Section 2

Your Name

13/12/2021

Contents

---

1. Introduction

Reproducible:

> A result is reproducible when the same analysis steps performed on the same dataset consistently produces the same answer.

---

2. RMarkdown

Have you ever used the internet?
The internet uses a language called HTML (Hypertext Markup Language). RMarkdown uses a similar language to link data, code, methods results, graphs and charts to produce an integrated formatted document. This tutorial was created in RMarkdown.

Have you ever completed a project and then realised there is an error in the data?
This probably took some time to fix if the graphs were created in Excel, the analysis in statistical software and the written text in Word. In RMarkdown you can create everything in the RMarkdown file. If you change the data and then run the RMarkdown file all the changes are done for you.

So how does this help reproducibility?
In a final Word document, publication or report where only the results are included, there is no visibility of the data, code or method. In order for research to be reproducible, to be able to check the accuracy and run it for our-self, we need to be able to understand how the result was formulated. In the RMarkdown file the names of the data sets, the manipulation of the data, the code and how the results are produced

are all documented in the RMarkdown file. If the data sets are also shared on a shared repository, such as GitHub, you would even be able to create the entire report yourself by simply running the Rmarkdown file. RMarkdown also doesn't change the original data, but uses a copy of the data so the orignal data source can still be shared.

Now that you know how valuable RMarkdown is for reproducibility, we will work through some basic formatting features in RMarkdown.

First, install RMarkdown and open an RMarkdown file.

Instructions on how to install RMarkdown can be found here.

File/New File/RMarkdown

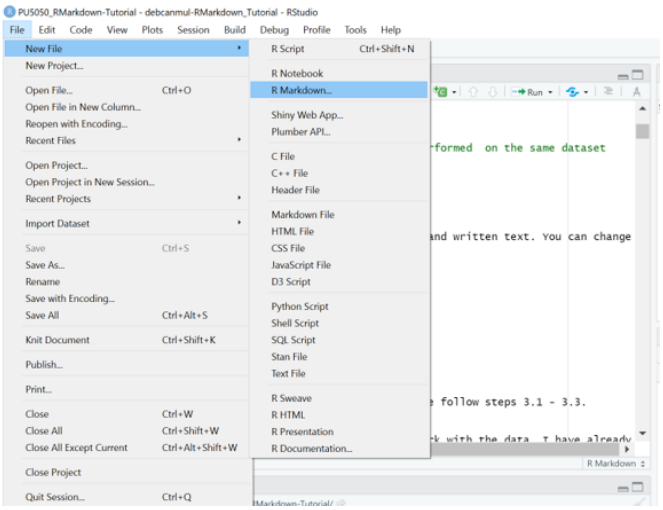


Figure 1: New RMarkdown File

We will work through each of these RMarkdown topics in turn:

- 2.1 RMarkdown File Header
- 2.2 Text Chunks
  - 2.2.1 Headers
  - 2.2.2 Lists
- 2.3 Code Chunks
- 2.4 Inline Code
- 2.5 Output File Type Specification
- 2.6 Generation of Output

2.1 RMarkdown File Header

What is an RMarkdown File Header?

A basic RMarkdown header is created by default when you create an RMarkdown file. It consists of the following text:

This header is called a YAML header. YAML stands for "YAML Ain't Markup Language". Yes, the terminology is rather odd. It consists of a title, author, date and output. You can change the default header by simply editing it in RMarkdown. However make sure it is in the same layout, indentation will change the output of the header. Also start and end on a On a separate line with —.

The YAML header can be changed to change the output of the document. In the basic header it is set to html_document. It can be changed to a pdf_document. We will talk about this in more detail in section 2.5. The font and font size for the entire document can also be changed by editing the YAML header. A contents page and a bibliography are also possible options.

2.2 Text Chunks

Text chunks are sections where you freely type text. This is anywhere in the RMarkdown file below the YAML header, where you haven't added a code chunk, see section (2.3) for more details on code chunks. You can edit the text by using different syntax commands. We will look at headers and then lists.

2.2.1 Headers

To create a header in RMarkdown in a text chunk you have to start with a # and then a space and then type the name of the header. The biggest header will start with a single #, ## would create a slightly smaller header and so on. I used ### for my first header, because if I had used a single # it would have automatically included this header as part of my contents at the start of the report.

```
### Large Header
##### Slightly smaller header
####### Even smaller header
```

The output would be as shown below:

Large Header

Slightly smaller header

Even smaller header

2.2.2 Lists

To create a list you can either use bullet points or a numbered list. To create bullet points start with * then a space, in front of the name of item in the list.

```
* Aberdeen
* Dundee
* Edinburgh
* Glasgow
* Inverness
```

The output would be as shown below:

- Aberdeen

- Dundee
- Edinburgh
- Glasgow
- Inverness

To create a numbered list you start with numbers in the front of the item in the list. Even if you don't number the items correctly from 1-5, if you number the list in any order RMarkdown will renumber them for you in ascending order starting from the top of the list.

