Introduction to R and the Tidyverse

ISAIR 2025

What is R?

R is a free and open-source programming language for statistical computing and graphics. R is widely used among statisticians and provides access to a wide range of state of the art statistical and graphical techniques.[1]

[1] A good resource for learning R is Datacamp's Introduction to R which is free online course. A good resource for becoming useful in R is R for Data Science which can be accessed online for free. If you wanting to take your skills to the next level then there is the book Advanced R which is also available free online.

RStudio

RStudio is an integrated development environment or IDE which can be used for R programming. R is the programming language and RStudio is an extra piece of software that we will use to run R. It provides a nicer user interface and has some extra features, as you will see throughout the course.

When you open RStudio you will see there are two key regions: the Console pane on the left, and the output pane in the bottom right[2]. For now, all you need to know is that you can type commands into the console and press enter to run it.

[2] The panes can be moved around and customised by going to Tools>Global Options>Pane Layout.

Running commands in R

R requires typed instructions from it's user.[3]

[3] Unlike some software where you can use a mouse to achieve all your analyses, in R you will have to type code. This may seem like a drawback at first but after practice it becomes normal and you may become reluctant to use software where you have to point and click, as it is slow and labourious to repeat tasks.

- Interactively - Type an R command into the console and press enter
- Using a script - You can type the command into a script and either click Run or press ctrl+Enter

Exercise 1

1. Open RStudio
2. Type `4+5` into the console and press enter.
3. Type `a <- 5` and press enter.
4. Type `a` and press enter.
5. Type `a+6` and press enter.

In R you can create objects by using the assignment operator `<-`. This operator is an arrow which assigns whatever is on the right of the arrow to the name of the object on the left of the arrow.

Using Scripts

A script is a file where you can write a sequence of commands. Although R can be used interactively it is a good idea to record your commands in a script so that they can be used the next time you use R. You can create a new script by going to File>New File>R Script.

Exercise 2

1. Open an R script
2. Type `55/9` in the script and click the run button
3. Put the cursor back on the line where you typed 55/9 and press Ctrl+Enter
4. Type `a <- 156` and run the line of code.
5. On a new line type `b` and then press Alt+- (Alt and the minus key) and see what happens in the script.
6. Assign `b` to a value of value of your choosing and run the line of code.
7. Type `# This is a comment` in the script. R will ignore anything after a number/hash key.
8. Press and hold Alt on the line with the comment and press the up or down arrow keys.

Getting help

There are a few ways to get help from within R. If you know the name of the command then you type `help('command')` or `?command`, alternatively if you're in RStudio you can place your cursor on the command and press F1. If you don't know the name of the command then searching online for the subject or task you want with R will typically return a lot of results.

Working directories

When you open R or RStudio, it is only able view one folder on your computer at a time, called your working directory. This can lead to issues when you tell the computer to look in the wrong place for a file. Often the computer will respond by saying the file does not exist, and one way to solve this issue is to set the working directory to the folder you want R to view. [4]

A common problem when using working directories is that they may change, or if you're collaborating with others, then your computer structure may be different. For instance, you may be working on a C drive and your colleague on a H drive.

[4] This can be achieved by using the `setwd()` command. You can see your current working directory by using the `getwd()` command.We won't focus on using these command in this course, but one thing to note is that the file paths require backslashes \ and won't work with forward slashes / in R

RStudio Projects

To alleviate the issues of working directories, RStudio have created projects. A project is a file which sits inside a folder. When you open the project file, it connect all the folders and documents within the folder and automatically sets the working directory for the specific project. The main benefit of using projects is that multiple people can work on the project without having to retype directories. It is useful to use projects even if you do not collaborate as they can be used to distinguish between different analyses you are conducting.

Exercise 3

1. Download the course data from the website if you haven't already and extract the folder onto your computer.
2. Click the drop down menu in the top right hand corner
3. Select new project
4. Select existing directory
5. Navigate to the course folder
6. Click create project
7. Close and Reopen RStudio
8. Reload the project from the menu in the top right hand corner.

Installing and loading packages

To install a package[5] you can click on the package window in the display panel of RStudio and click install. Alternatively to packages can be installed using the command **install.packages** command. This downloads a package onto your computer so it can be used at a later date. When you want to use the package you need to load it. Loading a package is achieved by using the **library()** function.

