering variables)
  - `RStudio` also has an editor for writing scripts, notebooks, markdown documents, and Shiny applications (**EXPLAIN BRIEFLY**)
  - It can also be used to write plain text
- **INTERACTIVE DEMO OF `R` SCRIPT**
- Click on `File` -> `New File` -> `Text File`. **NOTE THAT THE EDITOR WINDOW OPENS**
- Enter the following text, and **EXPLAIN CSV**
  - plain text file
  - one row per line
  - column entries separated by commas
  - first row is header data
  - **NEEDS A BLANK LINE AT THE END**
  - **DATA DESCRIBES CATS**

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

- **SAVE THE FILE AS `data/feline_data.csv`**
  - Click on disk icon
  - Navigate to `data/` subdirectory
  - Enter filename `feline_data.csv`
- **NOTE CHANGES IN `GIT` TAB**
  - The `data` directory appears!
  - Click on `Staged` and **THE FILE APPEARS**
  - Click `Commit` and add a commit string, e.g. "add cat dataset"
- **CLOSE THE EDITOR FOR THAT FILE**
- Click on `File` -> `New File` -> `R Script`.
- **EXPLAIN COMMENTS** while entering the code below
  - **COMMENTS ANNOTATE YOUR CODE**: reminders for you, and information for others
- **EXPLAIN `read.csv()`**
  - `read.csv()` is a **FUNCTION** that reads data from a **CSV-FORMAT FILE** into a variable in `R`

```
# Script for exploring data structures

# Load cat data as a dataframe
cats <- read.csv(file = "data/feline_data.csv")
```

- **SAVE THE SCRIPT**
  - Click on `File` -> `Save`
  - Navigate to the `scripts/` subdirectory
  - Enter filename `data_structures` (**EXTENSION IS AUTOMATICALLY APPLIED**) **NOTE CHANGES IN `GIT` TAB**
  - The `scripts` directory appears!
  - Click on `Staged` and the file appears
  - Click on `Commit` and add a commit string, e.g. "add data structures script"
- **DO YOU SEE THE VARIABLE IN THE ENVIRONMENT?**
  - **NO** - because the code hasn't been executed, only written.

- **RUN THE SCRIPT**
  - Click on `Source` and **NOTE THIS RUNS THE WHOLE SCRIPT**
- Go to the `Environment` tab
  - **NOTE THE DATA WAS LOADED IN THE VARIABLE `cats`**
  - Note that there is a description of the data (3 obs. of 3 variables)
  - **CLICK ON THE VARIABLE AND NOTE THAT THE TABLE IS NOW VISIBLE** - this is helpful
  - **YOU CANNOT EDIT THE DATA IN THIS TABLE** - you can sort and filter, but not modify the data. This **ENFORCES GOOD PRACTICE** (compare to Excel).

SLIDE: 03. A First Analysis in `RStudio`

SLIDE: Learning Objectives

- We're going to cover some ways of loading data into an `R` project/analysis
- We'll **EXPLORE SOME FUNCTIONS FOR SUMMARISING DATA**
- Sometimes we want to use only a portion of our data, and we'll see some **WAYS OF INDEXING DATASETS**
- We'll also look at some ways to **PLOT DATA IN R**, using the built-in *base graphics*

SLIDE: Our Task

- We've got some medical data relating to a new treatment for arthritis
- There are some measurements of patient inflammation, taken over a period of days post-treatment for each patient
- We've been **ASKED TO PRODUCE A SUMMARY AND SOME GRAPHS**
- **DOWNLOAD THE FILE FROM THE LINK TO `data/`**
- **EXPLAIN THE LINK IS ON THE ETHERPAD PAGE** (no need to type!)
- **DEMO THIS**
  - **NOTE:** A new directory is created *within* `data`, called `data`. **THIS IS UNTIDY, SO LET'S CLEAN**
  - **COPY ALL FILES BEGINNING WITH `inflammation` TO THE PARENT FOLDER**
  - **THEN DELETE THE SUBFOLDER AND ZIP FILE**
  - **ADD THE DATA FILES TO THE REPO** (can shift-click, here)
- **CHECK EVERYONE'S READY TO PROCEED**

SLIDE: Loading Data - Interactive Demo

- We've already created some cat data manually, but **THIS IS UNUSUAL** - most data comes in the form of plain text files

START DEMO * **INSPECT DATA IN FILES WINDOW** * Click on filename, and select `View File` * Note: **THERE IS NO HEADER** and **THERE ARE NO ROW NAMES** * Ask: **IS THIS WELL-FORMATTED DATA?** * I happen to know that there is **one row per patient, and the columns are days, in turn, post-treatment**, and **measurements are inflammation levels** * **WHAT IS THE DATA TYPE** * Tabular, with **EACH COLUMN SEPARATED BY A COMMA**, so **CSV** * **IN THE CONSOLE** use `read.csv()` to read the data in * Note: **IF WE DON'T ASSIGN THE RESULT TO A VARIABLE WE JUST SEE THE DATA** * **CREATE A NEW SCRIPT** * Click the **triangle next to the new document icon** * Add the code and **SAVE AS** **`scripts/inflammation`** (`RStudio` adds the extension) * See that the file appears in `Files` and `Git` windows

```
# Preliminary analysis of inflammation in arthritis patients

# Load data (no headers, CSV)
data <- read.csv(file = "data/inflammation-01.csv", header = FALSE)
```

- **INSPECT THE DATA**
  - `Source` the script
  - Check the `Environment` window: 60 observations (patients) of 40 variables (days)
  - **CLICK ON** **`data`**
  - **COLUMN HEADERS ARE PROVIDED:** **`Vn`** for *variable n*
  - `dim()` - *dimensions* of data: rows X columns
  - `length()` - number of columns in the table
  - `ncol()` - number of columns in the table
  - `nrow()` - number of rows in the table

```
> head(data, n = 2)
  V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20 V21 V22 V23
1  0  0  1  3  1  2  4  7  8   3   3   3  10   5   7   4   7   7  12  18   6  13  11
2  0  1  2  1  2  1  3  2  2   6  10  11   5   9   4   4   7  16   8   6  18   4  12
  V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38 V39 V40
1   4   6   8   8   4   4   5   7   3   4   2   3   0   0
2  11   5  11   3   3   5   4   4   5   5   1   1   0   1
> dim(data)
[1] 60 40
> length(data)
[1] 40
> ncol(data)
[1] 40
> nrow(data)
[1] 60
```

SLIDE: Challenge 03

SOLUTION

```
read.csv(file='file.csv', sep=';', dec=',')
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Indexing Data

- We can refer to an element in our dataset by *indexing* it
  - We **LOCATE A SINGLE ELEMENT as** `[row, column]` **in square brackets**

```
> ncol(data)
[1] 40
> data[1,1]
[1] 0
> data[50,1]
[1] 0
> data[50,20]
[1] 16
> data[30,20]
[1] 16
```

- To get a **RANGE OF VALUES**, use the `:` separator to mean 'to':

```
> data[1:4, 1:4]   # rows 1 to 4; columns 1 to 4
  V1 V2 V3 V4
1  0  0  1  3
2  0  1  2  1
3  0  1  1  3
4  0  0  2  0
> data[30:32, 20:22]
   V20 V21 V22
30  16  14  15
31  16  13   7
32   9  19  15
```

- To get a **WHOLE ROW OR COLUMN**, leave that entry blank

```
> data[5,]
  V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20 V21 V22 V23
5  0  1  1  3  3  1  3  5  2   4   4   7   6   5   3  10   8  10   6  17   9  14   9
  V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38 V39 V40
5  12   6   7   7   9   6   3   2   2   4   2   0   1   1
> data[,16]
 [1]  4  4 15  8 10 15 13  9 11  6  3  8 12  3  5 10 11  4 11 13 15  5 14 13  4  9 13
[32]  3 15  4 15 11  7 10 15  6  5  6 15 11 15  6 11 15 14  4 10 15 11  6 13  8  4 13
```

SLIDE: Summary Functions - Interactive Demo

- `R` **was designed for data analysis**, so has many built-in functions for analysing and describing data
    - **TALK THROUGH CODE IN CONSOLE**

```
> max(data)
[1] 20
> max(data[2,])
[1] 18
> max(data[,7])
[1] 6
> min(data[,7])
[1] 1
> mean(data[,7])
[1] 3.8
> median(data[,7])
[1] 4
> sd(data[,7])
[1] 1.725187
```

SLIDE: Challenge 04

```
> animal <- c('m', 'o', 'n', 'k', 'e', 'y')
> animal[1:3]
[1] "m" "o" "n"
> animal[4:6]
[1] "k" "e" "y"
> animal[3:1]
[1] "n" "o" "m"
> animal[-1]
[1] "o" "n" "k" "e" "y"
> animal[-4]
[1] "m" "o" "n" "e" "y"
> animal[-1:-4]
[1] "e" "y"
> animal[-1:4]
Error in animal[-1:4] : only 0's may be mixed with negative subscripts
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Repetitive Calculations - Interactive Demo

- We might want to **CALCULATE MEAN INFLAMMATION FOR EACH PATIENT**, but doing it the way we've just seen is tedious and slow.
- Happily **COMPUTERS WERE INVENTED TO SAVE US THE HASSLE**
- We could automate this task in any of several ways available in `R`
    - What we'd like to do is **APPLY A FUNCTION TO EACH ROW**
    - `R` has an `apply()` function exactly for this
    - **IN THE CONSOLE**

```
> apply(X = data, MARGIN = 1, FUN = mean)
 [1] 5.450 5.425 6.100 5.900 5.550 6.225 5.975 6.650 6.625 6.525 6.775 5.800 6.225 5.
[16] 6.300 6.550 5.700 5.850 6.550 5.775 5.825 6.175 6.100 5.800 6.425 6.050 6.025 6.
[31] 6.175 6.350 6.725 6.125 7.075 5.725 5.925 6.150 6.075 5.750 5.975 5.725 6.300 5.
[46] 5.925 7.225 6.150 5.950 6.275 5.700 6.100 6.825 5.975 6.725 5.700 6.250 6.400 7.
```

- **IN OUR ANALYSIS SCRIPT** we want to assign these values to a variable, and **ALSO CALCULATE AVERAGE BY DAY**
    - So long as we provide arguments in the correct order, **WE DON'T NEED TO PROVIDE ARGUMENT NAMES** - true for most `R` functions

```r
# Calculate average inflammation by patient and day
avg_inflammation_patient <- apply(X = data, MARGIN = 1, FUN = mean)
avg_inflammation_day <- apply(data, 2, mean)
```

- **RUN THE LINES**
  - Note that the values appear in the Environment tab
- Like many common operations, there's an `R` function that's a shortcut
  - **IN THE CONSOLE**

```
> rowMeans(data)
 [1] 5.450 5.425 6.100 5.900 5.550 6.225 5.975 6.650 6.625 6.525 6.775 5.800 6.225 5.
[16] 6.300 6.550 5.700 5.850 6.550 5.775 5.825 6.175 6.100 5.800 6.425 6.050 6.025 6.
[31] 6.175 6.350 6.725 6.125 7.075 5.725 5.925 6.150 6.075 5.750 5.975 5.725 6.300 5.
[46] 5.925 7.225 6.150 5.950 6.275 5.700 6.100 6.825 5.975 6.725 5.700 6.250 6.400 7.
> colMeans(data)
        V1         V2         V3         V4         V5         V6         V7
 0.0000000  0.4500000  1.1166667  1.7500000  2.4333333  3.1500000  3.8000000  3.88333
       V10        V11        V12        V13        V14        V15        V16         V
 5.5166667  5.9500000  5.9000000  8.3500000  7.7333333  8.3666667  9.5000000  9.58333
       V19        V20        V21        V22        V23        V24        V25         V
11.5666667 12.3500000 13.2500000 11.9666667 11.0333333 10.1666667 10.0000000  8.66666
       V28        V29        V30        V31        V32        V33        V34         V
 7.2500000  7.3333333  6.5833333  6.0666667  5.9500000  5.1166667  3.6000000  3.30000
       V37        V38        V39        V40
 2.4833333  1.5000000  1.1333333  0.5666667
```