1. Aberdeen
8. Dundee
3. Edinburgh
6. Glasgow
5. Inverness

The output would be as shown below:

1. Aberdeen
2. Dundee
3. Edinburgh
4. Glasgow
5. Inverness

To create a sublist in RMarkdown, you need to indent the sublist and use + in front of each item in the sublist.

1. Aberdeen
 + Aberdeen Art Gallery
 + Duthie Park
 + Aberdeen Maritime Museum

The output would be as shown below:

1. Aberdeen
    - Aberdeen Art Gallery
    - Duthie Park
    - Aberdeen Maritime Museum

2.3 Code Chunks

Code chunks are where you type all the code for your project. You can decide if you want all the code visible in your final report or if you want it hidden. You can also switch off error messages and decide if the output from the code chunk is to appear in your report or if it's just to be used within another part of your code in a different code chunk. The quickest and easiest way to do this is use the code chunk cog.

Code chunks start with three back ticks, {r} and end with 3 back ticks "' on a separate line. You can use the shortcut **CTRL** + **ALT** + **I** to create a code chunk. The + means "and" and not the plus sign on the keyboard.

"'{r}

Your code goes in here...

```

Code chunks in the RMarkdown file are highlighted in grey.

```
3+7
```

```
## [1] 10
```

I set the code chunk cog to have the settings:

Output: Show code and output

You can add comments in code chunks by adding # to the front of the comment. This will be used in the next section to show you how to add inline code.

2.4 Inline Code

You may wish to add a value calculated from your data somewhere within a sentence in your text. In this case code chunks would not be suitable, but you can use inline code. Instead of 3 back ticks you use only 1 before and afer r. You can use different syntax within the inline code. In this example I have used + to add 3 people and 7 people who are waiting. I could have used multiply *, divide / or even a result that has been saved to an object. We will talk more about objects in section

```
#The total number of people in the waiting room is `r 3+7`
```

The total number of people in the waiting room is 10.

2.5 Output File Type Specification

Output file types were mentioned briefly in section 2.1, YAML header. We can change the output file type in the header from html_document to pdf_document. You can also change the file type to word documents (word_document), markdown files (md_document) and even different types of slides for presentations. If you want to take it even a step further you can create interactive documents. Just a word of warning some changes to the YAML header might not turn out the way you want them in different output file types.

2.6 Generation of Output

To create the chosen file type you have to Knit the data, code and text into the final document. If you want to create pdf documents you will have to install a package called TinyTex.

---

3. Data Preparation

What is a fake dataset?
The data sets that were created for this tutorial are called fake datasets. They were created using code which generated random data specifically for this tutorial. If the data isn't real, why are fake datasets useful? The type of data for our scenario is health data. Health data cannot be openly shared due to confidentiality. If you were asked to find the answer to a similar question but with real data, you would have gained the knowledge and experience by practicing with the fake data. Not only that, if I had shared the method with you, the code in the RMarkdown file, then very quickly with little manipulation you would have an answer.
```

You may also look at my code and think of improvements or find errors. This could all be fixed and edited before the real data was analysed. However, there are some limitations to fake data. They can't be used for analysis that requires a specific statistical test. The data was just randomly generated. There is a possible solution however, synthetic datasets.

Synthetic data sets are a special type of fake data set that are created to keep the statistical properties. Synthetic datasets can be time consuming to create. If you are also doing research on a rare disease in a small population, it is likely the patient could still be identified. So great care must be taken, but the benefits of fake datasets and synthetic data sets are essential for the reproducibility of health data.

What is metadata?

Metadata is different forms of documentation that explains the data in detail. If the project is to be truly reproducible, detailed explanation must accompany the data. Some examples of metadata are data dictionaries, codebooks and ReadMe files. Other examples of metadata can be found here. The type of project will determine the type and amount of metadata that will be required.

Why is good metadata important?

If I was to give you a large dataset with no headings or any explanation of how the data was created, it would be useless. The better the metadata the more reproducible the overall project can be and the quicker the project can be reproduced. There are different standards for metadata that can be followed depending on which field of work or research you are interested in. Information on different standards can be found here.

The ReadMe file in this RProject was modified from Cornell, a copy of the template can be found here.

What is a tidy dataset?

A dataset is classed as tidy when the following 3 rules are obeyed:

1. **Each variable has it's own column**
2. **Each record/observation has it's own row**
3. **Each individual cell has a single value**

What are the benefits of tidy datasets?

Tidy datasets allow the use of tools that have been created by other people. The tools are designed to be used with tidy data. This saves time by not having to design your own tools. Reproducibility is also increased because other people working with your data will also have access to the same tools.It will also be easier for other people to understand the structure of your data if it is tidy. This would reduce the amount of time people would need to manipulate the data into a tidy form. You can design a workflow using many of these tools. The benefit of this is if you add some new data in the tidy format you can simply rerun the workflow.

Before we can start to prepare data we need to complete steps 3.1 - 3.3, install the packages, read the packages into R and read in the datasets.

3.1 Install the packages

First you want to install the packages you will need to work with the data. I have already installed the packages into R, so for you, you would complete the following.

