r-lesson.md - 2021-05-11-strathclyde-online

START THE SLIDES

---

1. DATA ANALYSIS

Data Analysis in the Scientific Cycle

- The **CYCLE OF SCIENCE** is often presented as:
    - Formulate a hypothesis
    - Test the hypotheses
    - Gather data
    - Analyse data
    - Update the hypothesis
    - and go round again
- But I'd like to suggest:
    - You **can't formulate a hypothesis without having first made some observations**
    - This scientific cycle has always started - if unacknowledged - with data collection and analysis
        - Darwin and Wallace had to make long journeys and many observations before they formed their hypotheses of descent with selection
    - **In the age of data-intensive research, the focus is very much on open-minded data analysis to formulate hypotheses *as a first step* in the cycle**
        - **It is especially important that your initial exploration is recorded, replicable and systematic**
- With the increasing amount of freely-available public data - especially genome data - you can get started on formulating hypotheses more easily than ever before

---

Data-Intensive Research

- You're probably here because you are doing, or maybe aim to do, data-intensive research
    - Science and the humanities are both becoming much more data-driven
    - Researchers who don't identify as "data scientists" are now having to work with large, complex data
- Early-career training may not have prepared everyone equally well for this
    - You may have had courses on basic statistical methods (t-tests, ANOVA), and specific software tools
    - Who has had training in developing a reproducible workflow, or coding a new analysis **ASK FOR HANDS/GET RESPONSES**

- Research workflows are central to systematic, replicable and reproducible research
    - But they're often left out of basic training
- The core skills that help us create these workflows are:
    - Design principles - what data is and how it is best processed
    - Software development methods - how to automate repetitive stuff with a computer

---

Pipelines and Workflows

- We can distinguish between two terms that often get used interchangeably: "pipelines" and "workflows"
- "Pipelines" are a set of instructions - run by your computer - that take a dataset and *pipe* it through some processes, until a result emerges from the end.
- "Workflows" are the processes that researchers follow to investigate a problem
    - A workflow might involve using one or more pipelines
    - But it includes things like exploring your data, forming hypotheses, writing code, and interpreting your results
- In this course, we're concerned with the whole workflow

Explore, Refine, Produce (ERP)

- One way to think about your workflow is as an "Explore, Refine, Produce (ERP)" cycle.
- In the Explore phase, you process and interrogate your data to identify potential solutions
- In the Refine phase, you narrow down your focus to the most promising approaches
- The Produce phase is really a companion to the Explore and Refine phases
    - you should always be recording your work and preparing it for future consumption either by yourself or others
- You can think of:
    - The Explore phase as being centred on you, as a researcher, as you dig deep into your data
    - The Refine phase as being how you and your team move towards a satisfying solution
    - and the Produce phase as being how you communicate your results to teh wider community
- In this course, we'll be covering several aspects of this model
    - Our goal is to empower you to explore and get the most out of your data, in a replicable, reproducible, and systematic way
- We'll be covering cleaning, organising, managing, and visualising your data - covering the Explore and Refine phases

    - We'll also try to cover documentation and using RMarkdown to integrate documentation and code - to help support the Refine and Produce phases

- That's probably enough context - let's get to the code.

---

2. WELCOME TO `R`

- Welcome to `R`!

Learning Objectives

Our goals in this course are:

- to introduce you to the fundamentals of `R` and `RStudio`, and of programming
- how to manage and process your data with an approach known as the `tidyverse`
- how to produce publication-quality data visualisations with the `ggplot2` package
- and how to combine documentation with analysis and code, using `RMarkdown`
- Working with a programming language (especially if it’s your first time) can be intimidating
    - But knowing even a little bit about `R` and how to manage and handle your own data is incredibly empowering
    - The rewards outweigh any frustrations.
- It's not a secret, but even experienced programmers find coding difficult and frustrating at times
    - I've been coding for 40 years and - though I know a bit more than when I started - every time a new language or technique turns up that I want to try, I'm a beginner all over again and feel like I know nothing.
    - So don't let intimidation stop you
    - With time and practice it just gets easier and easier to accomplish what you want.
- Learning to code opens up the full possibilities of computing for analysing your data and automating your work
    - Think of it this way: if you could only do biotechnology using kits you bought off Sigma, you could probably get a lot done.
    - But - if you don’t understand the biochemistry of the kit - how can you tell it's not working? How can you troubleshoot?
    - And how could you do experiments for which there are no kits?
    - Knowing a bit of code lets you troubleshoot, and go beyond kits to create new analyses
- R is one of the most widely-used and powerful programming languages, and it's popular in many areas.
    - R is specifically a statistical language
        - It's got a strong focus on data representation, as you'll see shortly

        - Particularly good for the generation of publication-quality graphs and figures.
    - It is a *general programming language*, though. Most of the concepts you will learn apply to Python and other programming languages.
        - Once you understand programming concepts in one language, it's easier to move across to other languages.