[5] Like many pieces of software or products, R can be extended beyond its base capabilities. The 'extensions' in R are called packages. R packages provide a systematic way for other users to develop extra functionality for R. Packages can be a collection of functions and they may contain data.

```
install.packages("NameofPackage") # Installs a package for use later
library(NameofPackage)  # Loads the package to be used in this session
```

There are a number of sources for packages and one of the largest is CRAN (Comprehensive R Archive Network) which has over 12,000 packages.[6]

[6] There are also CRAN Task Views which provide information on different packages by subject area. For example the Spatial Task View provides information on different types of spatial statistics packages that exist.

Data frames

A data frame is used for storing data tables. It is a two dimensional collection of objects where each column can be a different type.

```
## Warning: package 'gapminder' was built under R version 4.4.2
```

Table 1: Example data frame

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.801	8425333	779.4453
Afghanistan	Asia	1957	30.332	9240934	820.8530
Afghanistan	Asia	1962	31.997	10267083	853.1007

R has a range of data types[7]. For this course we will only use a few of them.

[7] A full list can be seen in the R documentation

- Character: Text such as "Hello"
- Numeric: A number such as 5 or 1.34
- Logical: TRUE or FALSE
- Factor: Numbers that have text labels. Such as 0 = "Male", 1 = "Female"
- Vectors: A one dimensional collection of objects all of the same type such as (TRUE, FALSE, TRUE, TRUE)

Loading data frames from a package

Some packages contain datasets. To load data from a package, first you must load the package using the `library()` command and then you use the `data()` command.

```
library(gapminder)
data("gapminder")
```

You can type `data()` into the console to get a list of datasets that are available for the packages you currently have loaded. These datasets can be really useful for creating a reproducible example when asking for help for someone else, especially when your own data is confidential and shouldn't be shared.

Working with data frame

In R you can manipulate objects by using functions. Functions are commands that take an input and give an output[8]. We can do some basic exploring of the data using functions that take a data frame as the input and output a display of information about the data frame.

[8] Although some functions do not require an input and may not display any output that you can see.

```
head(gapminder) # See the top 6 rows
tail(gapminder) # See the bottom 6 rows
names(gapminder) # See the variables names
summary(gapminder) # Summarise the data
str(gapminder) # See the structure of the object
class(gapminder) # See the class (In this case a data.frame)
```

Exercise 4

1. Load the `gapminder` package
2. Load the `gapminder` data frame using `data("gapminder")`
3. Look at the top 6 and bottom 6 rows
4. Look at the structure of the data frame
5. Look at the names of the columns (variables) of the `gapminder` data

The Tidyverse

R has been around since the early 1990's and since then many developments have occurred within the language. A recent major development is the invention of the tidyverse. The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Tidy data

The tidyverse packages aim to help create and work with tidy data. Tidy data is a format for storing your data sets that make it easier to analyse. Tidy data will have the following properties:

- Each variable forms a column.
- Each observation forms a row.
- Each value has it's own cell.

Two of the main packages in the tidyverse for working with tidy data are `ggplot2` and `dplyr`. First we will look at `ggplot2` to create graphs and visualisations. After that we will look at how to manipulate tidy data and how to read different types of data into R.

ggplot

R is well known for its data visualisation capabilities. One of the main reasons it has become so popular for data science and analysis is the package `ggplot2`.[9] `ggplot2` allows graphs to be made based on a grammar of graphics.

[9] R also has a really good plotting system as part of its standard installation often referred to as base plot. Base plot is definitely something you should check out at a later date but in this course we will use `ggplot2`.

The grammar of graphics allows individuals to move beyond describing named graphs such as a scatter plot or bar chart to describing the components of a graphic. One of the main parts of the grammar of graphics is the geometric building blocks which are referred to as `geoms` in `ggplot2`. The `geoms` describe the shapes that make up a plot. For example, a scatter plot is just a set of points, a time series graph consists of a line, and a histogram consists of a group of rectangular bars that touch the x axis. These `geoms` can be used individually or combined together to create more complex charts.

There are many `geoms` in ggplot and three common ones which we will make use of today are `geom_point()`, `geom_bar()`, `geom_line()`. `geom_point()` represent points, `geom_bar()` represents bars, and `geom_line()` represent lines.