- Start a new markdown file (2.2.1)
- Start a new code chunk (2.2.3)

Then type in the following code.

install.packages("tidyverse")
install.packages("here")

The package **tidyverse** is a collection of packages that are very useful for data science. We will use two of the packages within tidyverse in this tutorial, **dplyr** and **ggplot2**. The first package, dplyr, is used to transform data. The second package, ggplot2, is used to create visual representations of the data such as graphs and charts.

3.2 Read the packages into R

We will read in the tidyverse package that has been installed. The library function reads in the package. The message from this code chunk has been shown below so you can see which packages have been loaded.

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.1.2
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.4     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.2     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

The here function is used to create simple file referencing in projects. It sets the top level directory (folder) as the main reference point. Then any files are located from this top level. This is instead of having a very large file path consisting of personal directories.

```
library(here)
```

3.3 Read in the data sets

To be able to read in a dataset we need to assign it to an **object**. An object can be many different things such as a single number, a single word or a large dataset. You create your own names for objects but just be careful you don't use a function that already exists, a common object name is data but there is a function called data. Also don't start an object name with reserved characters, numbers or include any spaces within the name. When an object has been created you will the name of the object in the environment pane on the top right of the RStudio window. Also if you use the same object name it will override the data previously assigned. To assign an object we use the assignment operator, <-, a less than sign and then a dash.

R-Software is also case sensitive, so if you name an object with all lower case letters, R will only recognise the name in this form.

We have named an object demographics. The used the read_csv function to read in the 2021-12-02_PU5050_Demographics.csv file. We used the here function so that the file directory is located in relation to the top directory.

```
demographics<-read_csv(here("2021-12-02_PU5050_Input/2021-12-02_PU5050_Demographics.csv"))

# Now read in the ImagingWaitTimes
ImagingWaitTimes<-read_csv(here("2021-12-02_PU5050_Input/2021-12-02_PU5050_ImagingWaitTimes.csv"))
```

3.4 Inspect the data

Now that we have the data loaded into R and visible in the environment pane, using the object names demographics and ImagingWaitTimes, we can start to inspect the data. Let's look at demographics first...

We will use a function called glimpse to look at the variable names within the data set.

```
glimpse(demographics)
```

```
## Rows: 100
## Columns: 4
## $ chi  <dbl> 2865914370, 7429183650, 2041536789, 6705438219, 9598617423, 47351~
## $ dob  <chr> "04/06/1950", "14/08/1950", "16/03/1951", "11/07/1951", "26/10/19~
## $ sex  <chr> "female", "male", NA, "male", "female", "male", "female", "male",~
## $ city <chr> "Inverness", "Edinburgh", "Dundee", NA, "Glasgow", "Glasgow", "Gl~
```

We can see that the data has 4 columns: chi, dob, sex and city. Each column has a single variable, each row is for one observation and each cell has one value. Our data is therefore tidy data.

Variables come in difference types, the variable chi is a numeric variable. It's classed as a double , which really just means it has lots of accuracy. The other three variables are classed as characters , which is a categorical type of variable. Note however the dob variable is really a date. We will talk about this later in the tutorial.

There are many functions in R that can be used to do similar tasks. Another function to look at the data is the function, head.

```
head(demographics)
```

```
## # A tibble: 6 x 4
##          chi dob        sex    city
##        <dbl> <chr>      <chr>  <chr>
## 1 2865914370 04/06/1950 female Inverness
## 2 7429183650 14/08/1950 male   Edinburgh
## 3 2041536789 16/03/1951 <NA>   Dundee
## 4 6705438219 11/07/1951 male   <NA>
## 5 9598617423 26/10/1951 female Glasgow
## 6 4735162809 18/03/1954 male   Glasgow
```

Again we can see there are 4 groups and the types of data.

Notice however there is the word tibble on the top left of the output. If you look back in section 3.2 tibble was a package loaded within the tidyverse package. A tibble is a data frame but a more modern version.

Now that we have an understanding of the variable names and the type of variables, we need to look for missing values, NA, and any unusual data values.

I'm going to show you first of all how to find the number of missing values in a single column. Let's introduce two functions:

- **filter** - you can choose to include or remove data with specific criteria. In this case we want to find which cells have missing values, NA.
- **summarise** - you use the function to summarise any variable in the table. This might be the mean, standard deviation or simply to count how many values within that variable.

For each of the functions you can type in a code chunk the name of the function starting with ? and run the code chunk. In the bottom right pane in RStudio you will see additional information about the function.