- However, R is not the easiest-to-learn programming language ever created.
    - Please don't get too discouraged! Some bits and concepts *are* just a bit more difficult than others
    - Even with the modest amount of R we will cover today and tomorrow, you can start using some sophisticated R software packages, have a general sense of how to interpret an R script, and be able to get your own data into a workable form for more sophisticated exploration.

- Get through these lessons, and you will be on your way to being an accomplished R user!

- The other big "secret" of programming is that you can only learn so much by reading about it. You have to do it - and get things wrong - to learn.
    - Do the exercises in class, re-do them on your own, and then work on your own problems, and you'll be expert in no time!

- **Let's get started.**

## What is `R`?

- `R` is a **PROGRAMMING LANGUAGE**, and the **SOFTWARE** that runs programs written in that language.

- `R` is **AVAILABLE ON ALL MAJOR OPERATING SYSTEMS**

- This can sometimes be confusing - **IF AT ANY POINT I AM UNCLEAR, PLEASE ASK!**

- **WHY DO WE USE AND TEACH `R`?**
    - `R` is **FREE AS IN "FREE SAMPLES AT THE SUPERMARKET"**, and very **WIDELY-USED** across a range of disciplines.
    - There are **MANY USEFUL PACKAGES** or data analysis and statistics, **WRITTEN By EXPERTS** at the cutting edge of their fields
    - 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

---

What is `RStudio`?

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

- On the left is a screenshot **TAKEN 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**

Why not use `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 - and by the computer.
  - You can give your analysis to someone else and they can rerun it, or improve on it
  - 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.
- `R` code can also be **RUN ON EXTREMELY LARGE DATASETS**, quickly. That's much harder in `Excel`.

`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**

- **REMEMBER THE WINDOWS ARE MOBILE AND PEOPLE COULD HAVE THEM IN ANY CONFIGURATION - THE EXACT ARRANGEMENT IS UNIMPORTANT**

- 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 +
+

>
```

- **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
```

- 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
```

- **THE ORDER/CONSTRUCTION OF MATHEMATICAL OPERATIONS CAN MATTER**
  - Write somewhere if possible: $a = \log(0.01^{200})$, $b = 200 \times \log(0.01)$
  - These two mathematical expressions are exactly equal: $a = b$
  - But computers are not mathematicians, they're machines. Numbers are susceptible to this *rounding error*, so what happens is this:

```
> log(0.01^200)
[1] -Inf
> 200 * log(0.01)
[1] -921.034
```

- **COMPUTERS DO WHAT YOU TELL THEM, NOT NECESSARILY WHAT YOU WANT**

Variables

- **VARIABLES** are critical to programming in general, and also to working in `R`
- To do interesting and useful things, we need a way to label and refer to a piece of data
  - **We use *variables* for this**
- 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)
- Like boxes, a variable **CAN BE EMPTY**
- We can also take the contents out and replace them with something else, but **KEEP THE NAME**

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
```

- We can also print the output by putting the expression in parentheses

```
> (x <- 1/40)
[1] 0.025
```

- **CLICK ON THE ENVIRONMENT WINDOW**
  - You should see that `x` is now 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 VALUE**

```
> x + x
[1] 0.05
> 2 * x
[1] 0.05
> x ^ 2
[1] 0.000625
```

- such as an argument to a function

```
> log(x)
[1] -3.688879
> sin(x)
[1] 0.0249974
```