A good place when starting a new graph is The R Graph Gallery https://www.r-graph-gallery.com/. The website gives a number of different types of graphs and provides the source code. Unfortunately, we won't be able to spend much time looking at ggplot2, as we'll need to move on to spatial analyses soon. We will spend a bit of time creating a few graphs, discussing some basics, and looking at a really common error that can be made when making the graphs. For a more thorough introduction to `ggplot2` I would recommend the book R for Data Science https://r4ds.hadley.nz/ which can be accessed for free online.

Making a graph

To understand how to use `ggplot2` we are going to create some graphs and then go over the syntax. First we'll load and subset the gapminder data. Then we'll create a scatter plot of life expectancy against gdp using `geom_point()`. In the scatter plot, we will colour the points according to continent. To create a bar chart, we can change the code and use `geom_bar()` instead of `geom_point()`. We'll use `geom_bar()` to show the number of countries in each continent.

The syntax of ggplot involves first defining an empty plotting object with the `ggplot()` command, then geometries are added to create a physical plot.

```
# Load the gapminder data
library(gapminder)
library(dplyr)
library(ggplot2)

data("gapminder")
gm2007 <- filter(gapminder, year == 2007)

# Scatter plot
ggplot(gm2007) + # Define an empty plot with gm2007
  geom_point(mapping = aes(x = lifeExp, y = gdpPercap,
                           col = continent)) # Create a scatterplot
# Bar chart
ggplot(gm2007) +
  geom_bar(mapping = aes(x = continent)) # Create a bar chart
```

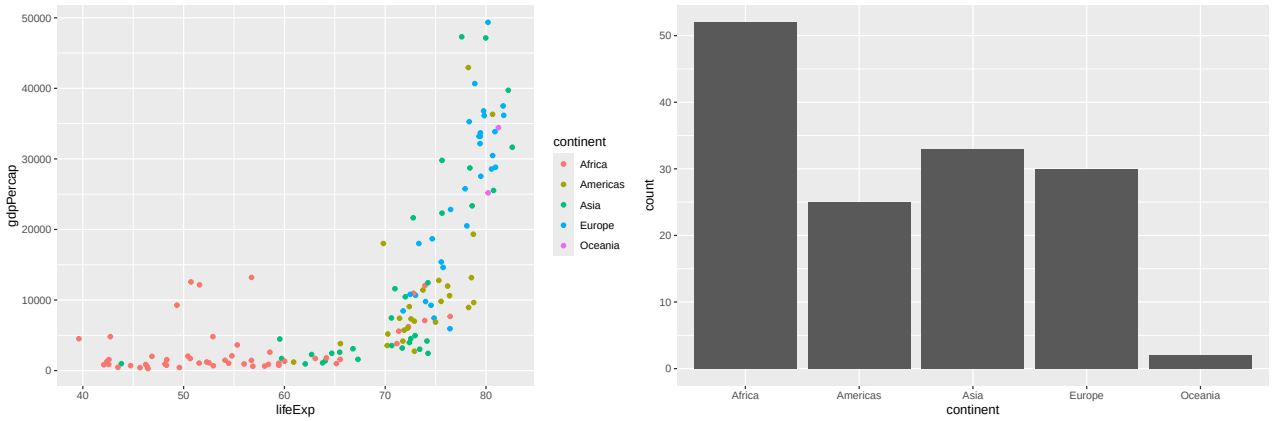


Figure 1: Scatter and bar chart in ggplot2

aes - aesthetics

The function `aes()` stand for aesthetics. This function is where you can tell `ggplot2` which variables you want to plot, and you can also define the size or colour of the `geom` based on another variable.