I also want to introduce the pipe operator, %>%. It looks more complicated than it is, it just means "and then". I will show you in the code below. If you like shortcuts the shortcut for the pipe operator is **CTRL + SHIFT + M**

```
# We will first look at the demographics dataset....and then (using the pipe operator)
demographics%>%

# We will then filter to find the missing values in the dob variable....and then (using the pipe operat
  filter(is.na(dob))%>%

# The summarise function will count how many missing values in each column (variable).

summarise(n())
```

```
## # A tibble: 1 x 1
##   `n()`
##   <int>
## 1     2
```

This shows that within the column dob there are 2 missing values. We want to remove these and save the results to a new object. Let's call the filtered data filtered_dob.

```
# The changes to demographics will be assigned to a new object called filtered_dob
filtered_dob<-demographics%>%

# Note the exclamation mark in the code this time. It means is not a missing value.
  filter(!is.na(dob))%>%

# The view function will open the object filtered_dob, the missing values in dob will have been removed
  view()
```

A better way of doing this, especially if you are working with large datasets is to use the summarise function again.

```
filtered_dob%>%
  filter(is.na(dob))%>%
  summarise(n())
```

```
## # A tibble: 1 x 1
##   `n()`
##   <int>
## 1     0
```

We can see now that the missing values have been removed within the new filtered data. Now if we were to continue with this method we would have to work our way along one column at a time. We would check for the number of missing values, save the new filtered data to a new object, remove the missing values and finally check the values have been removed. However, there is a way we can check all the columns at the same time.

```
# We are going to use the demographics dataset again.

demographics%>%

# This is a slightly different version of the function summarise, this time it summarises all the colum

  summarise_all(~sum(is.na(.)))
```

```
## # A tibble: 1 x 4
##     chi   dob   sex  city
##   <int> <int> <int> <int>
## 1     0     2     2     2
```

We have looked at the original demographics file and we can see there are 2 missing values in the columns dob, sex and city. Let's remove them and call the new object demo_filter

```
#We have a new object called demo_filter
demo_filter<-demographics%>%

# We use the function na.omit to remove all the missing values.
  na.omit()%>%
# We can view the dataset demo_filter
view()
```

Let's check that all the missing values have been removed.

```
demo_filter%>%
  summarise_all(~sum(is.na(.)))
```

```
## # A tibble: 1 x 4
##     chi   dob   sex  city
##   <int> <int> <int> <int>
## 1     0     0     0     0
```

We can see that all of the missing values have been removed.

Now we need to look for any unusual values within the data. As we mentioned earlier the dob variable is a character. We will first change this to a date variable.

- **mutate** - will create a new column with values created from variable/s within the original dataset. In this example we mutate the dob variable to be changed to a date.

```
#The new object name is demo_date, using the demo_filter dataset
demo_date<-demo_filter%>%

# The dob variable is changed from a character to a date in the format day, month, year.
  mutate(dob=as.Date(dob,format("%d/%m/%Y")))%>%

view()
```

```
glimpse(demo_date)
```

```
## Rows: 94
## Columns: 4
## $ chi  <dbl> 2865914370, 7429183650, 9598617423, 4735162809, 8742305196, 74925~
## $ dob  <date> 1950-06-04, 1950-08-14, 1951-10-26, 1954-03-18, 1954-04-05, 1954~
## $ sex  <chr> "female", "male", "female", "male", "female", "male", "female", "~
## $ city <chr> "Inverness", "Edinburgh", "Glasgow", "Glasgow", "Glasgow", "Aberd~
```

Now I will find the youngest and oldest person in the demo_date.

```
#Youngest person
demo_date%>%

#Using the function min() (minimum)
  summarise(min(dob))
```

```
## # A tibble: 1 x 1
##   `min(dob)`
##   <date>
## 1 1950-06-04
```

```
#Oldest person
demo_date%>%

#Using the function max() (maximum)
  summarise(max(dob))
```

```
## # A tibble: 1 x 1
##   `max(dob)`
##   <date>
## 1 2050-10-11
```

Now unless the person with the maximum date of birth has traveled back in time, you can see there is an error with this date of birth.

This row will need to be removed from demo_date.

```
# Filtered for dob before today's date
demo_date_filter<-demo_date%>%

#The filter is using to find dates before 2021-12-11
  filter(dob<"2021-12-11")%>%

  view()
```

Our filtered data is now ready to be saved as a csv file

```
# The write_csv function is used to save the demo_date_filter dataset as a csv file
write_csv(demo_date_filter,"2021-12-02_PU5050_Output/2021-12-11_PU5050_Demo_Filtered.csv")
```

We are now going to look at the ImagingWaitTimes data.

```
#Let's use the glimpse function that we used to inspect the demographics data
glimpse(ImagingWaitTimes)
```

```
## Rows: 100
## Columns: 5
## $ chi                        <dbl> 7154809632, 5986237140, 1520976843, 85623409~
## $ month                      <chr> "August", "September", "October", "August", ~
## $ DiagnosticTestType         <chr> "Imaging", "Imaging", "Imaging", "Imaging", ~
## $ DiagnosticTestDescription <chr> "Non-obstetric Ultrasound", "Computer Tomogr~
## $ WaitingDays                <dbl> 63, 60, 53, 35, 56, 44, 60, 31, 63, 62, 35, ~
```

We can see there are 5 columns, with variables, chi, month, DiagnosticTestType, DiagnosticTestDescription and WaitingDays.