SLIDE: Base Graphics

- We're doing all this work to try to **GAIN INSIGHT INTO OUR DATA**
- **VISUALISATION IS A KEY ROUTE TO INSIGHT**
- `R` has many graphics packages - some of which produce extremely beautiful images, or are tailored to a specific problem domain
- The built-in graphics are known as *base graphics*
- They may not be as pretty, or as immediately suited for all circumstances as some other packages, but they are still very powerful

---

SLIDE: Plotting - Interactive Demo

- **IN THE SCRIPT**

- `R` 's `plot()` **FUNCTION IS GENERAL AND WORKS FOR MANY KINDS OF DATA**
- **RUN EACH LINE IN TURN**
- **NOTE WHERE PLOTS SHOW** (Plot window)
- Opportunity to note: **VARIABLES HELP READABILITY**
- **USE ARROW BUTTONS** to cycle through plots
- **COMMIT CHANGES WHEN DONE**

```
# Plot data summaries
# Average inflammation by patient
plot(avg_inflammation_patient)

# Average inflammation per day
plot(avg_inflammation_day)

# Maximum inflammation per day
max_inflammation_day <- apply(data, 2, max)
plot(max_inflammation_day)

# Minimum inflammation per day
plot(apply(data, 2, min))

# Show a historgram of average patient inflammation
hist(avg_inflammation_patient)
```

- **THE `hist()` FUNCTION PLOTS A HISTOGRAM OF INPUT DATA FREQUENCY/COUNT**
  - The choice of bin sizes/*breaks* could be improved
  - We need to **PROVIDE THE BOUNDARIES BETWEEN BINS**
  - **IN THE CONSOLE**

```
hist(avg_inflammation_patient, breaks=c(5, 6, 7, 8))
```

- We'd have to **TYPE IN A LOT OF NUMBERS** to get smaller breaks, which is **SLOW**
  - The `seq()` function generates a sequence of numbers for us

```
> seq(5, 8)
[1] 5 6 7 8
> hist(avg_inflammation_patient, breaks=seq(5, 8))
```

- We can **SET THE INTERVAL OF THE SEQUENCE**

```
> seq(5, 8, by=0.2)
 [1] 5.0 5.2 5.4 5.6 5.8 6.0 6.2 6.4 6.6 6.8 7.0 7.2 7.4 7.6 7.8 8.0
> hist(avg_inflammation_patient, breaks=seq(5, 8, by=0.2))
```

- **IN THE SCRIPT**

- Add a line with the histogram for average patient inflammation

```
# Show a historgram of average patient inflammation
hist(avg_inflammation_patient, breaks=seq(5, 8, by=0.2))
```

- **IN THE SCRIPT**
  - Demonstrate changing the input file
  - **CHANGING FILENAME IN A SCRIPT IS QUICKER THAN RETYPING ALL THE COMMANDS**

SLIDE: Challenge 05

```
# Plot standard deviation by day
plot(apply(data, 2, sd))
```

SLIDE: 04. Data Structures in `R`

SLIDE: Learning Objectives

- In this section, you'll be learning about the data types in `R` : **WHAT DATA IS**
- You'll also be learning about the data *structures*: **WHAT DATA IS BUILT INTO - HOW IT IS ARRANGED**
- And you'll also learn how to find out what type/structure a particular piece of data has
- Putting it together, you'll see how `R` 's data types and structures relate to the types of data that you work with, yourself.

SLIDE: Data Types and Structures in `R`

- `R` is **MOSTLY USED FOR DATA ANALYSIS**
- `R` is set up with key, core data types designed to help you work with your own data
- A lot of the time, `R` focuses on tabular data (like our cat example)
- **INTERACTIVE DEMO**

- **SWITCH TO THE CONSOLE** (Establish that `cats` is available as a variable)

- If you type `cats` , you get a nice tabular representation of your data

```
> cats
    coat weight likes_string
1 calico    2.1            1
2  black    5.0            0
3  tabby    3.2            1
```

- **THINK ABOUT THE DATA TYPES** Are they all the same?
  - **NO** `coat` is text; `weight` is some real value (in kg or pounds, maybe), and `likes_string` looks like it should be `TRUE` / `FALSE`
  - **DOES IT MAKE SENSE TO WORK WITH THEM AS IF THEY'RE THE SAME THING?** (No)
- **EXTRACT A COLUMN FROM A TABLE**
  - Use `$` notation in the console
  - **NOTE THE AUTOCOMPLETION**

```
> cats$weight
[1] 2.1 5.0 3.2
```

- **WHAT DID `R` RETURN?**
  - A *vector* (1D ordered collection) of numbers
- **WE CAN OPERATE ON THESE *VECTORS* **
  - *Vectors* are an important concept, and `R` is largely built so that operations on vectors are central to data analysis.

```
> cats$weight + 2
[1] 4.1 7.0 5.2
```

- **WHAT ABOUT OTHER COLUMNS?**

```
> cats$coat
[1] calico black  tabby
Levels: black calico tabby
```

- **WHAT DID `R` RETURN?**
  - A *vector* of *levels*
  - We'll talk about these in more detail shortly, but the key point is that `R` **DOESN'T THINK THEY'RE ONLY WORDS** - it **THINKS THEY'RE NAMED CATEGORIES OF OBJECT**. `R` is assuming that you mean to import *data*
  - We can operate on this vector, too (**EXPLAIN** `paste()`)

```
> paste("My cat is", cats$coat)
[1] "My cat is calico" "My cat is black"  "My cat is tabby"
```

- **WHAT HAPPENS NEXT?**

```
> cats$weight + cats$coat
[1] NA NA NA
Warning message:
In Ops.factor(cats$weight, cats$coat) : ‘+’ not meaningful for factors
```

- You probably already realised that wasn't going to work, because adding "calico" to "2.1" is nonsense.
- **THESE DATA TYPES ARE NOT COMPATIBLE** for addition
- `R` 's data types reflect the ways in which data is expected to interact
- **UNDERSTANDING HOW YOUR DATA MAP TO `R` 's DATA TYPES IS KEY**
  - It's very important to understand how `R` sees your data (you want `R` to see your data the same way you do)
  - Many problems in `R` come down to incompatibilities between data and data types.

SLIDE: What Data Types Do You Expect?

- **ASK THE STUDENTS**
  - What data types would you expect to see?
  - What data types do you think you would **WANT OR NEED**, from your own experience?
- **SPEND A COUPLE OF MINUTES ON THIS**
  - The difference between a *data type* and a *data structure*

SLIDE: Data Types in `R`

- `R` 's data *types* are *atomic*: they are **FUNDAMENTAL AND EVERYTHING ELSE IS BUILT UP FROM THEM**, like matter is built up from atoms
  - In particular, all the data *structures* are built up from data *types*
- There are only **FIVE DATA TYPES** in `R` (though one is split into two…)
  - **logical**: Boolean, True/False (also `1` / `0`)
  - **numeric**: anything that's a number on the number line; two types of number are supported: `integer` and `double` (real)
  - **complex**: complex numbers, defined on the 2D plane
  - **character**: text data - readable symbols
  - **raw**: binary data (we'll not be dealing with this)
- **LET'S LEARN A BIT MORE ABOUT THEM IN THE DEMO**

- **ENTER DEFINITIONS INTO THE SCRIPT**
    - Covering the major data types

```
# Some variables of several data types
truth <- TRUE
lie <- FALSE
i <- 3L
d <- 3.0
c <- 3 + 0i
txt <- "TRUE"
```

- **EXECUTE THE VARIABLE DEFINITIONS**
    - Select the definition lines
    - Click on `Run`
    - **OBSERVE THAT THE LINES ARE RUN IN THE CONSOLE**
    - **OBSERVE THAT THE VALUES ARE DEFINED IN THE ENVIRONMENT**
    - Note the difference between `Data` and `Values` in the environment
    - Note that the script is updated in the `Git` tab: commit the change

- **USE `typeof()` TO FIND THE TYPE OF A VARIABLE**

```
> typeof(i)
[1] "integer"
> typeof(c)
[1] "complex"
> typeof(d)
[1] "double"
```

- **TO TEST IF A DATA ITEM HAS A TYPE, USE `is.<type>()`**

```
> is.numeric(3)
[1] TRUE
> is.numeric(d)
[1] TRUE
> is.double(i)
[1] FALSE
> is.integer(d)
[1] FALSE
> is.numeric(txt)
[1] FALSE
> is.character(txt)
[1] TRUE
> is.character(truth)
[1] FALSE
> is.logical(truth)
[1] TRUE
```

- **THE INTEGER, COMPLEX, AND DOUBLE ARE EQUAL** even if they're not the same data type
    - numbers are comparable, regardless of data type

```
> i == c
[1] TRUE
> i == d
[1] TRUE
> d == c
[1] TRUE
```

- **INTEGER, COMPLEX AND DOUBLE ARE NOT ALL `numeric`** though

```
> is.numeric(i)
[1] TRUE
> is.numeric(c)
[1] FALSE
```

SLIDE: Challenge 06

- Let the students work for a couple of minutes, then demonstrate.

Red sticky for a question or issue

Green sticky if complete

SLIDE: FIVE COMMON `R` DATA STRUCTURES

- These are perhaps the five data types you'll come across most often in `R`
- We'll deal with them through examples
- **INTERACTIVE DEMO IN SCRIPT**

- **VECTORS**
    - These are the **MOST COMMON DATA STRUCTURE**
    - Vectors can contain **ONLY A SINGLE DATA TYPE** (*atomic vectors*)
    - **ADD CODE TO SCRIPT** then use `Run` to run in console
    - To create a vector **USE THE `c()` FUNCTION** (`c()` is `combine` ; use `?c`)
    - First we define an **ATOMIC VECTOR OF NUMBERS** - each element is an integer

```
# Define an integer vector
x <- c(10, 12, 45, 33)
```

- We can use some `R` functions to find out more about this variable

- **RUN CODE IN CONSOLE**

```
> length(x)
[1] 4
> typeof(x)
[1] "double"
> str(x)
 num [1:4] 10 12 45 33
```

- The `str()` function **REPORTS THE STRUCTURE OF A VARIABLE**
  - Here, `num` means 'numeric'; `[1:4]` means there are four elements; the elements are listed
  - **NOTE THAT THIS INFORMATION IS IN THE ENVIRONMENT TAB**

- **DEFINE A SECOND VECTOR IN THE SCRIPT**

```
# Define a vector
xx <- c(1, 2, 'a')
```

- In the Environment tab, you can see **THIS IS A CHARACTER VECTOR**

```
> length(xx)
[1] 3
> typeof(xx)
[1] "character"
> str(xx)
 chr [1:3] "1" "2" "a"
```

- **IS THE TYPE OF THE VECTOR WHAT YOU EXPECTED?**
  - This is one of the things that trips people up with `R` - they think their data is of one type, but `R` thinks it makes more sense to have it as another type

---

SLIDE: Challenge 07

- Let the students work for a couple of minutes, then demonstrate.