Sometime you want to change the colour but not based on a variable. The `aes()` function of ggplot is for things that change according to the data. For anything that is fixed it should go outside of aes. If you try to assign an element to a variable outside of aes it will fail. The correct syntax here is to put colour outside of aes.

```
# Create gapminder data for one year
ggplot(gm2007) +
  geom_point(aes(lifeExp,gdpPercap),  col = "blue")
```

If you try to put something fixed in the aes such as the colour blue. It will act strangely

```
# Create gapminder data for one year
ggplot(gm2007) +
  geom_point(aes(lifeExp,gdpPercap, col = "blue")) +
  scale_y_log10() +
  ggtitle("Add a title")
```

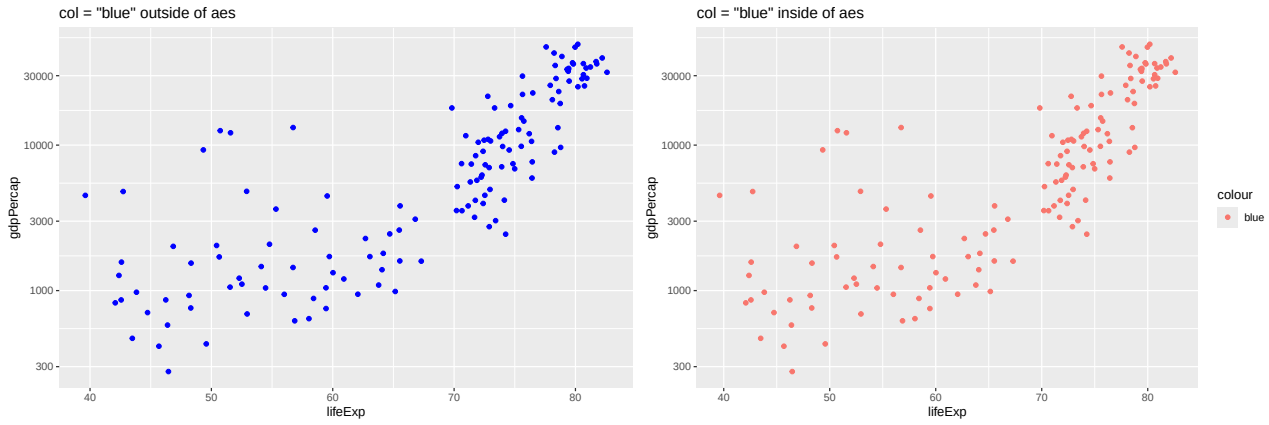


Figure 2: Common colour mistake in ggplot2

```
# Create gapminder data for one year
ggplot(gm2007) +
  geom_point(aes(lifeExp,gdpPercap), col = continent) +
  ggtitle("Text instead of variable")
```

```
## Error in eval(expr, envir, enclos): object 'continent' not found
```

Saving graphs and themes

Plots can be save using `ggsave()`.

```
g1 <- ggplot(gm2007) +
        geom_point(mapping = aes(x = lifeExp, y = gdpPercap,
                                 col = continent))

ggsave("Outputs/scatterplot.png", plot = g1)
```

There are also themes which can be added to the plots which allow quick customisation of a plot.[10]

[10] There are a number of extra packages for themes such as `ggthemes`

```
g1
g1 + theme_cleveland()
g1 + theme_bw()
g1 + theme_dark()
```