The summary function will find the minimum, 1st Quartile, median, mean, 3rd quartile and maximum value for each varible.

```
summary(ImagingWaitTimes)
```

```
##       chi               month           DiagnosticTestType
##  Min.   :1.236e+09   Length:100         Length:100
##  1st Qu.:4.328e+09   Class :character   Class :character
##  Median :6.303e+09   Mode  :character   Mode  :character
##  Mean   :5.840e+09
##  3rd Qu.:7.350e+09
##  Max.   :9.841e+09
##  DiagnosticTestDescription  WaitingDays
##  Length:100                 Min.   :    -50
##  Class :character           1st Qu.:     33
##  Mode  :character           Median :     44
##                             Mean   :  32722
##                             3rd Qu.:     55
##                             Max.   :3267890
```

This function will provide more insight to numeric variables than character variables, but it shows you another way that you can quickly get a feel for the data in your dataset. We are particularly interested in the variable WaitingDays. You can see that the minimum number of days is -50. This is obviously an error and will need to be removed. Notice the maximum value is 3267890. We would hope people are not having to wait 8953.1 years for an imaging appointment! This will also have to be removed.

Let's filter the data so that the waiting days is not less than zero and not greater than 365 days. Then use the summary function to look at the new maximum and minimum values.

```
ImagWaitTimesFilter<-ImagingWaitTimes%>%

# An and sign, &, can be used to combine criteria
  filter(WaitingDays>0 & WaitingDays<365)%>%

  view()
```

Let's check the maximum and minimum values for WaitingDays

```
summary(ImagWaitTimesFilter)
```

```
##       chi                month           DiagnosticTestType
##  Min.   :1.236e+09   Length:98          Length:98
##  1st Qu.:4.284e+09   Class :character   Class :character
##  Median :6.163e+09   Mode  :character   Mode  :character
##  Mean   :5.830e+09
##  3rd Qu.:7.350e+09
##  Max.   :9.841e+09
##  DiagnosticTestDescription  WaitingDays
##  Length:98                  Min.   :24.00
##  Class :character           1st Qu.:33.00
##  Mode  :character           Median :43.50
##                             Mean   :43.98
##                             3rd Qu.:55.00
##                             Max.   :63.00
```

We can see the minimum waiting time is 24 days and the maximum is 63 days. This seems more sensible. Now lets look for missing values.

```
ImagWaitTimesFilter%>%
  summarise_all(~sum(is.na(.)))%>%
   view()
```

There are 2 missing values in month and 2 in DiagnosticTestDescription. Let's remove these from the dataset.

```
# We will give the new object the name WaitTime_Tidy
WaitTime_Tidy<-ImagWaitTimesFilter%>%

  na.omit()%>%
  view()
```

We can check that the missing values have been successfully removed.

```
WaitTime_Tidy%>%
summarise_all(~sum(is.na(.)))
```

```
## # A tibble: 1 x 5
##     chi month DiagnosticTestType DiagnosticTestDescription WaitingDays
##   <int> <int>              <int>                       <int>       <int>
## 1     0     0                  0                           0           0
```

The WaitTime_Tidy data has now been filtered for unusual values and the rows with missing values have been removed. We are now ready to save the sorted data.

```
write_csv(WaitTime_Tidy,"2021-12-02_PU5050_Output/2021-12-12_PU5050_WaitTime_Tidy.csv")
```

We are now ready to join the data sets. There are many ways to join datasets. You have to have a good understanding of the data and how you want it to join. In our scenario we want all of the waiting time details matched with the corresponding chi number from the demo_date_filter data.

```
# The inner_join function will combine all the data in WaitTimeTidy with the relevant demo_date_filter
join_demo_wait <- inner_join(WaitTime_Tidy,demo_date_filter, by = "chi")%>%

view()
```

We will now check that the data has combined correctly.

```
glimpse(join_demo_wait)
```

```
## Rows: 91
## Columns: 8
## $ chi                       <dbl> 7154809632, 5986237140, 1520976843, 85623409~
## $ month                     <chr> "August", "September", "October", "August", ~
## $ DiagnosticTestType        <chr> "Imaging", "Imaging", "Imaging", "Imaging", ~
## $ DiagnosticTestDescription <chr> "Non-obstetric Ultrasound", "Computer Tomogr~
## $ WaitingDays               <dbl> 63, 60, 53, 35, 56, 44, 60, 63, 62, 35, 34, ~
## $ dob                       <date> 1995-05-09, 1958-08-05, 1994-07-14, 1994-01~
## $ sex                       <chr> "female", "female", "female", "male", "femal~
## $ city                      <chr> "Edinburgh", "Aberdeen", "Inverness", "Dunde~
```

We have 8 variables and they have been combined by the chi variable.

---

4. Data Visualisation

We are now ready to answer the question: What is the average number of waiting days, in each Scottish city, in October 2021?

We will create a graph to help answer the question. I am going to introduce you to the main terminology used when creating a graph or chart in ggplot2.

1. **Data** - to be able to plot a graph or chart you need data. Depending on the type of data you have and the question you want to answer will determine the type of graph or chart that you will choose to create. You will do this by choosing the variables to be visualised.
2. **Mapping** - mapping refers to the variables that you are going to use to plot on your graph or chart. In my example, the cities will be the categorical variable on the x-axis and the average number of waiting days a numeric continuous variable will be displayed on the y-axis.
3. **Geoms** - Geoms are geometric objects, such as a point, rectangle, line, connected line or an area. These objects are mapped on the blank ggplot to represent the data. Depending on the type of data you want to visualise will determine with geom you use. In the data I want to visualise I have a numeric continuous variable (average number of days) and a categorical variable (city). Therefore I have chosen a bar graph to display the data, so the geom is a rectangle.
4. **Themes** - are not a main parameter, but the overall style of the graph can be changed using different themes available within the package ggplot2.

