Introduction to quantitative analysis with R

Dr Sophie Lee

2025-12-07

Table of contents

Welcome!

Welcome to the Introduction to Quantitative Analysis with R course, designed for and with the Department for Levelling Up, Housing and Communities (DLUCH). This course aims to introduce R and RStudio software. R is an open-source software that was designed to make data analysis more accessible, reproducible, and user friendly.

This two-day course will equip you with the essential skills to leverage the power of R for your data analysis. We will begin with a gentle introduction to the RStudio interface and the basics of the R coding language (or syntax). We will then see how R can be used to efficiently load, clean and transform data. Finally, we will use R to produce clear, compelling visualisations and tables to communicate findings.

Throughout the course, we will discuss best practices for reproducible data analysis, ensuring that all code adheres to the Analysis Standards as recommended by the Aqua book.

> Style tip
>
> Throughout the notes, you will see boxes with 'style tips'. These are to ensure that your code follows the Tidyverse style guide, and ensuring your code adhered to Analysis Standards.

1 Introduction to RStudio

There are a number of software packages based on the R programming language aimed at making writing and running analyses easier for users. They all run R in the background but look different and contain different features. RStudio has been chosen for this course as it allows users to create script files, allowing code to be re-run, edited, and shared easily. RStudio also provides tools to help easily identify errors in R code, integrates help documentation into the main console and uses colour-coding to help read code at a glance.

Before installing RStudio, we must ensure that R is downloaded onto the machine. R is available to download for free for Windows, Mac, or Linux from the CRAN website.

Rstudio is also free to download from the Posit website.

1.1 The RStudio console window

The screenshot below shows the RStudio interface which comprises of four windows:

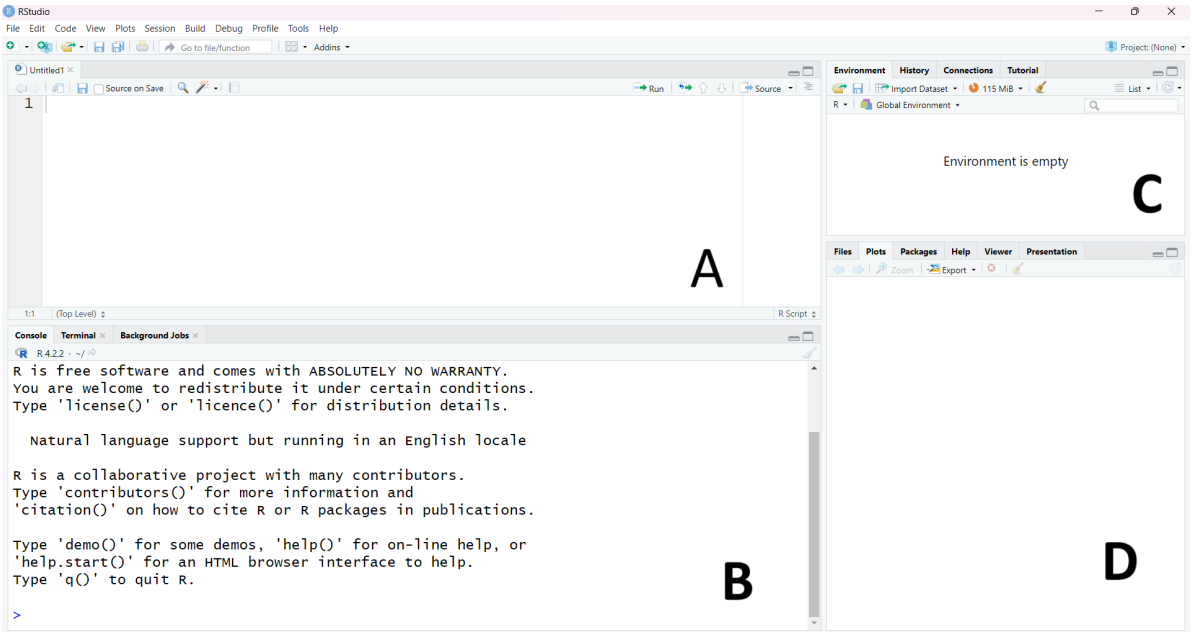


Figure 1.1: RStudio console window

1.1.1 Window A: R script files

All analysis and actions in R are carried out using the R syntax language. R script files allow us to write and edit code before running it in the console window.

> Style tip
>
> Limit script files to 80 characters per line to ensure it is readable.
> RStudio has an option to add a margin that makes this easier to adhere to. Under the *Tools* drop-down menu, select *Global options*. Select *Code* from the list on the right, then under the *Display* tab, tick the *Show margin* box.