- 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
> x
[1] 105
```

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

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

## Naming Variables

- **VARIABLE NAMES ARE DOCUMENTATION**
  - they should describe what the data represents
    - I expect you can tell, just by reading, what all these variables represent
  - **This makes the code more readable** - you can track the operations on a specific bit of data
  - ("anonymous" variables are also helpful, e.g. `x` or `i`)

- There are some rules and guidelines for allowed and effective variable names
  - The variable names should be explicit, but not too long
  - In `R` names **cannot** start with a digit
  - `R` is also case sensitive, so `Weight` with a capital letter is not the same name as `weight` with a lower-case letter
  - Problems will arise if you call any variables by the name of an existing `R` function
  - You should use consistent styling to handle multiple words

---

Functions

- You've already used some **functions** (`log()`, `sin()`, etc.). These are like "canned scripts"

- Functions have **THREE MAIN PURPOSES**
  - They **ENCAPSULATE AND AUTOMATE 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 that describe an action, 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

- 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`

- **ALL THE FUNCTIONS YOU'VE SEEN SO FAR ARE BUILT-IN**, the so-called `base` functions

- **YOU CAN WRITE YOUR OWN FUNCTIONS**

- **OTHER FUNCTIONS FOR SPECIFIC TASKS CAN BE BROUGHT IN, THROUGH `libraries`**

## Getting Help in `R`: INTERACTIVE DEMO

- **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")
```

Challenge 01

Link: https://strathsci.qualtrics.com/jfe/form/SV_eG17F5atskDqbC6

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

---

3. PROJECT MANAGEMENT IN R

How Projects Tend To Grow

- I'm sure you all recognise this situation
- Research tends to be incremental
  - Projects start out as random notes here, a bit of code there, some data over there
  - and eventually a manuscript is written
- There are many reasons why we should **ALWAYS** avoid working in a random manner, and naming files like they do in the cartoon.
  - Basically, good practice ensures reproducibility and makes everyone's lives easier.

## Good Practice

- 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 raw data in a **SEPARATE SUBFOLDER**
- **CLEAN THE DATA PROGRAMMATICALLY** (part of the analysis chain)
    - Most of the time, your data will be "dirty" - it won't be in the form necessary for your analysis.
    - It is often said that 80% of data analysis is spent on the process of cleaning and preparing the data
    - 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
    - keep in its own subdirectory

Example Directory Structure

- This is **ONE OF MANY WAYS 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)
    - Name this after your project
- `data/` is a subdirectory for storing data
    - raw data only, or raw and intermediate data? **YOUR DECISION**

  - store data and metadata (data *about* the data) together
  - `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, supporting material, 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 the executable code that automates your analysis
- The important thing is that the structure is **AS SELF-EXPLANATORY AS POSSIBLE**

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
- We're going to create a **NEW PROJECT** in `RStudio`
- **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 `ibioic-r-lesson`**
  - We are asked for a parent directory. **PUT YOURS ON THE DESKTOP; STUDENTS CAN CHOOSE ANYWHERE SENSIBLE**
- 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)
        - **CHANGE CWD IF NECESSARY**
        - **SHOW TERMINAL**
    - **ONE FILE**: `*.Rproj` - information about your project
- **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
    - Do the same for `data/`
- **NOTE THAT WE WILL NOW POPULATE THE DIRECTORY**

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`
- **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**)
- **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).

---

4. A FIRST ANALYSIS IN `RSTUDIO`

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**
- **CHECK EVERYONE'S READY TO PROCEED**

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

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**
    - **WE SHOULD RECORD THIS SOMEWHERE AS METADATA**
- **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` window

```
# 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 V24 V25 V26
1  0  0  1  3  1  2  4  7  8   3   3   3  10   5   7   4   7   7  12  18
6  13  11  11   7   7
2  0  1  2  1  2  1  3  2  2   6  10  11   5   9   4   4   7  16   8   6
18   4  12   5  12   7
  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
```

Challenge 02

*-SOLUTION**

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

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

Indexing Data

- Our inflammation is in a **TABLE**, also known as an **ARRAY** or **MATRIX**
- There is a **SET OF CONVENTIONS FOR REFERRING TO THIS DATA**
    - `R` keeps to these conventions
- If we want to refer to a particular row (equivalent to a patient), we refer to the number down the side, in green
    - Patient 2 is row 2
- If we want to refer to a particular column (equivalent to a day of study), we refer to the number along the top, in magenta
    - Day 30 is column 30
- To refer to a specific combination of patient and day, we need to refer to the row first, then the column
    - Patient 3 on day 2 is the element *a32*, here.

Indexing Data - INTERACTIVE DEMO

- **HOW DO WE GET ACCESS TO A SUBSET OF THE 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 V24 V25 V26
5  0  1  1  3  3  1  3  5  2   4   4   7   6   5   3  10   8  10   6  17
9  14   9   7  13   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  6  7  6 14
[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 12  9
```

## 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
```

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.