Below is the code to produce a simple bar chart. I start off with my blank piece of paper, ggplot().

ggplot()+
geom_col(aes(city,mean))

I chose the geom to be a rectangle in the form of a column bar chart, geom_col(). I then set the mapping using the **aesthetics function aes()** to choose the variables that I want mapped on the chart using the rectangles. The x-variable will be the city and the y-variable the mean. The mean variable will be created in the code below.

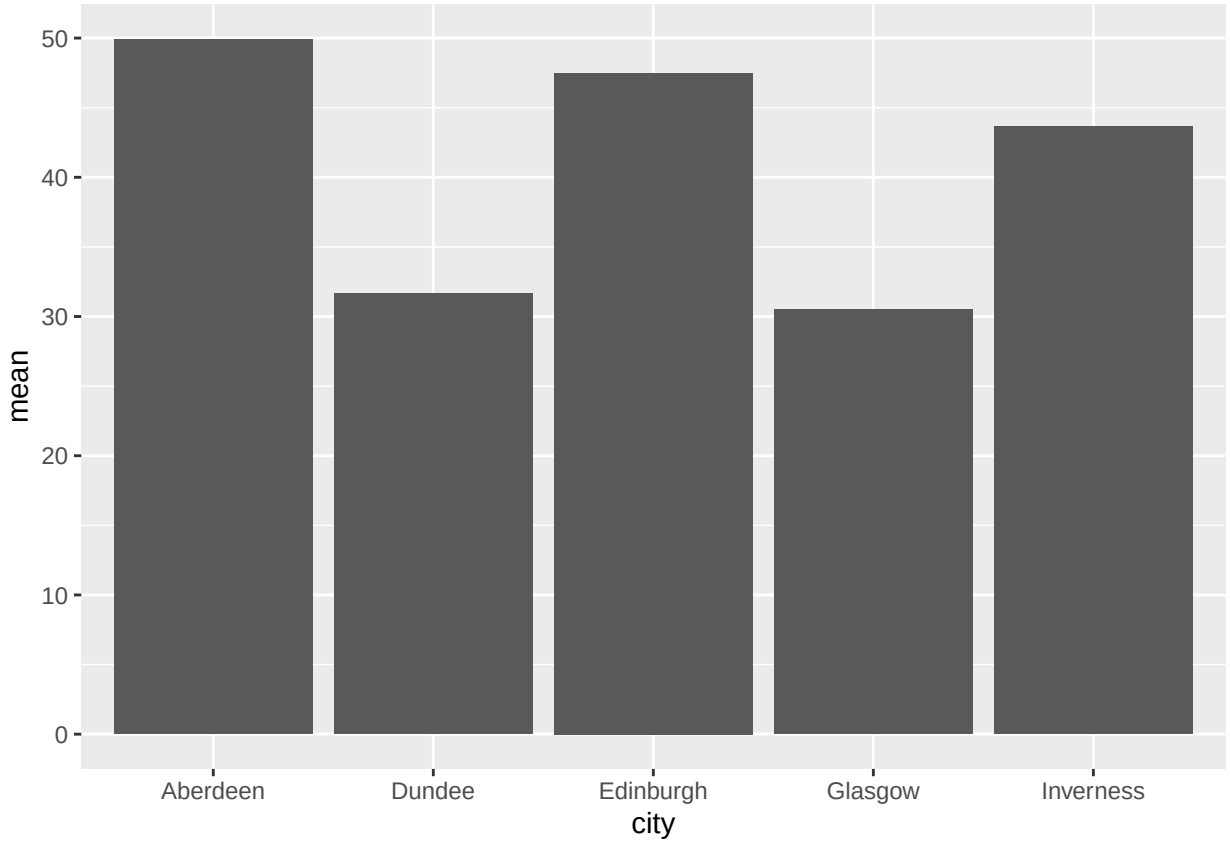
The first thing to do is to decide what data it is we need to answer the question. We need the variables month, DiagnosticTestDescription, WaitingDays and City.

Instead of manipulating one part of the data at a time we will now make more use of the pipe operator and have multiple lines of code in one code chunk. Before we do this I will explain each of the functions that we will use to prepare the data to create the graph.

- **select** - this function allows me to select which variables (columns) of data from a dataset that I want to analyse. If you add a minus sign in front of the variable it will keep all the variaables in the original dataset and only remove this variable.
- **group_by** - if you have a variable with different values, in this scenario the city variable has different variables. I can group by the different cities.
- **unique** - selects only the unique rows within a dataset.
- **ggplot** - gives us a blank canvas, I like to think of this as finding a blank piece of paper where we can draw our graph or chart.