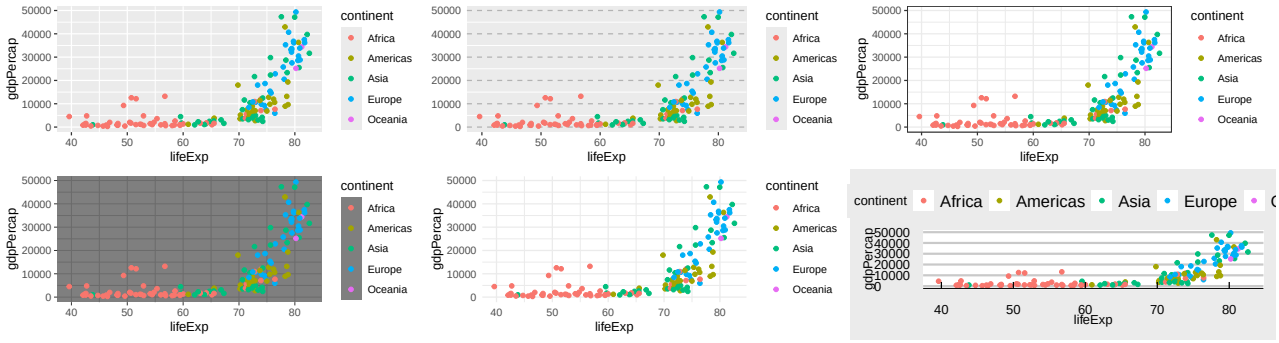


Figure 3: Themes in ggplot2

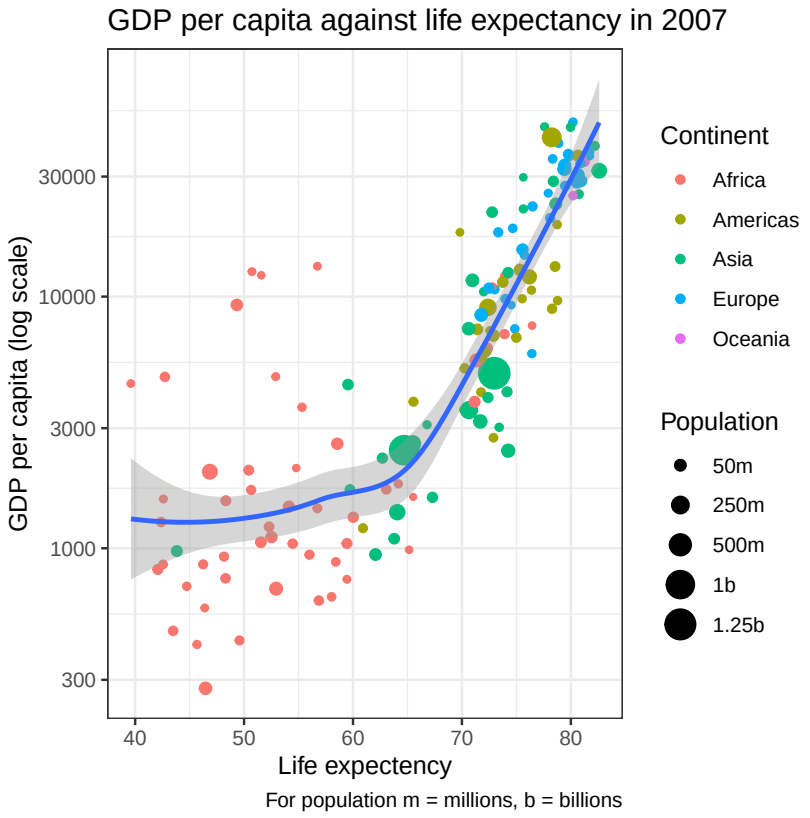
Exercise 5

1. Load the `gapminder` data.
2. Create an empty plot with the `ggplot()` function.
3. Add a line plot using `geom_line()`.
4. Inside `geom_line()` put `aes(x = year, y = lifeExp, group = country)`.
5. Run the command to produce a line graph of Life expectency for the different countries.
6. After the `aes()` type `alpha = 0.2` this reduces the transparency of the colour.
7. Add `col = "white"` to the geom_line (Inside `aes()` or outside?).
8. Add `theme_dark()` to the end of the plot to change the overall style.

Further customisation

There are many different options for customising your graphs. For example, the legend and axis can be changed and with practice you can create publication ready graphs.

```
# Create gapminder data for one year
ggplot(gm2007) + # Define the data
  geom_point(aes(lifeExp,gdpPercap, col = continent, size = pop)) + # Create scatterplot
  geom_smooth(aes(lifeExp,gdpPercap)) + # Add a smoothed regression line
  scale_y_log10() + # Scale y axis by log
  ggtitle("GDP per capita against life expectancy in 2007") + # Add title
  ylab("GDP per capita (log scale)") + # Label y axis
  xlab("Life expectency") + # Label x axis
  scale_size_continuous(breaks=c(5e7, 2.5e8,5e8, 1e9,1.25e9), # Change and label the scales for size
                        labels=c("50m", "250m","500m", "1b", "1.25b")) +
  guides(col = guide_legend(title = "Continent"),
         size = guide_legend(title = "Population")) + # Edit the title for the legends
  labs(caption = "For population m = millions, b = billions") + # Add a caption
  theme_bw() # Change to a simplified theme
```



```
library(gapminder)
data("gapminder")

# Basic line plot
ggplot(gapminder) +
  geom_line(aes(x = year, y = lifeExp, group = country),
            alpha = 0.2)
```