If this window is not visible, create a new script file using *File -> New File -> R Script* option from the drop-down menus or clicking the  icon above the console and selecting 'R Script'. This will open a new, blank script file. More than one script file can be open at the same time.

Code entered into the script file does not run automatically. To run commands, highlight the code from the script file and click the 

icon above the top right corner of the script window (this can be carried out by pressing `Ctrl + Enter` in Windows or `Command + Enter` on a Mac computer). Multiple lines of code can be run at once.

The main advantage of using the script file rather than entering the code directly into the console is that it can be saved, edited and shared. To save a script file, use *File -> Save As...* from the drop down menu, click the  icon at the top of the window, or use the keyboard shortcut `ctrl + s` for Windows and `command + s` for Mac. It is important to save the script files at regular intervals to avoid losing work.

> Style tip
>
> Script file names should be meaningful, lower case, and end in `.R`. Avoid using special characters in file names, including spaces. Use `_` instead of spaces.
> Where files should be run in a specific order, prefix the file name with numbers.

Past script files can be opened using *File -> Open File...* from the drop-down menu, by clicking the  icon, or using the keyboard shortcut `ctrl + o` for Windows and `command + o` for Mac, then selecting a *.R file.

1.1.2 Window B: R console

The R console window is where all commands run from the script file, results (other than plots), and messages, such as errors, are displayed. Commands can be written directly into the R

console after the `>` symbol and executed using `Enter` on the keyboard. It is not recommended to write code directly into the console as it is cannot be saved or replicated.

Every time a new R session is opened, details about version and citations of R will be given by default. To clear text from the console window, use the keyboard shortcut `control + l` (this is the same for both Windows and Mac users). Be aware that this clears all text from the console, including any results. Before running this command, check that any results can be replicated within the script file.

1.1.3 Window C: Environment and history

This window lists all data and objects currently loaded into R, and is not available in the basic R software. More details on the types of objects and how to use the Environment window are given in later sections.

1.1.4 Window D: Files, plots, packages and help

This window has many potential uses: plots are displayed and can be saved from here, and R help files will appear here. This window is only available in the RStudio interface and not in the basic R package.

1.2 Exercise 1

1. Open a new script file if you have not already done so.
2. Save this script file into an appropriate location.

2 R syntax

All analyses within R is carried out using syntax, the R programming language. It is important to note that R is case-sensitive so always ensure that you use the correct combination of upper and lower case letters when running functions or calling objects.

Any text written in the R console or script file can be treated the same as text from other documents or programmes: text can be highlighted, copied and pasted to make coding more efficient.

When creating script files, it is important to ensure they are clear and easy to read. Comments can be added to script files using the `#` symbol. R will ignore any text following the `#` on the same line.

> **Style tip**
>
> Combining `#` and `-` creates sections within a script file, making them easier to navigate and organise.
> For example:
>
> ```
> # Load data ----------
>
> # Tidy data ----------
> ```

> **Helpful hint**
>
> To comment out chunks of code, highlight the rows and use the keyboard shortcut *ctrl + shift + c* on Windows, and *Command + shift + c* on Mac

The choice of brackets in R coding is particularly important as they all have different functions:

- Round brackets `( )` are the most commonly used as they define arguments of functions. Any text followed by round brackets is assumed to be a function and R will attempt to run it. If the name of a function is not followed by round brackets, R will return the algorithm used to create the function within the console.
- Square brackets `[ ]` are used to set criteria or conditions within a function or object.

- Curly brackets `{ }` are used within loops, when creating a new function, and within `for` and `if` functions.

All standard notation for mathematical calculations (`+`, `-`, `*`, `/`, `^`, etc.) are compatible with R. At its simplest level, R is just a very powerful calculator!

> Style tip
>
> Although R will work whether a space is added before/after a mathematical operator, the style guide recommends to add them surrounding most mathematical operations (`+`, `-`, `*`, `/`), but not around `^`.
> For example:

```
# Stylish code
1959 - 683
(351 + 457)^2 - (213 + 169)^2

# Un-stylish code
1959-683
(351+457)^2 - (213 + 169) ^ 2
```

2.1 Exercise 2

1. Add your name and the date to the top of your script file (hint: comment this out so R does not try to run it)
2. Use R to answer to following and write the answers within the script file:

a. 64^2
b. $3432 \div 8$
c. 96×72

3 R objects, functions and packages

3.1 Objects

One of the main advantages to using R over other software packages such as SPSS is that more than one dataset can be accessed at the same time. A collection of data stored in any format within the R session is known as an **object**. Objects can include single numbers, single variables, entire datasets, lists of datasets, or even tables and graphs.