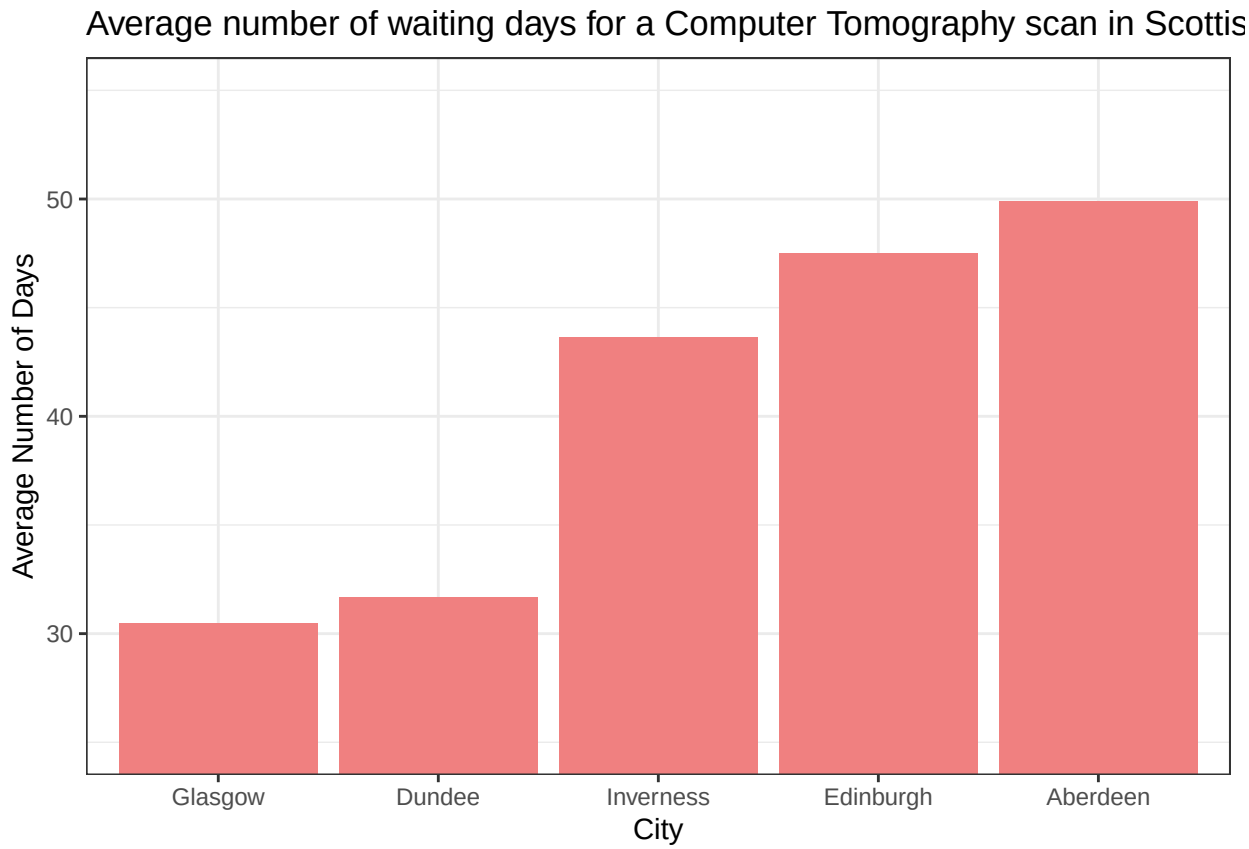
Remember for each of the functions you can type in a code chunk the name of the function starting with ? and run the code chunk.

```
# We will use the join_demo_wait dataset to answer the question.
join_demo_wait%>%

# The variables month, DiagnosticTestDescriptions, WaitingDays and city have been chosen.
  select(month,DiagnosticTestDescription,WaitingDays,city)%>%

# The DiagnsiticsTestDescription variable is to be filtered for Computer Tomography.
  filter(DiagnosticTestDescription=="Computer Tomography")%>%

# Now filter for only the month of October.
  filter(month=="October")%>%

# Group the data by each of the 5 cities.
  group_by(city)%>%

# Create a new column calcuating the mean number of waiting days for each city.
    mutate(mean=mean(WaitingDays))%>%

# Only select the columns city and mean.
  select(city,mean)%>%

# I only want the unique values, so I only need 5 rows of data, one for for each city.
unique()%>%

# Tell Rmarkdown you want to plot a graph.
ggplot()+

# The graph to be plotted is a column bar chart using the variable city as the x-axis
# and the mean number of days as the y-axis.
geom_col(aes(city,mean))
```



Instead of just the basic graph we will make some improvements.

```
join_demo_wait%>%
  select(month,DiagnosticTestDescription,WaitingDays,city)%>%
  filter(DiagnosticTestDescription=="Computer Tomography")%>%
  filter(month=="October")%>%
  group_by(city)%>%
    mutate(mean=mean(WaitingDays))%>%
  select(city,mean)%>%
unique()%>%

  # I reordered the data in ascending order so I could see at a glance the increase in average number o
ggplot(aes(fct_reorder(city,mean),mean))+

  # I gave the graph some colour to make it more visually pleasing. Initially, I just used coral as the
  geom_col(fill="light coral")+

  # I then fixed the names of the x and y axis to include capital letters and to provide more informati
   labs(x="City",y="Average Number of Days", title = "Average number of waiting days for a Computer Tom

# In the original graph the difference between Dundee and Glasgow is not very clear. I changed the y-ax
  coord_cartesian(ylim=c(25,55))+

# I changed the default theme from theme_gray(), grey background to theme_bw(), white background with g
  theme_bw()
```