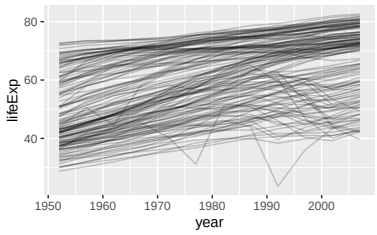


Figure 4: Basic plot

```
# Change the style
ggplot(gapminder) +
  geom_line(aes(x = year, y = lifeExp, group = country),
            alpha = 0.2, col = "white") +
  theme_dark()
```

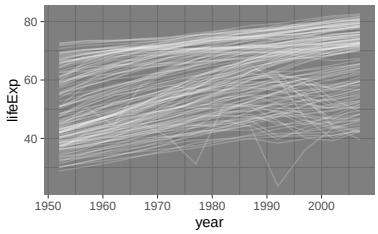
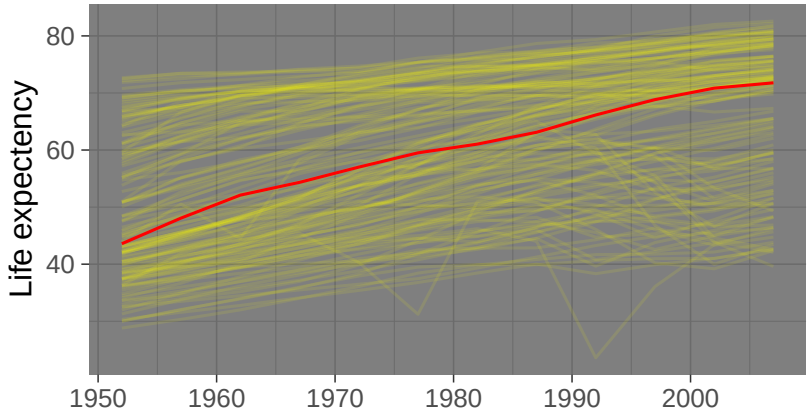


Figure 5: Plot with theme dark

With a bit of extra practice it become easy to do things like highlight specific countries.

```
gm_tky <- filter(gapminder, country == "Turkey")

ggplot(gapminder) +
  geom_line(aes(x = year, y = lifeExp, group = country),
            alpha = 0.1, col = "yellow") +
  geom_line(data = gm_tky, aes(x = year, y = lifeExp),
            col = "red") + # Add a line with a new data frame
  ylab("Life expectency") +
  xlab("") +
  labs(caption = "The rise in life expectancy of Turkey from 1950 to 2010") +
  theme_dark()
```



The rise in life expectancy of Turkey from 1950 to 2010

dplyr

`dplyr` is an R package that provides a grammar of data manipulation. The package is part of the tidyverse and it simplifies the process of common data manipulation tasks. Using `dplyr` requires learning the functions that it provides. In keeping with the package being a grammar, these functions are sometimes called verbs. The verbs/functions we will look at now are:

- `select()` picks variables based on their names.
- `filter()` picks rows based on their values.filter to get certain rows
- `arrange()` changes the ordering of the rows.
- `mutate()` adds new variables that are functions of existing variables.

The syntax for these verbs are nearly always they same. They require the name of the data frame we wish to manipulate, and the subsequent arguments relate to the variables we wish to work with.

For instance, if we want to select the country variable from the `gapminder` data frame, we type

```
select(gapminder, country)
```

If we want to filter the `gapminder` data frame to be only observations from the year 1952 then we use `filter()`

```
filter(gapminder, year == 1952) # Select only from 1952
filter(gapminder, year == 1952 & continent == "Africa") # African countries from 1952
# Either 1952 or countries from Africa at anyyear
filter(gapminder, year == 1952 | continent == "Africa")
```

Table 2: Logical operators in R

Operators	Description
==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
=<	less than or equal to
x & y	x and y
x | y	x or y
!x	not x
x %in% y	x is in y

To sort the data by year we can use the arrange command

```
arrange(gapminder, year) # default is increasing
arrange(gapminder, desc(year)) # decreasing
```

The `mutate()` function is used to create new variables. We `mutate` the data frame by adding extra variables.

```
mutate(gapminder, longlifeexp = lifeExp>80)
```

Exercise 6

1. Select the country and continent column from the `gapminder` data frame
2. Filter the `gapminder` data to get years after 1980
3. Assign the data from step 2 to a new object called `gapminder_after_1980`
4. Sort `gapminder_after_1980` by lifeExp

Grouping data and calculating summaries of the groups

Sometimes data will need to be aggregated for analysis.