Red sticky for a question or issue   Green sticky if complete

---

SLIDE: Coercion

- *Coercion* is what happens when you **COVERT ONE DATA TYPE INTO ANOTHER**
- If `R` thinks it needs to, it will **COERCE DATA IMPLICITLY** without telling you

- There is a set order for coercion
    - `logical` can be coerced to `integer`, but `integer` cannot be coerced to `logical`
    - That's because `integer` can describe all `logical` values, but not *vice versa*
    - Everything can be represented as a `character`, so that's the fallback position for `R`
- **IF THERE'S A FORMATTING PROBLEM IN YOUR DATA, `R` MIGHT CONVERT THE TYPE TO COPE**
    - `R` will choose the simplest data type that can represent all items in the vector
- **INTERACTIVE DEMO IN CONSOLE** More useful things to do with vectors
- You can (usually) **COERCE VECTORS MANUALLY** with `as.<type>()`

```
> as.character(x)
[1] "10" "12" "45" "33"
> as.complex(x)
[1] 10+0i 12+0i 45+0i 33+0i
> as.logical(x)
[1] TRUE TRUE TRUE TRUE
> xx
[1] "1" "2" "a"
> as.numeric(xx)
[1]  1  2 NA
Warning message:
NAs introduced by coercion
> as.logical(xx)
[1] NA NA NA
```

- You can generate **NUMBER SEQUENCES** as vectors
    - The `seq()` function returns a vector
    - As does the `:` operator

```
> seq(10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(1, 10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(35, 40, by=0.5)
 [1] 35.0 35.5 36.0 36.5 37.0 37.5 38.0 38.5 39.0 39.5 40.0
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 5:8
[1] 5 6 7 8
```

- You can **APPEND ELEMENTS TO A VECTOR WITH `c()`**

```
> x
[1] 10 12 45 33
> c(x, 57)
[1] 10 12 45 33 57
> x
[1] 10 12 45 33
> x <- c(x, 57)
> x
[1] 10 12 45 33 57
```

- There are useful **FUNCTIONS TO GET INFORMATION ABOUT A VECTOR**

```
> x <- 0:10
> tail(x)
[1]  5  6  7  8  9 10
> head(x)
[1] 0 1 2 3 4 5
> head(x, n=2)
[1] 0 1
```

- You can **GIVE VECTOR ELEMENTS NAMES**
  - They're then referred to as *named vectors*

```
> x <- 1:4
> names(x)
NULL
> str(x)
 int [1:4] 1 2 3 4
> names(x) <- c("a", "b", "c", "d")
> x
a b c d
1 2 3 4
> str(x)
 Named int [1:4] 1 2 3 4
 - attr(*, "names")= chr [1:4] "a" "b" "c" "d"
```

SLIDE: Factors

- In general **DATA COMES AS ONE OF TWO TYPES**
  - *Quantitative data* represents measurable values. These are usually either **CONTINUOUS** (real values like a height in centimetres) or a **COUNT** (like number of beans in a tin).
  - *Categorical* data representing **DISCRETE GROUPS**, which can be **UNORDERED** (like "types of computer"; "educational establishments") or **ORDERED** (like floors of a building, or grades in school)

- **THIS DISTINCTION IS CRITICAL IN MANY STATISTICAL/ANALYTICAL METHODS**
- `R` **WAS MADE FOR STATISTICS** so has a special way of dealing with the difference
- **FACTORS ARE SPECIAL VECTORS REPRESENTING *CATEGORICAL* DATA**
  - Factors are stored as **VECTORS OF LABELLED INTEGERS**
  - Factors **CANNOT BE TREATED AS TEXT**
- **CREATE FACTOR IN SCRIPT**
  - We create a **FACTOR WITH THREE ELEMENTS**
  - `Run` the line
  - Commit the change

```
# Create a factor with three elements
> f <- factor(c("no", "yes", "no"))
```

- **INSPECT THE FACTOR IN THE CONSOLE**
  - When we look at the *structure* of the vector, it reports **TWO LEVELS**: `"yes"` and `"no"`
  - It also reports a list of values: `1 2 1`
  - There is a mapping `"no" -> 1` and `"yes" -> 2`
  - The **VECTOR STORES INTEGERS `1` and `2`, BUT THESE ARE LABELLED `"no"` and `"yes"`**

```
> length(f)
[1] 3
> str(f)
 Factor w/ 2 levels "no","yes": 1 2 1
> levels(f)
[1] "no"  "yes"
> f
[1] no  yes no
Levels: no yes
```

- **IN OUR `cats` DATA THE COAT WAS STORED AS A FACTOR**
- **DEMO IN CONSOLE**
  - The `class()` function **IDENTIFIES DATA STRUCTURES**
  - **NOTE THAT BY DEFAULT FACTORS ARE NUMBERED IN ALPHABETICAL ORDER OF LABEL**

```
> cats$coat
[1] calico black  tabby
Levels: black calico tabby
> class(cats$coat)
[1] "factor"
> str(cats$coat)
 Factor w/ 3 levels "black","calico",..: 2 1 3
```

SLIDE: Challenge 08

```
> f <- factor(c("case", "control", "case", "control", "case"))
> str(f)
 Factor w/ 2 levels "case","control": 1 2 1 2 1
> f <- factor(c("case", "control", "case", "control", "case"), levels=c("control", "c
> str(f)
 Factor w/ 2 levels "control","case": 2 1 2 1 2
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Matrices

- *Matrices* may be the **MOST IMPORTANT DATA STRUCTURE IN NUMERICAL ANALYSIS**, central to pretty much any statistical operation
- **THEY GET THEIR OWN SPECIAL DATA STRUCTURE IN `R`**
- They are 2D `vector` s (so contain atomic values)

- **CREATE MATRICES IN SCRIPT**

  - `Run` the lines when done
  - Commit the changes

```
# Create matrix of zeroes
m1 <- matrix(0, ncol = 6, nrow = 3)

# Create matrix of numbers 1 and 2
m2 <- matrix(c(1, 2), ncol = 3, nrow = 4)
```

- **INSPECT THE RESULTS IN THE CONSOLE**

  - `ncol` and `nrow` define the size of the matrix
  - providing a single value as the first argument fills the matrix with that value
  - The `length()` of a matrix **IS THE TOTAL NUMBER OF ELEMENTS**

```
> class(m1)
[1] "matrix"
> m1
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    0    0    0    0    0    0
[3,]    0    0    0    0    0    0
> str(m1)
 num [1:3, 1:6] 0 0 0 0 0 0 0 0 0 0 ...
 > length(m1)
[1] 18
```

- **PROVIDING TWO VALUES TO THE FIRST ARGUMENT** fills the matrix, too
    - So long as the **LENGTH OF THE MATRIX IS A MULTIPLE OF THE INPUT LENGTH**
    - We can **INDEX AND SLICE** just like before

```
> m2
     [,1] [,2] [,3] [,4]
[1,]    1    2    1    2
[2,]    2    1    2    1
[3,]    1    2    1    2
> m2[1, ]
[1] 1 2 1 2
> m2[2:3, 3:4]
     [,1] [,2]
[1,]    2    1
[2,]    1    2
```

SLIDE: Challenge 09 (5min)

```
> m <- matrix(1:50, nrow = 5, ncol = 10)
> m
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    6   11   16   21   26   31   36   41    46
[2,]    2    7   12   17   22   27   32   37   42    47
[3,]    3    8   13   18   23   28   33   38   43    48
[4,]    4    9   14   19   24   29   34   39   44    49
[5,]    5   10   15   20   25   30   35   40   45    50
> ?matrix
> m <- matrix(1:50, nrow = 5, ncol = 10, byrow = TRUE)
> m
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
[2,]   11   12   13   14   15   16   17   18   19    20
[3,]   21   22   23   24   25   26   27   28   29    30
[4,]   31   32   33   34   35   36   37   38   39    40
[5,]   41   42   43   44   45   46   47   48   49    50
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Lists

- `list` s are like *vectors*, **EXCEPT THEY CAN HOLD ANY DATA TYPE**
- **CREATE NEW LIST IN SCRIPT**
  - `Run` from script
  - Commit to repo

```
# Create a list
l <- list(1, 'a', TRUE, matrix(0, nrow = 2, ncol = 2), f)

# Create a named list
l_named <- list(a = "SWC", b = 1:4)
```

- **INSPECT THE LISTS IN THE CONSOLE**
  - The elements are identified with **DOUBLE SQUARE BRACKETS** `[[n]]`
  - We use this **LIKE ANY OTHER INDEX**

```
> class(l)
[1] "list"
> class(l_named)
[1] "list"
> str(l)
List of 5
 $ : num 1
 $ : chr "a"
 $ : logi TRUE
 $ : num [1:2, 1:2] 0 0 0 0
 $ : Factor w/ 2 levels "no","yes": 1 2 1
> str(l_named)
List of 2
 $ a: chr "SWC"
 $ b: int [1:4] 1 2 3 4
> l
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
     [,1] [,2]
[1,]    0    0
[2,]    0    0

[[5]]
[1] no  yes no
Levels: no yes
> l[[4]][1,1]
[1] 0
```

- **THE NAMED LIST IS SLIGHTLY DIFFERENT**
    - **CAN STILL INDEX**
    - But can also **NOW USE NAMES** with `$`
    - **INDICES CAN CHANGE IF DATA IS MODIFIED - NAMES ARE MORE ROBUST**
    - **NAMES CAN ALSO BE MORE DESCRIPTIVE (HELPS UNDERSTANDING)**

```
> l_named
$a
[1] "SWC"

$b
[1] 1 2 3 4
> l_named[[1]]
[1] "SWC"
> l_named[[2]]
[1] 1 2 3 4
> l_named$a
[1] "SWC"
> l_named$b
[1] 1 2 3 4
> names(l_named)
[1] "a" "b"
```

SLIDE: Logical Indexing

- We've seen **INDEXING** and **NAMES** as ways to get elements from variables **SO LONG AS WE KNOW WHICH ELEMENTS WE WANT**
- **LOGICAL INDEXING** allows us to **SPECIFY CONDITIONS FOR THE DATA WE WANT TO RECOVER**
  - For instance, we might want **ALL VALUES OVER A THRESHOLD** or **ALL NAMES STARTING WITH 'S'**
- **DEMO IN SCRIPT** (`data_structures.R`)
  - We create a vector of values as an example
  - We make a **MASK OF TRUE/FALSE (i.e. logical)** values
  - `Run` the lines

```
# Create a vector for logical indexing
v <- c(5.4, 6.2, 7.1, 4.8, 7.5)
mask <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
```

- **DEMO IN CONSOLE**
- Now, when we **USE THE MASK AS AN INDEX** we only get **THE ELEMENTS WHERE THE MASK IS TRUE**

```
> v
[1] 5.4 6.2 7.1 4.8 7.5
> v[mask]
[1] 5.4 7.1 7.5
```

- **COMPARATORS IN `R` RETURN VECTORS OF TRUE/FALSE VALUES**

- These can be **USED AS LOGICAL MASKS fOR DATA**
- Comparators **CAN BE COMBINED**

```
> v
[1] 5.4 6.2 7.1 4.8 7.5
> v < 7
[1]  TRUE  TRUE FALSE  TRUE FALSE
> v[v < 7]
[1] 5.4 6.2 4.8
> v < 7
[1]  TRUE  TRUE FALSE  TRUE FALSE
> v > 5 & v < 7
[1]  TRUE  TRUE FALSE FALSE FALSE
> v[v > 5 & v < 7]
[1] 5.4 6.2
> v > 5 | v < 7
[1] TRUE TRUE TRUE TRUE TRUE
> v[v > 5 | v < 7]
[1] 5.4 6.2 7.1 4.8 7.5
```

- **COMMIT CHANGES**

SLIDE: 05. Dataframes

- Dataframes are probably the most important thing you will ever learn about, in `R`.
- Almost everything in `R`, on a practical day-to-day basis, involves dataframes

---

SLIDE: Learning Objectives

- After this section **YOU WILL UNDERSTAND WHAT A DATAFRAME IS AND HOW IT IS BUILT UP FROM `R` DATA STRUCTURES YOU ALREADY KNOW**
- You'll also know **HOW TO ACCESS ANY PART OF A DATA FRAME, INCLUDING CONDITIONAL ACCESS**
- We'll also see how to read data in to, and write it out from, a data frame.