> Style tip
>
> Object names should only contain lower case letters, numbers and `_` (instead of a space to separate words). The should be meaningful and concise.

Objects are defined in R using the `<-` symbol or `=`. For example,

```
object_1 <- 81
```

Creates an object in the environment named `object_1`, which takes the value `81`. This will appear in the environment window of the console (window C from the interface shown in the introduction).

> Style tip
>
> Although both work, use `<-` for assignment, not `=`.

To retrieve an object, type its name into the script or console and run it. This object can then be included in functions or operations in place of the value assigned to it:

```
object_1
```

```
[1] 81
```

```
sqrt(object_1)
```

```
[1] 9
```

R has some mathematical objects stored by default such as pi that can be used in calculations.

```
pi
```

```
[1] 3.141593
```

3.2 Functions

Functions are built-in commands that allow R users to run analyses. All functions require the definition of arguments within round brackets `()`. Each function requires different information and has different arguments that can be used to customise the analysis. A detailed list of these arguments and a description of the function can be found in the function's associated **help file**.

3.2.1 Help files

Each function that exists within R has an associated help file. RStudio does not require an internet connection to access these help files if the function is available in the current session of R.

To retrieve help files, enter `?` followed by the function name into the console window, e.g `?mean`. The help file will appear in window D of the interface shown in the introduction.

Help files contain the following information:

- Description: what the function is used for
- Usage: how the function is used
- Arguments: required and optional arguments entered into round brackets necessary for the function to work
- Details: relevant details about the function in question
- References
- See also: links to other relevant functions
- Examples: example code with applications of the function

3.2.2 Error and warning messages

Where a function or object has not been correctly specified, or their is some mistake in the syntax that has been sent to the console, R will return an error message. These messages are generally informative and include the location of the error.

The most common errors include misspelling functions or objects:

```
sqrt(ojbect_1)
```

```
Error in eval(expr, envir, enclos): object 'ojbect_1' not found
```

```
Sqrt(object_1)
```

```
Error in Sqrt(object_1): could not find function "Sqrt"
```

Or where an object has not yet been specified:

```
plot(x, y)
```

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

When R returns an error message, this means that the operation has been completely halted. R may also return warning messages which look similar to errors but does not necessarily mean the operation has been stopped.

Warnings are included to indicate that R suspects something in the operation may be wrong and should be checked. There are occasions where warnings can be ignored but this is only after the operation has been checked.

3.2.3 Cleaning the environment

To remove objects from the RStudio environment, we can use the `rm` function. This can be combined with the `ls()` function, which lists all objects in the environment, to remove all objects currently loaded:

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

> ⚠ Warning
>
> There are no undo and redo buttons for R syntax. The `rm` function will permanently delete objects from the environment. The only way to reverse this is to re-run the code that created the objects originally from the script file.

3.3 Packages

R packages are a collection of functions and datasets developed by R users that expand existing R capabilities or add completely new ones. Packages allow users to apply the most up-to-date methods shortly after they are developed, unlike other statistical software packages that require an entirely new version.

3.3.1 Installing packages from CRAN

The quickest way to install a package in R is by using the `install.packages` function. This sends RStudio to the online repository of tested and verified R packages (known as CRAN) and downloads the package files onto the machine you are currently working from in temporary files. Ensure that the package you wish to install is spelled correctly and surrounded by `''`.

> ⚠ Warning
>
> The `install.packages` function requires an internet connection, and can take a long time if the package has a lot of dependent packages that also need downloading.
> This process should only be carried out the first time a package is used on a machine, or when a substantial update has taken place, to download the latest version of the package.

3.3.2 Loading packages to an R session

Every time a new session of RStudio is opened, packages must be reloaded. To load a package into R (and gain access to the associated functions and data), use the `library` function.

Loading a package does not require an internet connection, but will only work if the package has already been installed and saved onto the computer you are working from. If you are unsure, use the function `installed.packages` to return a list of all packages that are loaded onto the machine you are working from.

Style tip

Begin any script file that requires packages by loading them into the current session. This ensures that there will be no error messages from functions that are not available in the current session.

3.3.3 The pacman package

The `pacman` package is a set of package management functions which is designed to make tasks such as installing and loading packages simpler, and speeds up these processes. There are lots of useful functions included in this package, but the one that we will be using in this course is `p_load`.

`p_load` acts as a wrapper for the `library` function. It first checks the computer to see whether the package(s) listed is installed. If they are, `p_load` loads the package(s) into the current RStudio session. If not, it attempts to install the package(s) from the CRAN repository.

