2025-11-04-strathclyde 3. Creating Publication-Quality Graphics instructor notes

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 1

- 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 2

- **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* 1

- 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* 2

- 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

---

4. 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: "04/11/2025"
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**
- We can add elements of a document, e.g

```
A list:

* bold with double-asterisks
* italics with underscores
* code-type font with backticks

A second list:

- bold with double-asterisks
- italics with underscores
- code-type font with backticks

Or numbered lists

1. bold with double-asterisks
2. italics with underscores
3. code-type font with backticks

Even section headers of different sizes

# Title
## Main section
### Sub-section
#### Sub-sub section
```

- **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: "04/11/2025"
```

```
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 <- function(data, letter=letters, wrap=FALSE)
{
    starts.with <- substr(data$country, start=1, stop=1)
    az.countries <- data[starts.woth %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)
}
```

```
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**

---

12. 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

```r
# 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