SLIDE: Let's look at a `data.frame`

- The `cats` data is a small `data.frame`
- **DEMO IN CONSOLE**
    - **NOTE: TABULAR**
    - **NOTE: 9 elements but length 3?**
    - Try list indexes… **IT'S A LIST**
    - Try `names()` … **IT'S A NAMED LIST**

- What are the classes of each list element? **THEY'RE VECTORS**

```
> class(cats)
[1] "data.frame"
> cats
    coat weight likes_string
1 calico    2.1            1
2  black    5.0            0
3  tabby    3.2            1
> length(cats)
[1] 3
> cats[[1]]
[1] calico black  tabby
Levels: black calico tabby
> typeof(cats)
[1] "list"
> names(cats)
[1] "coat"         "weight"       "likes_string"
> class(cats$coat)
[1] "factor"
> class(cats$weight)
[1] "numeric"
> class(cats$likes_string)
[1] "integer"
```

SLIDE: What is a `data.frame`

- This structure is **THE MOST IMPORTANT THING IN `R`**
  - It's the standard structure for storing any king of tabular, 'rectangular' data
- We've seen that it's a **NAMED LIST** where each element is a **VECTOR**
  - All the vectors have the same length
- This is **VERY SIMILAR TO A SPREADSHEET**, but it's more finicky:
  - We require every element in a column to be the same data type
  - We need all the columns to be the same length
  - **In spreadsheets, neither of these conditions are enforced**
  - **This makes `R` a bit more data-safe**

SLIDE: Creating a `data.frame`

- **DEMO IN SCRIPT**

- `Run` when done
- Commit to repo

```
# Create a data frame
df <- data.frame(a=c(1,2,3), b=c('eeny', 'meeny', 'miney'),
                 c=c(TRUE, FALSE, TRUE))
```

- **DEMO IN CONSOLE**
  - **!!!!STRINGS ARE INTERPRETED AS FACTORS!!!!**
  - The `summary()` function **SUMMARISES PROPERTIES OF EACH COLUMN**
  - The **summary depends on the column type**

```
> str(df)
'data.frame':	3 obs. of  3 variables:
 $ a: num  1 2 3
 $ b: Factor w/ 3 levels "eeny","meeny",..: 1 2 3
 $ c: logi  TRUE FALSE TRUE
> df$c
[1]  TRUE FALSE  TRUE
> length(df)
[1] 3
> dim(df)
[1] 3 3
> summary(df)
       a             b          c
 Min.   :1.0   eeny :1   Mode :logical
 1st Qu.:1.5   meeny:1   FALSE:1
 Median :2.0   miney:1   TRUE :2
 Mean   :2.0
 3rd Qu.:2.5
 Max.   :3.0
```

SLIDE: Challenge 10

```
author_book <- data.frame(author_first = c('Charles', 'Ernst', "Theodosius"),
                          author_last = c("Darwin", "Mayr", "Dobzhansky"),
                          year = c(1859, 1942, 1970))
```

Red sticky for a question or issue   Green sticky if complete

SLIDE: Challenge 11

```
> country_climate <- data.frame(country=c("Canada", "Panama",
+                               "South Africa", "Australia"),
+                               climate=c("cold", "hot",
+                                         "temperate", "hot/temperate"),
+                               temperature=c(10, 30, 18, "15"),
+                               northern_hemisphere=c(TRUE, TRUE,
+                                                     FALSE, "FALSE"),
+                               has_kangaroo=c(FALSE, FALSE,
+                                              FALSE, 1))
> str(country_climate)
'data.frame':	4 obs. of  5 variables:
 $ country            : Factor w/ 4 levels "Australia","Canada",..: 2 3 4 1
 $ climate            : Factor w/ 4 levels "cold","hot","hot/temperate",..: 1 2 4 3
 $ temperature        : Factor w/ 4 levels "10","15","18",..: 1 4 3 2
 $ northern_hemisphere: Factor w/ 2 levels "FALSE","TRUE": 2 2 1 1
 $ has_kangaroo       : num  0 0 0 1
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Challenge 12

```
> df <- data.frame(a=c(1,2,3), b=c('eeny', 'meeny', 'miney'),
+                  c=c(TRUE, FALSE, TRUE),
+                  stringsAsFactors = FALSE)
> str(df)
'data.frame':	3 obs. of  3 variables:
 $ a: num  1 2 3
 $ b: chr  "eeny" "meeny" "miney"
 $ c: logi  TRUE FALSE TRUE
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Adding rows and columns

- To add a row or column, we **BIND A VECTOR OR LIST** to the dataframe
- **DEMO IN THE CONSOLE**
    - **CHECK THE STRUCTURE FIRST**
    - If you try to **BIND A LIST WITH THE WRONG TYPES** you'll get an error **BUT THE ROW WILL BE BOUND ANYWAY!**

```
> df
  a     b     c
1 1  eeny  TRUE
2 2 meeny FALSE
3 3 miney  TRUE
> df <- cbind(df, vals = 3:1)
> df
  a     b     c vals
1 1  eeny  TRUE    3
2 2 meeny FALSE    2
3 3 miney  TRUE    1
> df <- rbind(df, list(4, 'mo', FALSE, 0))
Warning message:
In `[<-.factor`(`*tmp*`, ri, value = "mo") :
  invalid factor level, NA generated
> levels(df$b) <- c('eeny', 'meeny', 'miney', 'mo')
> df <- rbind(df, list(4, 'mo', FALSE, 0))
> > df
  a     b     c vals
1 1  eeny  TRUE    3
2 2 meeny FALSE    2
3 3 miney  TRUE    1
4 4  <NA> FALSE    0
5 4    mo FALSE    0
```

- **NOW WE HAVE A GARBAGE ROW!!**
- We can **REMOVE THE ROW IN SEVERAL WAYS**
    - Use the `-` syntax
    - Remove all rows with `NA` values
- We reassign `df` with one of these

```
> df[-4,]
  a     b     c vals
1 1  eeny  TRUE    3
2 2 meeny FALSE    2
3 3 miney  TRUE    1
5 4    mo FALSE    0
> na.omit(df)
  a     b     c vals
1 1  eeny  TRUE    3
2 2 meeny FALSE    2
3 3 miney  TRUE    1
5 4    mo FALSE    0
> df <- na.omit(df)
> df
  a     b     c vals
1 1  eeny  TRUE    3
2 2 meeny FALSE    2
3 3 miney  TRUE    1
5 4    mo FALSE    0
```

SLIDE: Writing `data.frame` to file

- **DEMO IN CONSOLE**
  - The `write.table()` function **WRITES A DATAFRAME TO A FILE**
  - We pass: the dataframe, the filename, the column separator, and whether the header should be written
  - **`\t` means 'tab' - it puts a gap between columns**

```
write.table(df, "data/df_example.tab", sep="\t")
```

- **INSPECT THE FILE**
  - Navigate there in the `Files` tab
  - View the file in `RStudio`
  - **NOTE:** row and column names are written automatically
  - Using `\t` has given spaces as column separators

SLIDE: Reading into a `data.frame`

- **DEMO IN SCRIPT**
- **DOWNLOAD DATA**
  - Use the link from the Etherpad document
  - Place the file in `data/`

- **CREATE A NEW SCRIPT**
    - Call it `gapminder`
    - Add the code
    - We need to provide a data source (**here, a file**), the separator character, and whether there's a header row

```
# Load gapminder data from a URL
gapminder <- read.table("data/gapminder-FiveYearData.csv", sep=",", header=TRUE)
```

- **ADD AND COMMIT TO REPO**
- **CHECK THE DATA IN THE `Environment` TAB**
    - Click on `gapminder` in `Evironment` tab.
    - **NOTE COLUMNS**
    - **DEMO IN CONSOLE**

SLIDE: Investigating `gapminder`

- Now we've loaded our data, let's take a look at it
- **DEMO IN CONSOLE**
    - 1704 rows, 6 columns
    - Investigate types of columns
    - **POINT OUT THAT THE TYPE OF A COLUMN IS INTEGER IF IT'S A FACTOR**
    - **LENGTH OF A DATAFRAME IS THE NUMBER OF COLUMNS**

```
> str(gapminder)
'data.frame':	1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
> typeof(gapminder$year)
[1] "integer"
> typeof(gapminder$country)
[1] "integer"
> str(gapminder$country)
 Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
> length(gapminder)
[1] 6
> nrow(gapminder)
[1] 1704
> ncol(gapminder)
[1] 6
```

```
> dim(gapminder)
[1] 1704    6
> colnames(gapminder)
[1] "country"   "year"      "pop"       "continent" "lifeExp"   "gdpPercap"
> head(gapminder)
      country year      pop continent lifeExp gdpPercap
1 Afghanistan 1952  8425333      Asia  28.801  779.4453
2 Afghanistan 1957  9240934      Asia  30.332  820.8530
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
4 Afghanistan 1967 11537966      Asia  34.020  836.1971
5 Afghanistan 1972 13079460      Asia  36.088  739.9811
6 Afghanistan 1977 14880372      Asia  38.438  786.1134
> summary(gapminder)
        country          year           pop               continent       lifeExp
 Afghanistan:  12   Min.   :1952   Min.   :6.001e+04   Africa  :624   Min.   :23.60
 Albania    :  12   1st Qu.:1966   1st Qu.:2.794e+06   Americas:300   1st Qu.:48.20
 Algeria    :  12   Median :1980   Median :7.024e+06   Asia    :396   Median :60.71
 Angola     :  12   Mean   :1980   Mean   :2.960e+07   Europe  :360   Mean   :59.47
 Argentina  :  12   3rd Qu.:1993   3rd Qu.:1.959e+07   Oceania : 24   3rd Qu.:70.85
 Australia  :  12   Max.   :2007   Max.   :1.319e+09                  Max.   :82.60
 (Other)    :1632
   gdpPercap
 Min.   :   241.2
 1st Qu.:  1202.1
 Median :  3531.8
 Mean   :  7215.3
 3rd Qu.:  9325.5
 Max.   :113523.1
```

SLIDE: Subsets of `data.frame` s

- **DATAFRAMES ARE LISTS** so subset like lists
- **DATAFRAMES ARE ALSO 2D DATA** so subset like matrices
- **DEMO IN CONSOLE**

```
# Extract a single column, get a dataframe
> head(gapminder[3])
       pop
1  8425333
2  9240934
3 10267083
4 11537966
5 13079460
6 14880372
> class(head(gapminder[3]))
[1] "data.frame"
```

```
# Extract a single named column, get a vector/factor
> head(gapminder[["lifeExp"]])
[1] 28.801 30.332 31.997 34.020 36.088 38.438
> class(head(gapminder[["lifeExp"]]))
[1] "numeric"
> head(gapminder$year)
[1] 1952 1957 1962 1967 1972 1977
> class(head(gapminder$year))
[1] "integer"

# Slice rows like a matrix, get a dataframe
> gapminder[1:3,]
      country year      pop continent lifeExp gdpPercap
1 Afghanistan 1952  8425333      Asia  28.801  779.4453
2 Afghanistan 1957  9240934      Asia  30.332  820.8530
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
> class(gapminder[1:3,])
[1] "data.frame"
> gapminder[3,]
      country year      pop continent lifeExp gdpPercap
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
> class(gapminder[3, ])
[1] "data.frame"

# Slice columns like a matrix, get vector/factor
> head(gapminder[, 3])
[1]  8425333  9240934 10267083 11537966 13079460 14880372
> class(head(gapminder[, 3]))
[1] "numeric"

# Slice columns like a matrix get dataframe
> head(gapminder[, 3, drop=FALSE])
       pop
1  8425333
2  9240934
3 10267083
4 11537966
5 13079460
6 14880372
> class(head(gapminder[, 3, drop=FALSE]))
[1] "data.frame"
```