    - What we'd like to do is **APPLY THE SAME FUNCTION TO EACH ROW**
- Happily **COMPUTERS WERE INVENTED TO SAVE US THE HASSLE**
- We could automate this task in any of several ways available in `R`
    - `R` has an `apply()` function exactly for this
    - **IN THE CONSOLE**
- `apply()` takes three arguments:
    - `X` is your dataset
    - `MARGIN` is the *axis* of the dataset
        - rows are `1`
        - columns are `2`
        - datasets can be multidimensional, so the margins can go higher than 2
    - `FUN` is the function we want to apply
- So, here, we are applying the `mean()` function to each `row` in the dataset called `data`
    - This gives us the average inflammation per patient

```
> 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 5.750 5.225
[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 6.175 6.550
[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 5.900 6.750
[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 7.050 5.900
```

- **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

```
# 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 5.750 5.225
[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 6.175 6.550
[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 5.900 6.750
[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 7.050 5.900
> colMeans(data)
        V1         V2         V3         V4         V5         V6
V7         V8         V9
 0.0000000  0.4500000  1.1166667  1.7500000  2.4333333  3.1500000
3.8000000  3.8833333  5.2333333
       V10        V11        V12        V13        V14        V15
V16        V17        V18
 5.5166667  5.9500000  5.9000000  8.3500000  7.7333333  8.3666667
9.5000000  9.5833333 10.6333333
       V19        V20        V21        V22        V23        V24
V25        V26        V27
11.5666667 12.3500000 13.2500000 11.9666667 11.0333333 10.1666667
10.0000000  8.6666667  9.1500000
       V28        V29        V30        V31        V32        V33
V34        V35        V36
 7.2500000  7.3333333  6.5833333  6.0666667  5.9500000  5.1166667
3.6000000  3.3000000  3.5666667
       V37        V38        V39        V40
 2.4833333  1.5000000  1.1333333  0.5666667
```

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
    - `ggplot2` is built out of base graphics

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

```
# 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 histogram 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 histogram 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**
  - **THE ANALYSIS IS REPRODUCIBLE ON NEW DATA**

Challenge 03 (5min)

```
# Plot standard deviation by day
plot(apply(data, 2, sd))
hist(avg_inflammation_day, breaks=seq(5, 8, by=0.2))
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

4. Data Types and Structures in R

Learning Objectives**

- This lesson might sound like it's going to be a bit dry: "Why should I care about data types and structures?"
- **IF YOU UNDERSTAND YOUR DATA, AND YOU UNDERSTAND HOW R SEES YOUR DATA, YOUR ANALYSIS WILL BE MUCH EASIER AND MORE EFFECTIVE**
- 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.

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 always assuming that you mean to import *data* not *words*
  - 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.

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*

Data Types in R

- R's data *types* are *atomic*: they are **FUNDAMENTAL AND EVERYTHING ELSE IS BUILT UP FROM THEM**, the same way 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
- **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
```

Challenge 04 (2min)

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

*-SOLUTION**

```
> answer = TRUE
> height = 183
> dog_name = "Spot"
> is.logical(answer)
[1] TRUE
> is.numeric(height)
[1] TRUE
> is.character(dog_name)
[1] TRUE
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

FOUR COMMON R DATA STRUCTURES**

- These are perhaps the four data structures 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

---

## Challenge 05 (5min)

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

*-SOLUTION**

```
> xx <- c(1.7, "a")
> typeof(xx)
[1] "character"
> yy <- c(TRUE, 2)
> typeof(yy)
[1] "double"
> zz <- c("a", TRUE)
> typeof(zz)
[1] "character"
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

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
```

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

```
# 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
```

Challenge 06 (5min)

```
> 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", "case"))
> str(f)
 Factor w/ 2 levels "control","case": 2 1 2 1 2
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

Lists

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

```
# Create a list
l <- list(1, 'a', TRUE, seq(2, 5), 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
 $ : int [1:4] 2 3 4 5
 $ : 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 3 4 5

[[5]]
[1] no  yes no
Levels: no yes
> l[[4]][1]
[1] 2
```

- **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/READABILITY)**

```
> 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"
```

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
```

---

5. 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

---

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
> typeof(cats)
[1] "list"
> cats[[1]]
[1] calico black  tabby
Levels: black calico tabby
> names(cats)
[1] "coat"          "weight"       "likes_string"
> class(cats$coat)
[1] "factor"
> class(cats$weight)
[1] "numeric"
> class(cats$likes_string)
[1] "integer"
```

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**

Creating a `data.frame`

- **DEMO IN SCRIPT**
  - `Run` when done