This section of the code shows the actual values of mean waiting days visualised on the chart.

```
join_demo_wait%>%
  select(month,DiagnosticTestDescription,WaitingDays,city)%>%
  filter(DiagnosticTestDescription=="Computer Tomography")%>%
  filter(month=="October")%>%
  group_by(city)%>%
    mutate(mean=mean(WaitingDays))%>%
  select(city,mean)%>%
unique()
```

```
## # A tibble: 5 x 2
## # Groups:   city [5]
##   city       mean
##   <chr>     <dbl>
## 1 Aberdeen   49.9
## 2 Inverness  43.7
## 3 Edinburgh  47.5
## 4 Dundee     31.7
## 5 Glasgow    30.5
```

Following on from the list of terminology for graphs you may be interested in some other aspects of graphs. If you would rather come back to this, move on to section 5.

5. **Facets** - if you have a graph or chart that contains different values from a variable you may wish to view these separately. If my question had asked, What was the average number of waiting days for CT scans in each Scottish city in Oct 2021 for each gender. I could then use the facets function to create 2 graphs, one for male and female. I would need to make sure that I had the sex variable in the dataset that I was using to create the graph.
6. **Scales** - this is when you want to change the aesthetics using a variable in the dataset.
7. **Stats** - additional analysis can be added to your graphs, such as a line of best fit.

You can find more information on these graphing terms mentioned, here.

---

5. Summary and Reflection

What has been your experience using R for reproducible data analysis?

I have seen the benefits of reproducible work, when I had to change some data within my report. I realised that I didn't have any WaitingDays for 2 of the Scottish cities. The graph therefore only had 3 bars, which I had already created. I wanted more detail in the graph so I had to change the fake data set for ImagingWaitTimes. I added more October Dates and more Computer Tomography Imaging appointments. I was then able to run the RMarkdown file and a new graph was created showing all 5 Scottish cities. This saved lots of time and it provided an opportunity to check the reproducibility of my RMarkdown file.

Please reflect on any problems or issues you encountered, for example whilst creating and merging datasets, or using the R language.

I found it challenging at first to understand the very basic concepts, such as reading in data or how the file structures worked. I didn't full understand the connection between, RProject, Working Directories and having multiple panes in RStudio and then throw in different branches within GitHub for good measure. On several occasions I would receive the error message I'm not in the correct working directory. All I had to do was make sure I was working in the same folder as the RProject file, simple!

However, this initial lack of understanding limited the reproducibility of my work as my folder structures were terrible. Not only this, because I was spending so much time on the basics and constantly revising I felt I hadn't spend enough time practising more complex programming activities such as graphing. I spent a considerable amount of time studying the theory and not enough time doing. I kept studying and practising the programming and I have been able to create this report, which I feel is quite a personal achievement.

I also realised very quickly I had to slow down and really think ahead about what it was I was trying to achieve. I had to ask myself constant questions;

- How is this dataset formatted?
- What functions can I use?
- Is this the expected outcome and if not, why?

The constant reflection alone is not enough to improve reproducibility, but all of the reflection must be documented for others to follow. This is something I am still learning more about. I also found a new file structure that I would like to use in my next project, the structure can be found here. I found it difficult to decide what datasets go in the Input, Method and Output folders, but this new folder structure is much clearer. I also have to learn how to save the output directly into the correct folder.

I enjoy working on problems and learning new skills and I'm sure I'll encounter many more as I continue to develop my RMarkdown and reproducibility skills.

---

6. References

References

1. Bates C. R markdown tips, tricks, and shortcuts [Internet]. Dataquest.io. 2020 [cited 2021 Dec 13]. Available from: https://www.dataquest.io/blog/r-markdown-tips-tricks-and-shortcuts/

2. Foster E. Library guides: How to write a good documentation: Home. 2017 [cited 2021 Dec 13]; Available from: https://guides.lib.berkeley.edu/how-to-write-good-documentation

3. FAIRsharing [Internet]. Fairsharing.org. [cited 2021 Dec 13]. Available from: https://fairsharing.org/standards/

4. Documentation and metadata — the Turing way [Internet]. Netlify.app. [cited 2021 Dec 13]. Available from: https://the-turing-way.netlify.app/reproducible-research/rdm/rdm-metadata.html

5. Europa.eu. [cited 2021 Dec 13]. Available from: https://open-research-europe.ec.europa.eu/for-authors/data-guidelines

6. Tidy data for efficiency, reproducibility, and collaboration [Internet]. Openscapes.org. [cited 2021 Dec 14]. Available from: https://www.openscapes.org/blog/2020/10/12/tidy-data/