SLIDE: Challenge 13

```
# Extract observations collected for the year 1957
> head(gapminder[gapminder$year == 1957,])
        country year      pop continent lifeExp gdpPercap
2   Afghanistan 1957  9240934      Asia  30.332   820.853
```

```
2  Afghanistan 1957  9240934      Asia  30.332   820.853
14     Albania 1957  1476505    Europe  59.280  1942.284
26     Algeria 1957 10270856    Africa  45.685  3013.976
38      Angola 1957  4561361    Africa  31.999  3827.940
50   Argentina 1957 19610538  Americas  64.399  6856.856
62   Australia 1957  9712569   Oceania  70.330 10949.650

# Extract all columns except 1 through 4
> head(gapminder[, -(1:4)])
  lifeExp gdpPercap
1  28.801  779.4453
2  30.332  820.8530
3  31.997  853.1007
4  34.020  836.1971
5  36.088  739.9811
6  38.438  786.1134
> head(gapminder[, -1:-4])
  lifeExp gdpPercap
1  28.801  779.4453
2  30.332  820.8530
3  31.997  853.1007
4  34.020  836.1971
5  36.088  739.9811
6  38.438  786.1134

# Extract all rows where life expectancy is greater than 80 years
> head(gapminder[gapminder$lifeExp > 80,])
              country year      pop continent lifeExp gdpPercap
71          Australia 2002 19546792   Oceania  80.370  30687.75
72          Australia 2007 20434176   Oceania  81.235  34435.37
252            Canada 2007 33390141  Americas  80.653  36319.24
540            France 2007 61083916    Europe  80.657  30470.02
671 Hong Kong China 2002  6762476      Asia  81.495  30209.02
672 Hong Kong China 2007  6980412      Asia  82.208  39724.98

# ADVANCED: Extract rows for years 2002 and 2007
> head(gapminder[gapminder$year == 2002 | gapminder$year == 2007,])
       country year      pop continent lifeExp gdpPercap
11 Afghanistan 2002 25268405      Asia  42.129  726.7341
12 Afghanistan 2007 31889923      Asia  43.828  974.5803
23     Albania 2002  3508512    Europe  75.651 4604.2117
24     Albania 2007  3600523    Europe  76.423 5937.0295
35     Algeria 2002 31287142    Africa  70.994 5288.0404
36     Algeria 2007 33333216    Africa  72.301 6223.3675
> head(gapminder[gapminder$year %in% c(2002, 2007),])
       country year      pop continent lifeExp gdpPercap
11 Afghanistan 2002 25268405      Asia  42.129  726.7341
12 Afghanistan 2007 31889923      Asia  43.828  974.5803
23     Albania 2002  3508512    Europe  75.651 4604.2117
```

```
24      Albania 2007  3600523    Europe  76.423 5937.0295
35      Algeria 2002 31287142    Africa  70.994 5288.0404
36      Algeria 2007 33333216    Africa  72.301 6223.3675

# The %in% operator
> 1 %in% c(1, 2, 3, 4, 5)
[1] TRUE
> 6 %in% c(1, 2, 3, 4, 5)
[1] FALSE
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: 06. Packages

SLIDE: Learning Objectives

- In this short section, we'll learn
  - **what packages are**
  - **how to install them**
  - **how to use them in your code**

SLIDE: Packages

- Packages are **THE FUNDAMENTAL UNIT OF REUSABLE CODE IN `R`**
- People write code, and **DISTRIBUTE IT IN PACKAGES**
- Packages exist for many **SPECIALIST AND USEFUL TOOLS**
- Over 10,000 packages can be found at CRAN - the Comprehensive R Archive Network
- When you write your own code, you can distribute it as a package

- **DEMO IN CONSOLE**

  - You can **SEE INSTALLED PACKAGES** with the function `installed.packages()`
  - To install a new package, use `install.packages("packagename")` as a string **EXPLAIN DEPENDENCIES**
  - **DEMO INSTALLATION IN `RStudio` : `Tools` $\rightarrow$ `Install packages...`**
  - **DEMO PACKAGE UPDATES IN `RStudio`**
  - You can update your installed packages to the newest version in the console with `update.packages()` **DON'T DO THIS - CAN TAKE TIME!**

```
> installed.packages()
                     Package
BiocInstaller        "BiocInstaller"
bit                  "bit"
bit64                "bit64"
data.table           "data.table"
[...]
> install.packages("dplyr")
Installing package into ‘/Users/lpritc/Library/R/3.4/library’
(as ‘lib’ is unspecified)
also installing the dependencies ‘bindrcpp’, ‘glue’, ‘rlang’
[...]
> update.packages(ask=FALSE)
> library(dplyr)
```

SLIDE: Challenge 14

Red sticky for a question or issue   Green sticky if complete

07. Creating Publication-Quality Graphics

SLIDE: Visualisation is Critical

- Visualisation **HELPS US UNDERSTAND OUR DATA**
- But **IT'S NOT FOOLPROOF** - people can interpret the same visualisation differently
- Good visualisation is **MORE THAN JUST USING A PLOTTING TOOL**

SLIDE: Learning Objectives

- After this section, you should **understand the Grammar of Graphics**
    - You'll be able to produce **INFORMATIVE, BEAUTIFUL GRAPHS THAT EXPLAIN YOUR DATA**
- You'll also be able to use `ggplot2` to generate those plots

SLIDE: The Grammar of Graphics

- We'll be using the `ggplot2` package, which is part of the **TIDYVERSE**, created initially by Hadley Wickham.

    - The Tidyverse provides **OTHER USEFUL PACKAGES** but you can use `ggplot2` on its own

- `ggplot2` implements **A SET OF CONCEPTS CALLED THE GRAMMAR OF GRAPHICS**

    - This **SEPARATES DATA FROM THE WAY IT'S REPRESENTED** and we'll discuss it in detail
    - It's not the usual way you might have seen to create plots, but it's **highly effective for generating powerful visualisations**

SLIDE: A Basic Scatterplot

- You can use `ggplot2` in the **SAME WAY YOU'D USE BASE GRAPHICS**

    - This is not the best way to use all the power of the package

- **DEMO IN CONSOLE**

    - **IMPORT LIBRARY**
    - `ggplot2` has **`qplot()` - the equivalent to `plot()` in base graphics**
    - `plot()` takes `x` and `y` values, and will assign colours to `factor` columns
    - `qplot()` takes the name of `x` and `y` columns, plus the name of the source `data.frame`, and will assign colours to `factor` columns

```
> library(ggplot2)
> plot(gapminder$lifeExp, gapminder$gdpPercap, col=gapminder$continent)
> qplot(lifeExp, gdpPercap, data=gapminder, colour=continent)
```

- **COMPARE THE GRAPHS**

    - Clearly, both graphs **show the same data**
    - The **FORMATTING IS QUITE DIFFERENT**
    - Your preference is your preference - **both methods can be heavily restyled**
    - My view is that **`ggplot2` has nicer default styles**
    - **`ggplot2` provides gridlines and legends by default, and the labelling is clearer** (no `gapminder$` prefix)

- **THIS ISN'T WHAT'S POWERFUL ABOUT `ggplot2`!**

**SLIDE: What is a Plot? *aesthetics* **

- **TALK THROUGH THE POINTS**
- Each observation in the data is a *point*
- The *aesthetics* of a point determine how it is rendered in the plot

    - co-ordinates (x, y values) **ON THE IMAGE**
    - size

- shape
- colour
- transparency

- *aesthetics* can be

  - *constant* (e.g. all points the same colour)
  - *mapped to variables* (e.g. colour mapped to continent)

**SLIDE: What is a Plot? *aesthetics* **

- The *aesthetics* of a plot **define a new dataset** for each point
- **THIS SHOULD REMIND YOU STRONGLY OF A `data.frame`**

SLIDE: What is a Plot? `geom` s

- So far **we've only defined the data and aesthetics**

  - **THIS ONLY TELLS US HOW DATA POINTS ARE REPRESENTED, NOT THE TYPE OF PLOT**

- `geom` s (short for *geometries*) **DEFINE THE KIND OF PLOT WE PRODUCE**

  - Showing the data **as points** is a *scatterplot*
  - Showing the data **as lines** is a *line plot*
  - Showing the data **as bars** is a *barchart*

- We can use **different `geom` s with the *same data and aesthetics* **

SLIDE: What is a Plot? `geom` s

- **DEMO IN SCRIPT** (`gapminder.R`)

  - We **create a plot with the `ggplot()` function**.
  - We define the *data* as `data` , and *aesthetics* with `aes`
  - **WE PUT THE RESULT IN A VARIABLE FOR CONVENIENCE**
  - *Data* and *aesthetics* aren't enough to define a plot. **WE NEED A `geom`**
  - Use `geom_point()`

```
# Generate plot of GDP per capita against life Expectancy
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap, color=continent))
p + geom_point()
```

- **WE'VE RECREATED THE SCATTERPLOT WE SAW EARLIER**
- **COMMIT CHANGES TO SCRIPT**

- **What happens if we change the `geom` ?**
- **DEMO IN THE SCRIPT**

```
p + geom_line()
```

- This looks terrible. **CHANGE IT BACK**

SLIDE: Challenge 15

```
# Plot life expectancy against time
p <- ggplot(data=gapminder, aes(x=year, y=lifeExp, colour=continent))
p + geom_point()
```

Red sticky for a question or issue

Green sticky if complete

**SLIDE: What is a Plot? *layers* **

- Without knowing it, **WE'VE JUST BEEN USING THE LAYERS CONCEPT**
  - **all `ggplot2` plots are built as layers**
- **ALL LAYERS HAVE TWO COMPONENTS**
  - *data* to be shown, and *aesthetics* for showing them
  - a `geom` defining the type of plot
- **The `ggplot` object describes a *base* layer, and can contain *data* and *aesthetics* **
  - **THESE ARE INHERITED BY THE OTHER LAYERS IN THE PLOT**
  - **The values can also be overridden in specified layers**

**SLIDE: What is a Plot? *layers* **

- In our first plot we defined a *base* with:
  - *data* from `gapminder`
  - *aesthetics* with *x* and *y* coordinates, and colouring by continent
- We defined a layer that:
  - had a `geom_point` `geom`
  - inherited *data* and *aesthetics* from the *base*

LAYERS ARE ADDED WITH THE `+` OPERATOR

**SLIDE: What is a Plot? *layers* **

- Now we will **override the base layer *aesthetics* **
- **DEMO IN SCRIPT**
  - We'll **change the `geom`** to `geom_line`
  - We'll extend the *aesthetics* to **group datapoints by country**

```
# Generate plot of GDP per capita against life expectancy
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap, color=continent))
p + geom_line(aes(group=country))
```

- **RENDER THE PLOT**

**SLIDE: What is a Plot? *layers* **

- We can **BUILD UP LAYERS OF `geom` S** to produce a more complex plot
- We **ADD A NEW `geom_point()` LAYER WITH `+`**
  - We use the layer's `alpha` argument to control transparency
- **DEMO IN SCRIPT**

```
# Generate plot of GDP per capita against life expectancy
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap, color=continent))
p + geom_line(aes(group=country)) + geom_point(alpha=0.4)
```

- **RENDER THE PLOT**
- **COMMIT THE CHANGES**

SLIDE: Challenge 16

```
# Generate plot of life expectancy against time
p <- ggplot(data=gapminder, aes(x=year, y=lifeExp, color=continent))
p + geom_line(aes(group=country)) + geom_point(alpha=0.35)
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Transformations and `scale` s

- Another kind of layer is a *transformation* - handled with `scale` layers
- These map *data* to new aesthetics on the plot
    - new axis scales, e.g. log scale, reverse scale, time scale
    - colour scaling (changing palettes)
    - shape and size scaling
- **DEMO IN SCRIPT** (`gapminder.R`)
    - Rescale the plot first
    - Then change the colours

```
# Generate plot of GDP per capita against life expectancy
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap, color=continent))
p <- p + geom_line(aes(group=country)) + geom_point(alpha=0.4)
p + scale_y_log10() + scale_color_grey()
```

SLIDE: Statistics layers

- Some `geom` layers transform the dataset
    - Usually this is a data summary (e.g. smoothing or binning)
    - The layer **may provide a new summary visual object**
- **DEMO IN SCRIPT**
    - This is working towards an informative figure
    - **Start with a new basic scatterplot**
    - **NOTE:** setting opacity helps see density in the data - **looks like two main points of density**
    - **NOTE:** looks like a general trend of GDP and life expectancy correlating

```
# Generate summary plot of GDP per capita against life expectancy
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap))
p + geom_point(alpha=0.4) + scale_y_log10()
```

- **ADD A SMOOTHED FIT**
    - **NOTE:** The correlation is made quite clear

```
# Generate summary plot of GDP per capita against life expectancy
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap))
p <- p + geom_point(alpha=0.4) + scale_y_log10()
p + geom_smooth()
```

- **ADD A CONTOUR PLOT OF DENSITY**
    - **NOTE:** Two populations are clear

- **We might speculate that there is a difference in wealth/life expectancy across continents**

```
# Generate summary plot of GDP per capita against life expectancy
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap))
p <- p + geom_point(alpha=0.4) + scale_y_log10()
p + geom_density_2d(color="purple")
```

- **ADD CONTINENT COLOURING**
    - **NOTE:** It's now clear that the two populations are centred on Europe (wealthy, long-lived) and Africa (poor, short-lived), respectively.

```
p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap))
p <- p + geom_point(alpha=0.4, aes(color=continent)) + scale_y_log10()
p + geom_density_2d(color="purple")
```

- **COMMIT CHANGES**

SLIDE: Multi-panel figures

- All our plots so far have been single figures, but **multi-panel plots can give clearer comparisons**
- The `facet_wrap()` **layer allows us to make grids of plots, SPLIT BY A FACTOR**
- **DEMO IN THE SCRIPT**
    - We set a **default aesthetic grouping by country**
    - We generate a line plot, with log y axis
    - **The result is a bit messy.**

```
# Compare life expectancy over time by country
p <- ggplot(data=gapminder, aes(x=year, y=lifeExp, colour=continent, group=country))
p + geom_line() + scale_y_log10()
```

- **using** `facet_wrap()` **to split by continent is clearer**
    - **NOTE:** the axes are consistent across facets

```
p <- ggplot(data=gapminder, aes(x=year, y=lifeExp, colour=continent, group=country))
p <- p + geom_line() + scale_y_log10()
p + facet_wrap(~continent)
```

SLIDE: Challenge 17 (10min)

```r
# Contrast GDP per capita against population
p <- ggplot(data=gapminder, aes(x=pop, y=gdpPercap))
p <- p + geom_point(alpha=0.8, aes(color=continent))
p <- p + scale_y_log10() + scale_x_log10()
p + geom_density_2d(alpha=0.5) + facet_wrap(~year)
```

Red sticky for a question or issue

Green sticky if complete

08. Working with `data.frame`s in `dplyr`

SLIDE: Learning Objectives

- You're going to **learn to manipulate `data.frame`s with the six *verbs* of `dplyr`**
- `select()`
- `filter()`
- `group_by()`
- `summarize()`
- `mutate()`
- `%>%` (pipe)

SLIDE: What and Why is `dplyr`?