```
# 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                            
```

## Saving a `data.frame`

- **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

Loading 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)
```

- **CHECK THE DATA IN THE `Environment` TAB**
  - Click on `gapminder` in `Evironment` tab.
  - **NOTE COLUMNS**

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
```

7. PACKAGES

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)
```

Challenge 07 (5min)

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

## 8. CREATING PUBLICATION-QUALITY GRAPHICS

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**

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**

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`!**

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)

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***

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

```
p + geom_line()
```

- This looks terrible. **CHANGE IT BACK**
  - Preparing these plots in a script allows us to explore large and small differences in representation easily, and is very flexible

---

Challenge 08 (2min)

```
# 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

---

What is a Plot? *layers*

- Without drawing attention to 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**

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**

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**

Challenge 09 (5min)

```
# 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

---

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()
```

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")
```

Multi-panel figures

- All our plots so far have been single figures, but **multi-panel plots can give clearer comparisons
  - These are also known as *small multiples plots*
- 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)
```

Challenge 10 (5min)

```
# 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

---

9. Data Cleaning/Tidy Data

## Why Tidy Data?

- It is **not just a first step** - it must be repeated many time over the course of an analysis
  - new data, new ideas, etc. turn up as you're working
- About **80% of the effort of data analysis** is cleaning and preparing data for analysis

- The principles of **tidy data** provide a standard way to organise data values within a dataset
    - "Tidy datasets are all alike, but every messy dataset is messy in its own way"

## A Messy Dataset (1)

- Here's a dataset like you might receive it from a colleague
- It's a **RECTANGULAR TABLE**
    - Made up of **ROWS** and **COLUMNS**, just like the data you've been working with
- Each row describes a treatment
- Each column gives the results for a different individual, for each treatment
- But the data doesn't have to be structured this way.

## A Messy Dataset (2)

- Here we've transposed the rows and columns of the table
- The **data is the same**
- The **layout is different**
- To understand what this means, **AND WHY IT IS MESSY**, we need to consider some data semantics.

## Data Semantics

- We need to define three terms
- A dataset, like the one shown, is a collection of **VALUES**
- Each value belongs to a variable, and to an observation
- A **VARIABLE** is something that *can change or vary*
    - They may be values that measure the same underlying attribute, such as height, temperature, or some kind of output result
    - They may be experimental conditions under a researcher's control, such as a treatment, how long that treatment is applied, or other experimental settings
- An **OBSERVATION** is a collection of **values** measured across all **variables** for the same individual or unit
    - The unit is often a person, a collective group like a religion or company, or a physical item like a reactor vessel

---

## Challenge 11 (2min)

- So for our first messy dataset, how would you describe the rows and columns of the table?
  - Are they observations, or variables, or neither?
- **THE ROWS AND COLUMNS IN THE MESSY DATA ARE NEITHER OBSERVATIONS NOR VARIABLES**

## A Tidy Dataset (1)

- The dataset contains 18 values:
  - six observations of three variables
- The variables are:
  - **PERSON**: John, Mary and Jane
  - **TREATMENT**: A or B
  - **RESULT**: NA, 16, 3, 2, 11, 1
- Each **OBSERVATION** includes values for all three variables

---

A Tidy Dataset (2)

- Here is the dataset represented in tidy form
- You can see each variable has its own column
- Each observation has one value per column
  - **Note: Missing data counts as a value**

Tidy Data

- Tidy data is a **STANDARD** way of structuring a dataset, but it is not the only way
  - It does make it easy to extract the variables you need
  - It is also very well suited to R because it supports vectorisation (which you saw earlier)
- In Tidy Data:
  - Each variable forms a column
  - Each observation forms a row
- Most of the data you receive is unlikely to be Tidy, so you'll probably need to clean it
- And **MESSY DATA CAN BE USEFUL**
  - If your design is completely crossed: e.g. every individual tries every medicine
    - Messy Data (rows=patients, columns=medicines) is compact
      - Also useful if matrix operations are appropriate

- **INTERACTIVE DEMO**
    - Show that the `gapminder` data is Tidy
    - **IN THE CONSOLE**
        - Each column in the `gapminder` dataset is a variable
        - Each row is a set of values: one per variable, comprising a single observation fo a combination of country and year
- **WE CAN USE TIDY DATA METHODS ON THIS DATASET**

```
> 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
```

---

10. WORKING WITH TIDY DATA

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)