- `group_by()` groups the data by one or more variables
- `summarise()` summarise the data based on groupings

The `group_by()` verb can be used to group data. Unlike the previous verbs where they were used on their own, `group_by()` only becomes useful when combined with other verbs like `summarise()`. `summarise()` can be used to calculate the aggregate values for the group.[11]

[11] The `n()` function counts the number of rows in each category. It is a helper function within the package `dplyr`.

```
gapminderworld <- group_by(gapminder, year)
summarise(gapminderworld, meanLifeExp = mean(lifeExp), countries = n())
```

Table 3: Useful functions for summarise

Operators	Description
n()	count rows in each group
min()	minimum value
mean()	mean
sd()	standard deviation
max()	maximum value
first()	first value in each group
last()	last value in each group

Piping or chaining together verbs!

The verbs in `dplyr` allow for a wide range of manipulations that are required in data analysis. [12] As we have seen the verbs can be used one at a time which creates a new dataframe each time. Now we will look at how the verbs can be put together to create sentences using an object called a pipe.

[12] There are more verbs and they can be found by searching for `dplyr` or visiting `http://dplyr.tidyverse.org/`.

A pipe place the object on the current line as the first argument in the next functions. This can sound a bit confusing at first but we'll go through an example to show what this means.

In R a pipe is represented by the symbol %>%. This symbol can be created in RStudio using ctrl+shift+m. With the vocabulary of `dplyr` and more generally the %>% can be read as saying "and then."[13]

[13] This code can be read as "Take the dataframe gapminder, and then `group` the data by year, and then, `summarise` the groups by calculating the mean life expectency and count the number of countries in each group"

```
# Piping
gapminder %>%
  group_by(year) %>%
  summarise(meanLifeExp = mean(lifeExp), countries = n())

# Without piping
summarise(group_by(gapminder, year), meanLifeExp = mean(lifeExp), countries = n())
```

Exercise 7

1. Type `gapminder` then create a pipe by pressing ctrl+shirt+m
2. Press enter to go onto a new line
3. Group the data by year and continent then add a pipe and go to new line
4. Summarise by calculating the mean population
5. Run the entire command by highlighting it all and pressing run or ctrl+enter
6. Create your own piped command using `dplyr` verbs.

The pipe operator allows even more complex data manipulations to be performed in one function call.[14]

```
gapminder %>%
  select(year, continent, lifeExp) %>%
  filter(continent== "Asia") %>%
  group_by(year) %>%
  summarise(meanLifeExp = mean(lifeExp), countries = n())
```

[14] This code can be read as "Take the dataframe gapminder, and then `select` the variables year, continent, and lifeExp, and then `filter` the row where the variable continent equals Asia, and then `group by` by year, and then `summarise` by calculating the mean life expectency and count the number of countries in each group"

Now you hopefully have some idea on how you can manipulate some data. We will now consider how to visualise data. This will become very useful when we start looking at the spatial data in future sessions.

readr - reading data into r

First we will look at how to load data into R. A common format for data is a comma seperated value (csv) file. A csv file uses comma's to record each element in a dataset.

We can load a csv file using the read_csv function from the readr package. R can be used to read in many different types of data. There are also many ways of reading in the same data format into R. For now we will focus on reading in a csv[15] file.

```
library(readr)
# Read in a csv file
dataname <- read_csv('PathtoData/data.csv')
# Write a csv file
write_csv(name_of_object, path = 'PathtoData/data.csv')
```

[15] csv stand for comma seperated value, you may associate these files with excel because on windows computers excel will usually be used to display the file. However, they are more general than that and represent a structure for storing data where each element is seperated by a comma. Other similar forms are used such as tab seperated value which is abbreviated as tsv

rmarkdown

`rmarkdown` is a document format that allows code and text to be combined together. The `rmarkdown` document can be used to create reports and notebooks without having to copy and paste graphs or results to other documents. `rmarkdown` is no harder to use than what we have done so far. We will demonstrate how to start an `rmarkdown` file using the scripts we've used in the session so far.