- `dplyr` is a package in the **TIDYVERSE**; it exists to enable **rapid analysis of data by groups**
  - For example, if we wanted numerical (rather than graphical) analysis of the `gapminder` data by continent, we'd use `dplyr`
- So far, we know how to **subset, but repetitive application is tedious**
  - **WE MIGHT MANAGE TO REPEAT BY CONTINENT, LIKE HERE - BUT *BY COUNTRY*?**
- **AVOIDING REPETITION IMPROVES YOUR CODE**
  - More **robust**
  - More **readable**

- More **reproducible**

SLIDE: Split-Apply-Combine

- The **general principle** `dplyr` supports is **SPLIT-APPLY-COMBINE**
- We have a **dataset with several groups** (column `x`)
- We **want to perform the same operation on each group, independently** - take a mean of `y` for each group, for example
  - So we **SPLIT** the data into groups, on `x`
  - Then we **APPLY** the operation (take the mean for each group)
  - Then we **COMBINE** the results into a new table

SLIDE: `select()` - Interactive Demo

- **DEMO IN CONSOLE**
  - Import `dplyr`

```
> library(dplyr)
```

- The `select()` *verb* **SELECTS COLUMNS**
  - **DEMO IN CONSOLE**
  - If we wanted to select only year, country and GDP data from `gapminder`
  - Specify: **data, then columns**

```
> head(select(gapminder, year, country, gdpPercap))
  year     country gdpPercap
1 1952 Afghanistan  779.4453
2 1957 Afghanistan  820.8530
3 1962 Afghanistan  853.1007
4 1967 Afghanistan  836.1971
5 1972 Afghanistan  739.9811
6 1977 Afghanistan  786.1134
```

- Here, we **applied a function**, but we can also **'PIPE' DATA FROM ONE VERB TO ANOTHER**
  - These work **like pipes in the shell**
  - **SPECIAL PIPE SYMBOL:** `%>%`
  - Specify **only columns**

```
> head(gapminder %>% select(year, country, gdpPercap))
  year     country gdpPercap
1 1952 Afghanistan  779.4453
2 1957 Afghanistan  820.8530
3 1962 Afghanistan  853.1007
4 1967 Afghanistan  836.1971
5 1972 Afghanistan  739.9811
6 1977 Afghanistan  786.1134
```

SLIDE: `filter()`

- `filter()` selects rows on the basis of some condition, or combination of conditions
    - We can **use it as a function, with *pipes* **
- **DEMO IN CONSOLE**

```
> head(filter(gapminder, continent=="Europe"))
  country year     pop continent lifeExp gdpPercap
1 Albania 1952 1282697    Europe   55.23  1601.056
2 Albania 1957 1476505    Europe   59.28  1942.284
3 Albania 1962 1728137    Europe   64.82  2312.889
4 Albania 1967 1984060    Europe   66.22  2760.197
5 Albania 1972 2263554    Europe   67.69  3313.422
6 Albania 1977 2509048    Europe   68.93  3533.004
```

- **DEMO IN SCRIPT** (`gapminder.R`)
    - One **advantage of pipes** is that they make chaining *verbs* together **MORE READABLE**
    - **END THE LINES WITH THE PIPE SYMBOL** so `R` knows that there's a continuation
    - `Run` the lines and **check the output** in `Environment`
    - `Commit` the changes

```
# Select gdpPercap by country and year, only for Europe
eurodata <- gapminder %>%
              filter(continent == "Europe") %>%
              select(year, country, gdpPercap)
```

**SLIDE: Challenge 18

```
# Select life expectancy by country and year, only for Africa
afrodata <- gapminder %>%
  filter(continent == "Africa") %>%
  select(year, country, lifeExp)
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: `group_by()`

- The `group_by()` *verb* **SPLITS `data.frame` s INTO GROUPS ON A COLUMN PROPERTY**
- **DEMO IN CONSOLE**
  - It returns a `tibble` - a table with extra metadata describing the groups in the table

```
> group_by(gapminder, continent)
# A tibble: 1,704 x 6
# Groups:   continent [5]
       country  year      pop continent lifeExp gdpPercap
        <fctr> <int>    <dbl>    <fctr>   <dbl>     <dbl>
 1 Afghanistan  1952  8425333      Asia  28.801  779.4453
 2 Afghanistan  1957  9240934      Asia  30.332  820.8530
 3 Afghanistan  1962 10267083      Asia  31.997  853.1007
 4 Afghanistan  1967 11537966      Asia  34.020  836.1971
 5 Afghanistan  1972 13079460      Asia  36.088  739.9811
 6 Afghanistan  1977 14880372      Asia  38.438  786.1134
 7 Afghanistan  1982 12881816      Asia  39.854  978.0114
 8 Afghanistan  1987 13867957      Asia  40.822  852.3959
 9 Afghanistan  1992 16317921      Asia  41.674  649.3414
10 Afghanistan  1997 22227415      Asia  41.763  635.3414
# ... with 1,694 more rows
```

**SLIDE: `summarize()`

- The **combination of `group_by()` and `summarize()` is very powerful**
  - We can **CREATE NEW VARIABLES** using functions that repeat for each group
- Here, we've split the original table into three groups, and now **CREATE A NEW VARIABLE `mean_b` THAT IS FILLED BY CALCULATING THE MEAN OF `b`**
- **DEMO IN SCRIPT**
  - We use the same principle to **calculate mean GDP per continent**

```
> # Produce table of mean GDP by continent
> gapminder %>%
+     group_by(continent) %>%
+     summarize(meangdpPercap=mean(gdpPercap))
# A tibble: 5 x 2
  continent meangdpPercap
     <fctr>         <dbl>
1    Africa      2193.755
2  Americas      7136.110
3      Asia      7902.150
4    Europe     14469.476
5   Oceania     18621.609
```

SLIDE: challenge 19

- **IN THE SCRIPT**

```
# Find average life expectancy by nation
avg_lifexp_country <- gapminder %>%
  group_by(country) %>%
  summarize(meanlifeExp=mean(lifeExp))
```

- **IN THE CONSOLE**

```
> avg_lifexp_country[avg_lifexp_country$meanlifeExp == max(avg_lifexp_country$meanlif
# A tibble: 1 x 2
  country meanlifeExp
   <fctr>       <dbl>
1 Iceland    76.51142
> avg_lifexp_country[avg_lifexp_country$meanlifeExp == min(avg_lifexp_country$meanlif
# A tibble: 1 x 2
       country meanlifeExp
        <fctr>       <dbl>
1 Sierra Leone    36.76917
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: `count()` and `n()`

- Two other useful functions are related to `summarize()`
  - `count()` **reports a new table of counts by group**
  - `n()` **is used to represent the count of rows, when calculating new values in**

`summarize()`

DEMO IN CONSOLE * **NOTE:** standard error is (std dev)/sqrt(n)

```
> gapminder %>% filter(year == 2002) %>% count(continent, sort = TRUE)
# A tibble: 5 x 2
  continent     n
     <fctr> <int>
1    Africa    52
2      Asia    33
3    Europe    30
4  Americas    25
5   Oceania     2
> gapminder %>% group_by(continent) %>% summarize(se_lifeExp = sd(lifeExp)/sqrt(n()))
# A tibble: 5 x 2
  continent se_lifeExp
     <fctr>      <dbl>
1    Africa  0.3663016
2  Americas  0.5395389
3      Asia  0.5962151
4    Europe  0.2863536
5   Oceania  0.7747759
```

SLIDE: `mutate()`

- `mutate()` **CALCULATES NEW VARIABLES (COLUMNS) ON THE BASIS OF EXISTING COLUMNS**
- **DEMO IN SCRIPT**
    - Say we want to calculate the **total GDP of each nation, each year, in $bn**
    - We'd multiply the GDP per capita by the total population, and divide by 1bn
- **INSPECT THE OUTPUT**
    - We have a new data table, which is the `gapminder` data, plus an extra column

```
# Calculate GDP in $billion
gdp_bill <- gapminder %>%
  mutate(gdp_billion = gdpPercap * pop / 10^9)
```

- **WE CAN CHAIN ALL THESE OPERATIONS TOGETHER WITH PIPES**
- **We can calculate several summaries in a single** `summarize()` **command**
- We can use the output of `mutate()` in the `summarize()` command
- **DEMO IN SCRIPT**
    - We're going to calculate the **total (and standard deviation) of GDP per continent, per year**

- Calculate total GDP first
- Group by continent and year
- Summarise mean and sd of GDP per capita, and total GDP
- **INSPECT THE OUTPUT**
- `Commit` the changes

```
# Calculate total/sd of GDP by continent and year
gdp_bycontinents_byyear <- gapminder %>%
  mutate(gdp_billion=gdpPercap*pop/10^9) %>%
  group_by(continent,year) %>%
  summarize(mean_gdpPercap=mean(gdpPercap),
            sd_gdpPercap=sd(gdpPercap),
            mean_gdp_billion=mean(gdp_billion),
            sd_gdp_billion=sd(gdp_billion))
```

---

09. Program Flow Control

SLIDE: Learning Objectives

- In this short section, you'll learn how to **perform actions depending on values of data** in `R`
- You'll also learn how to **repeat operations, using `for()` loops**
- **These are very important general concepts, that recur in many programming languages**
- Much of the time, you can avoid using them in `R` data analyses, because `dplyr` exists, and because `R` is **vectorised**

SLIDE: `if()` ... `else`

- We **often want to run a piece of code, or take an action, dependent on whether some data has a particular value (is true or false, say**
- When this is the case, we can use the general `if()` ... `else` structure, which is common to most programming languages

- **DEMO IN SCRIPT**

- **CREATE NEW SCRIPT** (`flow_control.R`)

  - Let's say that we want to print a message if some value is greater than 10
  - **NOTE AUTOCOMPLETION/BRACKETS ETC.**
  - **THE CODE TO BE RUN GOES IN CURLY BRACES**
  - `Source` the file
  - **NOTHING HAPPENS** (`x > 10` is `FALSE`)
  - The `if()` block executes **if the value in the parentheses evaluates to `TRUE`**

```
# A data point
x <- 8

# Example if statement
if (x > 10) {
  print("x is greater than 10")
}
```

- **MODIFY THE SCRIPT**
  - Add the `else` block
  - `Source` the code: **we get a message**
  - **BUT IS THE MESSAGE TRUE?**

```
# Example if statement
if (x > 10) {
  print("x is greater than 10")
} else {
  print("x is less than 10")
}
```

- **SET `x <- 10` AND TRY AGAIN**
- **MODIFY THE SCRIPT WITH `else if()` STATEMENT**
  - `Source` the script: **NO OUTPUT**

```
# A data point
x <- 10

# Example if statement
if (x > 10) {
  print("x is greater than 10")
} else if (x < 10) {
  print("x is less than 10")
}
```