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`
    - It enables group-level analyses without using repetitive code
- **AVOIDING REPETITION IMPROVES YOUR CODE**

- More **robust**
- More **readable**
- More **reproducible**

Split-Apply-Combine

- The **general principle** `dplyr` supports is **SPLIT-APPLY-COMBINE**
- We have a **dataset with several groups in a variable** (column `x`)
  - For example, each patient in our messy data might be a "group"
- 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

`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
```

`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`

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

Challenge 12

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

- 624 observations

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

`group_by()`

- The `group_by()` *verb* **SPLITS `data.frame`s INTO GROUPS ON A VARIABLE/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
```

`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
```

Challenge 13

- **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$meanlifeExp ==
max(avg_lifexp_country$meanlifeExp),]
# A tibble: 1 x 2
  country meanlifeExp
   <fctr>       <dbl>
1 Iceland    76.51142
> avg_lifexp_country[avg_lifexp_country$meanlifeExp ==
min(avg_lifexp_country$meanlifeExp),]
# 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

---

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
```

`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**

```
# 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))
```

---

13. DYNAMIC REPORTS

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**
- This makes it possible to share useful documents with people who *do* and *do not* know about coding
- Documents can be reproduced/modified easily when data changes
- We can produce these documents in `RStudio`

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**

## 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**

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)
```

- **CHANGE LETTERS IN `az`**
  - If our boss or PI comes over and says, "we need plots for countries starting with G, I and R" - it's a simple task to modify the value in the variable `az` and rerun the document
- **KNIT DOCUMENT**

---

14. CONCLUSION

You have learned

- About `R`, `RStudio` and how to set up a project
- How to load data into `R` and produce summary statistics and plots with *base* tools
- All the data types in `R`, the most important data structures
- How to install and use packages
- How to use the Tidyverse to manipulate and plot data
- How to create dynamic reports in `R`

## The End Is The Beginning

- You've learned a lot in the last couple of days
  - More than enough to be productive and save yourself a lot of time
  - More than enough to make your analyses reproducible and rerunnable
- There's a whole lot more you can do with `R`, `OpenRefine` and the shell

- This is just the beginning of a whole world opening up where you can make computers do exactly what you want, in service of your research

---

BONUS. PROGRAMMING IN R

Learning Objectives

- What we've covered so far will **get you a long way with your analyses**
  - As you saw with the Unix Shell, the real power of using computers is putting all the pieces together into larger pieces of code - scripts and programs - that can automate complex tasks in a reusable way
- To help you with this, we're going to cover some programming in R
- 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** - you'll have seen loops in the shell lesson
  - Much of the time, you can avoid using them in R data analyses, because `dplyr` exists, and because R is **vectorised**
  - But there are times when you do need them
- We'll also cover how to write *functions*, which let you package up your code into reusable chunks that you can apply again and again to different datasets.

`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`**
- **MAKE `x` 11 FIRST TO DEMONSTRATE**

- **THEN MAKE `x` 8**

```
# 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")
}
```

Challenge 14 (2min)

```
# 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

---

`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**)
  - **OR USE VECTORISATION (COMING UP)**

`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)
  }
```

Challenge 15 (2min)

```
# Challenge solution
vowels <- c('a', 'e', 'i', 'o', 'u')
for (l in letters) {
  if (l %in% vowels) {
    print(paste(l, "is a vowel"))
  } else {
    print(paste(l, "is not a vowel"))
  }
}
```

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

## 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
```

## Challenge 16 (2min)

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

🟥 Red sticky for a question or issue 🟩 Green sticky if complete

---

FUNCTIONS

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**

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**
  - This is what I mean about functions being "building blocks" of programs

- **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
```

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)
}
```

## Function Arguments

- **DEMO IN SCRIPT** (`functions.R`)
  - `Source` script

```
# Calculate total GDP in gapminder data
calcGDP <- function(data) {
  # Returns dataset with additional column of total GDP
  #
  # data            - gapminder dataframe
  #
  # Example:
  # gapminderGDP <- calcGDP(gapminder)
  gdp <- data %>% 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** so it can be used
  - 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

```
# Calculate total GDP in gapminder data
calcGDP <- function(data, year_in) {
  # 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 <- data %>%
    mutate(gdp=(pop * gdpPercap)) %>%
    filter(year %in% year_in)
  }
  return(gdp)
}
```

```
> 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 <- data %>% 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

```
# 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 <- data %>% 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
```

## Challenge 17 (10min)

```
# 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

---