- **MODIFY THE SCRIPT WITH A FINAL `else` STATEMENT**
  - `Source` the script: **EQUALS** output

```
# A data point
x <- 9

# Example if statement
if (x > 10) {
  print("x is greater than 10")
} else if (x < 10) {
  print("x is less than 10")
} else {
  print("x is equal to 10")
}
```

SLIDE: Challenge 20

```
# Are there any records for a year
year <- 2002
if(any(gapminder$year == year)){
   print("Record(s) for this year found.")
}
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: `for()` loops

- If you want to iterate over a set of values, then `for()` loops can be used
- `for()` loops are **a very common programming construct**
- They express the idea: **FOR EACH ITEM IN A GROUP, DO SOMETHING (WITH THAT ITEM)**

- **DEMO IN SCRIPT** (`flow_control.R`)

  - Say we have a *vector* `c(1,2,3)` , and we want to print each item
  - We can **loop over all the items** and print them

- **The loop structure is**

  - `for()` , where the argument names a variable (`i`) - the *iterator*, and a set of values: **`for(i in c('a', 'b', 'c'))`**
  - A **CODE BLOCK** defined by curly braces (**note automated completion)
  - The **contents of the code block are executed for each value of the iterator**

```
# Basic for loop
for(i in c('a', 'b', 'c')){
  print(i)
}
```

- **Loops can (but shouldn't always) be nested**
- **DEMO IN SCRIPT**
  - The outer loop is executed and, **for each value in the outer loop, the inner loop is executed to completion**

```
# Nested loop example
for (i in 1:5) {
  for (j in c('a', 'b', 'c')) {
    print(paste(i, j))
  }
}
```

- The simplest way to capture output is to add a new item to a vector each iteration of the loop
- **DEMO IN SCRIPT**
  - **REMIND:** using `c()` to append to a vector

```
# Capture loop output
output <- c()
for (i in 1:5) {
  for (j in c('a', 'b', 'c', 'd', 'e')) {
    output <- c(output, paste(i, j))
  }
}
(output)
```

- **GROWING OUTPUT FROM LOOPS IS COMPUTATIONALLY VERY EXPENSIVE**
  - Better to define the empty output container first (**if you know the dimensions**)
- **MODIFY IN SCRIPT**

```
# Capture loop output
output_matrix <- matrix(nrow=5, ncol=5)
j_letters <- c('a', 'b', 'c', 'd', 'e')
for (i in 1:5) {
  for (j in 1:5) {
    output_matrix[i, j] <-paste(i, j_letters[j])
  }
}
(output_matrix)
```

SLIDE: `while()` loops

- Sometimes you need to perform some action **WHILE A CONDITION IS TRUE**
  - This isn't as common as a `for()` loop
  - It's a **general programming construct**
- **DEMO IN SCRIPT**
  - We'll **generate random numbers until one falls below a threshold**
  - `runif()` generates random numbers from a uniform distribution
  - We print random numbers until one is less than 0.1
- **run a couple of times to show the output is random**

```
# Example while loop
z <- 1
while(z > 0.1){
  z <- runif(1)
  print(z)
}
```

- **COMMIT THE SCRIPT**

SLIDE: Challenge 21

```
# Challenge solution
for (l in letters) {
  if (l %in% c('a', 'e', 'i', 'o', 'u')) {
    value <- TRUE
  } else {
    value <- FALSE
  }
  print(paste(l, value))
}
```

Red sticky for a question or issue

Green sticky if complete

SLIDE: Vectorisation

- Although `for()` and `while()` loops can be useful, they are **rarely the most efficient way to work in `R`**
- **MOST FUNCTIONS IN `R` ARE VECTORISED**

- **When applied to a vector, they work on all elements in the vector**
- So no need to use a loop.

- **DEMO IN CONSOLE**
    - **Operators** are vectorised

```
> x <- 1:4
> x
[1] 1 2 3 4
> x * 2
[1] 2 4 6 8
```

- **You can operate on vectors together**

```
> y <- 6:9
> y
[1] 6 7 8 9
> x + y
[1]  7  9 11 13
> x * y
[1]  6 14 24 36
```

- **Comparison operators are vectorised**

```
> x > 2
[1] FALSE FALSE  TRUE  TRUE
> y < 7
[1]  TRUE FALSE FALSE FALSE
> any(y < 7)
[1] TRUE
> all(y < 7)
[1] FALSE
```

- **Functions working on vectors**

```
> log(x)
[1] 0.0000000 0.6931472 1.0986123 1.3862944
> x^2
[1]  1  4  9 16
> sin(x)
[1]  0.8414710  0.9092974  0.1411200 -0.7568025
```

- **MATRICES**
    - The `*` multiplication operator is a vectorised/elementwise multiplication
    - **To perform the matrix multiplication you might expect, use the `%*%` operator**

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> m
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> m * m
     [,1] [,2]
[1,]    1    9
[2,]    4   16
> m %*% m
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```

SLIDE: Challenge 21

```
> v = 1:10000
> v <- 1/(v^2)
> sum(v)
[1] 1.644834
```

Red sticky for a question or issue   Green sticky if complete

10. Functions

- **SLIDE: Learning objectives**
- **YOU'VE ALREADY BEEN USING FUNCTIONS (e.g. `log()`)** and, I hope, have found them useful
- Functions let us run a complex series of commands in one go
  - **You wouldn't want to have to repeat the calculations for `log()` each time**
  - They keep the operation under a memorable or descriptive name, **which makes the code readable and understandable**, and they are invoked with that name
  - There are a **defined set of inputs and outputs** for a function, so **WE KNOW WHAT BEHAVIOUR TO EXPECT**

- **SLIDE: Why Functions?**
- Functions let us **run a complex series of logically- or functionally-RELATED commands in one go**

- It helps when functions have **descriptive and memorable names**, as this makes code **READABLE AND UNDERSTANDABLE**

- We invoke functions with their name

- We **expect functions to have A DEFINED SET OF INPUTS AND OUTPUTS** - aids clarity and understanding

- **FUNCTIONS ARE THE BUILDING BLOCKS OF PROGRAMMING**

- As a **rule of thumb** it is good to write small functions with one obvious, clearly-defined task.

    - As you will see **we can chain smaller functions together to manage complexity**

- **SLIDE: Defining a Function**

- Functions have a **STANDARD FORM**

    - We **declare a** **`<function_name>`**
    - We use the `function` *function*/keyword to assign the function to `<function_name>`
    - Inputs (*arguments*) to a function are defined in parentheses: **These are defined as variables for use within the function AND DO NOT EXIST OUTSIDE THE FUNCTION**
    - The code block (**curly braces**) encloses the function code, the *function body*.
    - **NOTE THE INDENTATION** - *Easier to read, but does not affect execution*
    - The code `<does_something>`
    - The `return()` function returns the value, when the function is called

- **DEMO IN SCRIPT**

    - **Create new script** **`functions.R`**
    - Write and `Source`

```
# Example function
my_sum <- function(a, b) {
  the_sum <- a + b
  return(the_sum)
}
```

- **DEMO IN CONSOLE**

```
> my_sum(3, 7)
[1] 10
> a
Error: object 'a' not found
> b
Error: object 'b' not found
```

- **DEMO IN SCRIPT**
    - Let's define another function: convert temperature from fahrenheit to Kelvin

```
# Fahrenheit to Kelvin
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

- **DEMO IN SCRIPT**

```
> fahr_to_kelvin(32)
[1] 273.15
> fahr_to_kelvin(-40)
[1] 233.15
> fahr_to_kelvin(212)
[1] 373.15
> temp
Error: object 'temp' not found
```

- **LET'S MAKE ANOTHER FUNCTION CONVERTING KELVIN TO CELSIUS**
- **DEMO IN SCRIPT**
    - `Source` the script

```
# Kelvin to Celsius
kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}
```

- **DEMO IN CONSOLE**

```
> kelvin_to_celsius(273.15)
[1] 0
> kelvin_to_celsius(233.15)
[1] -40
> kelvin_to_celsius(373.15)
[1] 100
```

- **WE COULD DEFINE A NEW FUNCTION TO CONVERT FAHRENHEIT TO CELSIUS**
    - **But it's easier to combine the two functions we've already written**
- **DEMO IN CONSOLE**

```
> fahr_to_kelvin(212)
[1] 373.15
> kelvin_to_celsius(fahr_to_kelvin(212))
[1] 100
```

- **DEMO IN SCRIPT**

```
# Fahrenheit to Celsius
fahr_to_celsius <- function(temp) {
  celsius <- kelvin_to_celsius(fahr_to_kelvin(temp))
  return(celsius)
}
```

- **DEMO IN CONSOLE**
  - **NOTE: AUTOMATICALLY TAKES ADVANTAGE OF `R` 'S VECTORISATION**

```
> fahr_to_celsius(212)
[1] 100
> fahr_to_celsius(32)
[1] 0
> fahr_to_celsius(-40)
[1] -40
> fahr_to_celsius(c(-40, 32, 212))
[1] -40   0 100
```

SLIDE: Documentation

- It's important to have well-named functions (this is itself a form of documentation)
- But it's **not a detailed explanation**
- You've found `R` 's help useful, but it doesn't exist for your functions until you write it
- **YOUR FUTURE SELF WILL THANK YOU FOR DOING IT!**
- **SOME GOOD PRINCIPLES TO FOLLOW WHEN WRITING DOCUMENTATION ARE:**
  - Say what the code does (and why) - *more important than **how** *
  - Define your inputs and outputs
  - Provide an example
- **DEMO IN CONSOLE**

```
> ?fahr_to_celsius
No documentation for ‘fahr_to_celsius’ in specified packages and libraries:
you could try ‘??fahr_to_celsius’
> ??fahr_to_celsius
```

- **DEMO IN SCRIPT**
    - We add documentation as comment strings in the function
    - **SOURCE** the script

```
# Fahrenheit to Celsius
fahr_to_celsius <- function(temp) {
  # Convert input temperature from fahrenheit to celsius scale
  #
  # temp        - numeric
  #
  # Example:
  # > fahr_to_celsius(c(-40, 32, 212))
  # [1] -40   0 100
  celsius <- kelvin_to_celsius(fahr_to_kelvin(temp))
  return(celsius)
}
```

- **DEMO IN CONSOLE**
    - We read the documentation by providing the function name **only**

```
> fahr_to_celsius
function(temp) {
  # Convert input temperature from fahrenheit to celsius scale
  #
  # temp        - numeric
  #
  # Example:
  # > fahr_to_celsius(c(-40, 32, 212))
  # [1] -40   0 100
  celsius <- kelvin_to_celsius(fahr_to_kelvin(temp))
  return(celsius)
}
```

- **SLIDE: Function Arguments**
- **DEMO IN SCRIPT** (`functions.R`)
    - `Source` script

```
# Calculate total GDP in gapminder data
calcGDP <- function(data) {
  # Returns the gapminder data with additional column of total GDP
  #
  # data            - gapminder dataframe
  #
  # Example:
  # gapminderGDP <- calcGDP(gapminder)
  gdp <- gapminder %>% mutate(gdp=pop * gdpPercap)
  return(gdp)
}
```

- **DEMO IN CONSOLE**

```
> calcGDP(gapminder)
Error in gapminder %>% mutate(gdp = pop * gdpPercap) :
  could not find function "%>%"
```

- **WHAT HAPPENED?**
  - **The code in the `functions.R` file doesn't know about `dplyr`**
  - We need to **import the module in our script**
  - Use the `require()` function
- **DEMO IN SCRIPT** (`functions.R`)
  - Place `require()` calls at the top of your script
  - `Source` script

```
require(dplyr)
```

- **DEMO IN CONSOLE**
  - The new column has been added

```
> head(calcGDP(gapminder))
      country year      pop continent lifeExp gdpPercap         gdp
1 Afghanistan 1952  8425333      Asia  28.801  779.4453  6567086330
2 Afghanistan 1957  9240934      Asia  30.332  820.8530  7585448670
3 Afghanistan 1962 10267083      Asia  31.997  853.1007  8758855797
4 Afghanistan 1967 11537966      Asia  34.020  836.1971  9648014150
5 Afghanistan 1972 13079460      Asia  36.088  739.9811  9678553274
6 Afghanistan 1977 14880372      Asia  38.438  786.1134 11697659231
```

- **So, that's *all* the `gapminder` data - but what if we want to get the data by year?**
- **DEMO IN SCRIPT** (`functions.R`)
  - `Source` script

```
> source('~/Desktop/swc-r-lesson/scripts/functions.R')
> head(calcGDP(gapminder, 2002))
      country year      pop continent lifeExp  gdpPercap          gdp
1 Afghanistan 2002 25268405      Asia  42.129   726.7341  18363410424
2     Albania 2002  3508512    Europe  75.651  4604.2117  16153932130
3     Algeria 2002 31287142    Africa  70.994  5288.0404 165447670333
4      Angola 2002 10866106    Africa  41.003  2773.2873  30134833901
5   Argentina 2002 38331121  Americas  74.340  8797.6407 337223430800
6   Australia 2002 19546792   Oceania  80.370 30687.7547 599847158654
> head(calcGDP(gapminder, c(1997, 2002)))
      country year      pop continent lifeExp gdpPercap          gdp
1 Afghanistan 1997 22227415      Asia  41.763  635.3414  14121995875
2 Afghanistan 2002 25268405      Asia  42.129  726.7341  18363410424
3     Albania 1997  3428038    Europe  72.950 3193.0546  10945912519
4     Albania 2002  3508512    Europe  75.651 4604.2117  16153932130
5     Algeria 1997 29072015    Africa  69.152 4797.2951 139467033682
6     Algeria 2002 31287142    Africa  70.994 5288.0404 165447670333
> head(calcGDP(gapminder))
 Show Traceback

 Rerun with Debug
 Error in filter_impl(.data, quo) :
  Evaluation error: argument "year_in" is missing, with no default.
```

- **Now we have an issue - NO YEAR PROVIDED MEANS NO OUTPUT**
    - We need to handle this
    - 1 - **PROVIDE A DEFAULT VALUE** (`NULL`)
    - 2 - **TEST FOR VALUE AND TAKE ALTERNATIVE ACTIONS**
- **DEMO IN SCRIPT**
    - `Source` script

```
# Calculate total GDP in gapminder data
calcGDP <- function(data, year_in=NULL) {
  # Returns the gapminder data with additional column of total GDP
  #
  # data           - gapminder dataframe
  # year_in        - year(s) to report data
  #
  # Example:
  # gapminderGDP <- calcGDP(gapminder)
  gdp <- gapminder %>% mutate(gdp=(pop * gdpPercap))
  if (!is.null(year_in)) {
    gdp <- gdp %>% filter(year %in% year_in)
  }
  return(gdp)
}
```

- **DEMO IN CONSOLE**

```
> source('~/Desktop/swc-r-lesson/scripts/functions.R')
> head(calcGDP(gapminder))
[1] country   year      pop       continent lifeExp   gdpPercap gdp
<0 rows> (or 0-length row.names)
> head(calcGDP(gapminder))
      country year      pop continent lifeExp gdpPercap         gdp
1 Afghanistan 1952  8425333      Asia  28.801  779.4453  6567086330
2 Afghanistan 1957  9240934      Asia  30.332  820.8530  7585448670
3 Afghanistan 1962 10267083      Asia  31.997  853.1007  8758855797
4 Afghanistan 1967 11537966      Asia  34.020  836.1971  9648014150
5 Afghanistan 1972 13079460      Asia  36.088  739.9811  9678553274
6 Afghanistan 1977 14880372      Asia  38.438  786.1134 11697659231
> head(calcGDP(gapminder, year_in=2002))
      country year      pop continent lifeExp  gdpPercap          gdp
1 Afghanistan 2002 25268405      Asia  42.129   726.7341  18363410424
2     Albania 2002  3508512    Europe  75.651  4604.2117  16153932130
3     Algeria 2002 31287142    Africa  70.994  5288.0404 165447670333
4      Angola 2002 10866106    Africa  41.003  2773.2873  30134833901
5   Argentina 2002 38331121  Americas  74.340  8797.6407 337223430800
6   Australia 2002 19546792   Oceania  80.370 30687.7547 599847158654
```

- **Now let's do the same for country**
- **DEMO IN SCRIPT**
  - `Source` script

```r
# Calculate total GDP in gapminder data
calcGDP <- function(data, year_in=NULL, country_in=NULL) {
  # Returns the gapminder data with additional column of total GDP
  #
  # data             - gapminder dataframe
  # year_in          - year(s) to report data
  #
  # Example:
  # gapminderGDP <- calcGDP(gapminder)
  gdp <- gapminder %>% mutate(gdp=(pop * gdpPercap))
  if (!is.null(year_in)) {
    gdp <- gdp %>% filter(year %in% year_in)
  }
  if (!is.null(country_in)) {
    gdp <- gdp %>% filter(country %in% country_in)
  }
  return(gdp)
}
```

- **DEMO IN CONSOLE**

```
> source('~/Desktop/swc-r-lesson/scripts/functions.R')
> head(calcGDP(gapminder))
      country year      pop continent lifeExp gdpPercap         gdp
1 Afghanistan 1952  8425333      Asia  28.801  779.4453  6567086330
2 Afghanistan 1957  9240934      Asia  30.332  820.8530  7585448670
3 Afghanistan 1962 10267083      Asia  31.997  853.1007  8758855797
4 Afghanistan 1967 11537966      Asia  34.020  836.1971  9648014150
5 Afghanistan 1972 13079460      Asia  36.088  739.9811  9678553274
6 Afghanistan 1977 14880372      Asia  38.438  786.1134 11697659231
> head(calcGDP(gapminder, 1957))
      country year      pop continent lifeExp gdpPercap          gdp
1 Afghanistan 1957  9240934      Asia  30.332   820.853   7585448670
2     Albania 1957  1476505    Europe  59.280  1942.284   2867792398
3     Algeria 1957 10270856    Africa  45.685  3013.976  30956113720
4      Angola 1957  4561361    Africa  31.999  3827.940  17460618347
5   Argentina 1957 19610538  Americas  64.399  6856.856 134466639306
6   Australia 1957  9712569   Oceania  70.330 10949.650 106349227169
> head(calcGDP(gapminder, 1957, "Egypt"))
  country year      pop continent lifeExp gdpPercap         gdp
1   Egypt 1957 25009741    Africa  44.444  1458.915 36487093094
> head(calcGDP(gapminder, "Egypt"))
[1] country   year      pop       continent lifeExp   gdpPercap gdp
<0 rows> (or 0-length row.names)
> head(calcGDP(gapminder, country_in="Egypt"))
  country year      pop continent lifeExp gdpPercap          gdp
1   Egypt 1952 22223309    Africa  41.893  1418.822  31530929611
2   Egypt 1957 25009741    Africa  44.444  1458.915  36487093094
3   Egypt 1962 28173309    Africa  46.992  1693.336  47706874227
4   Egypt 1967 31681188    Africa  49.293  1814.881  57497577541
5   Egypt 1972 34807417    Africa  51.137  2024.008  70450495584
6   Egypt 1977 38783863    Africa  53.319  2785.494 108032201472
```

- **SLIDE: Challenge 23**

```r
# Plot grid of country life expectancy
plotLifeExp <- function(data, letter=letters, wrap=FALSE) {
  # Return ggplot2 chart of life expectancy against year
  #
  # data         - gapminder dataframe
  # letter       - start letters for countries
  # wrap         - logical: wrap graphs by country
  #
  # Example:
  # > plotLifeExp(gapminder, c('A', 'Z'), wrap=TRUE)
  starts.with <- substr(data$country, start = 1, stop = 1)
  az.countries <- data[starts.with %in% letter, ]
  p <- ggplot(az.countries, aes(x=year, y=lifeExp, colour=country))
  p <- p + geom_line()
  if (wrap) {
    p <- p + facet_wrap(~country)
  }
  return(p)
}
```

Red sticky for a question or issue

Green sticky if complete

11. Dynamic Reports

SLIDE: Learning Objectives

- In this final section, we'll be learning how to **create reproducible, attractive, dynamic reports**
- To do so, we'll learn some **markdown syntax**, and how to put **working R code** into a document
- We'll also look at **generating the report in a number of file formats**, for sharing.

SLIDE: Literate Programming

- What we're about to do is an example of **Literate Programming**, a concept introduced by Donald Knuth
- The idea of Literate Programming is that
    - **The program or analysis is explained in natural language**
    - **The code needed to run the program/analysis is embedded in the document**
    - **The whole document is executable**

- We can produce these documents in `RStudio`

SLIDE: Create an `R Markdown` file

- In `R`, literate programming is **implemented in `R Markdown` files
- To create one: `File` $\rightarrow$ `New File` $\rightarrow$ `R Markdown`

  - There is a dialog box - **enter a title** (`Literate Programming`)
  - Save the file (`Ctrl-S`) - **create new subdirectory (`markdown`)** - `literate_programming.Rmd`

- The file **gets the extension `.Rmd`**

  - The **file is autopopulated with example text**

SLIDE: Components of an `R Markdown` file

- The **HEADER REGION IS FENCED BY `---`**

  - **Metadata** (author, title, date)
  - Requested **output format**

```
---
title: "Literate Programming"
author: "Leighton Pritchard"
date: "04/12/2017"
output: html_document
---
```

- Natural language is written as plain text, **with some extra characters to define formatting**

  - **NOTE THE HASHES `#`, ASTERISKS `*` AND ANGLED BRACKETS `<>`**

- `R` code runs in the document, and is **fenced by backticks**

- **CLICK ON `KNIT`**

  - A new (pretty) document is produced in a new window

- **CROSS REFERENCE MARKDOWN TO DOCUMENT**

  - **Title, Author, Date**
  - **Header**
  - **Link**
  - **Bold**
  - **`R` code and output**

    - **Plots**
- **CLICK ON `KNIT TO PDF`**
    - A new `.pdf` document opens in a new window
- **CROSS REFERENCE MARKDOWN TO DOCUMENT**
    - **NOTE:** The formatting isn't identical
- **CLICK ON `KNIT TO WORD`**
    - A new `Word` document opens up
- **CROSS REFERENCE MARKDOWN TO DOCUMENT**
    - **NOTE:** The formatting isn't identical
- **NOTE THE LOCATION OF THE OUTPUT FILES - ALL IN THE SOURCE DIRECTORY**
    - **CLOSE THE OUTPUT**

SLIDE: Creating a Report

- We'll **create a report on the `gapminder` data**
- **DELETE THE EXISTING TEXT/CODE CHUNKS** (`literate_programming.Rmd`)
    - **Change the title** (`Life Expectancies`)
    - **Define the input data location in the `setup` section**
        - Code in the `setup` section is run, but not shown (**knit to demo**)
        - `include = FALSE`
    - **Write introduction and KNIT**
        - Header notation with the hash `#`
        - Inline `R` to name the data used
        - **We can define the location of the data in one place, and reuse the variable/have it propagate when we update the data**
        - Import the data in `setup`
    - **Write next section** (`Life expectancy in countries`)
        - `Source` the `functions.R` file to get our solution to Challenge 23 (`plotLifeExp`)
        - Use the imported function
        - `{r echo=FALSE}` shows output but not the code

- **Change the letters**

    - Change the letters to something else
    - Re-run the document

- **Add Numbered Table of Contents (where possible)**

    - Make the required changes in the header

```
---
title: "Life Expectancies"
author: "Leighton Pritchard"
date: "04/12/2017"
output:
  pdf_document:
    toc: true
    number_sections: true
  html_document:
    toc: true
    toc_float: true
    number_sections: true
  word_document:
    toc: true
---

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

# Path to gapminder data
datapath <- "../data/gapminder-FiveYearData.csv"

# Letters to report on
az <- c('G', 'Y', 'R')

# Load gapminder data
gapminder <- read.csv(datapath, sep=",", header=TRUE)

# Source functions from earlier lesson
source("../scripts/functions.R")
```

Introduction

We will present the life expectancies over time in a set of countries, using the gapminder data in the file `r datapath`.

We will specifically focus on countries beginning with the letters: `r az`.
```

Life expectancy in `r az` countries

In countries starting with these letters, the life expectancy is as plotted below.

We use the code from our earlier challenge solution

```
plotLifeExp
```

```
plotLifeExp(gapminder, az, wrap=TRUE)
```