If you have never used the `pacman` package before, run the following code to ensure that it is installed on your machine:

```
install.packages('pacman')
```

4 Opening and exploring data

4.1 Styles of R coding

Up to this point, beyond the style tips sprinkled through these notes, we have not thought about the style of R coding we will be using. There are different approaches to R coding that we can use, they can be thought of as different dialects of the R programming language.

The choice of R 'dialect' depends on personal preference. Some prefer to use the 'base R' approach that does not rely on any packages that may need updating, making it a more stable approach. However, base R can be difficult to read for those not comfortable with coding.

The alternative approach that we will be adopting in this course is the 'tidyverse' approach. Tidyverse is a set of packages that have been designed to make R coding more readable and

efficient. They have been designed with reproducibility in mind, which means there is a wealth of online (mostly free), well-written resources available to help use these packages. Tidyverse is also the preferred coding style of the Government Analysis Function guidance.

If you have not done so already, install the Tidyverse packages to your machine using the following code:

```
install.packages('tidyverse')
```

> **Warning**
>
> This can take a long time if you have never downloaded the tidyverse packages before as there are many dependencies that are required.
> Do not stress if you get a lot of text in the console! This is normal, but watch out for any error messages.

Once the tidyverse package is installed, we must load it into the current working session. At the beginning of your script file add the following syntax:

```
pacman::p_load(tidyverse)
```

> **Style tip**
>
> The double colon in R can be used to run a function within an installed package without loading the entire package to an R session.

4.2 The working directory

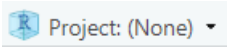
The working directory is a file path on your computer that R sets as the default location when opening, saving, or exporting documents, files, and graphics. This file path can be specified manually but setting the working directory saves time and makes code more efficient.

The working directory can be set manually by using the *Session -> Set Working Directory -> Change Directory…* option from the drop-down menu, or the `setwd` function. Both options require the directory to be specified each time R is restarted, are sensitive to changes in folders within the file path, and cannot be used when script files are shared between colleagues.

An alternative approach that overcomes all these issues is to create an R project.

4.2.1 R projects

R projects are files (saved with the `.Rproj` extension) that keep associated files (including scripts, data, and outputs) grouped together. An R project automatically sets the working directory relative to its current location, which makes collaborative work easier, and avoids issues when a file path is changed.

Projects are created by using the *File -> New project* option from the drop-down menu, or using the 

icon from the top-right corner of the RStudio interface. Existing projects can be opened under the *File -> Open project…* drop-down menu or using the project icon.

When creating a new project, we must choose whether we are creating a new directory or using an existing one. Usually, we will have already set up a folder containing data or other documents related to the analysis we plan to carry out. If this is the case, we are using an existing directory and selecting the analysis folder as the project directory.

> Style tip
>
> Have a clear order to your analysis folder. Consider creating separate folders within a project for input and output data, documentation, and outputs such as graphs or tables.

4.3 Loading data

To ensure our code is collaborative and reproducible, we should strive to store data in formats that can be used across multiple platforms. One of the best ways to do this is to store data as a comma-delimited file (.csv). CSV files can be opened by a range of different softwares (including R, SPSS, STATA and excel), and base R can be used to open these files without requiring additional packages.

Unfortunately, we are not always able to choose the format that data are stored in. For example, the English Housing Survey (EHS) data is stored as a .sav (SPSS) data file. Fortunately for us, R has a wide range of packages that have been developed to load data from every conceivable format.

The package that we will be using the load SPSS data is the `haven` package. To ensure this is loaded in at the beginning of each session, adapt the previous `p_load` function:

```
pacman::p_load(tidyverse, haven)
```

To avoid any errors arising from spelling mistakes, we can use the `list.files` function, which returns a list of files and folders from the current working directory. The file names can be

copied from the console and pasted into the script file. As the data are saved in a folder within the working directory, we add the argument `path =` to specify the folder we want to list files from.

```
list.files(path = "data")
```

```
[1] "Detailed_forecast_tables_Economy_March_2024.xlsx"
[2] "generalfs21_EUL.sav"
[3] "interviewfs21_EUL.sav"
```

The first data set we will load is the`generalfs21_EUL.sav` file. This contains general information taken from the English Housing Survey (EHS) from 2021, including a unique identifier, the responders' region, and the tenure type.

The EHS data can be loaded into R using the `read_spss` function, and saved as an object using the `<-` symbol:

```
ehs_general <- read_spss(file = "Data/generalfs21_EUL.sav")
```

The imported data will appear in the environment with its given name. The contents of the object can be viewed by clicking on the object name in the environment, opening a tab next to script files. This window is a preview so cannot be edited here.

Some useful functions that can be used to explore a dataset include:

```
# Return variable names
names(ehs_general)
```

```
[1] "serialanon" "aagfh21"    "paired"     "tenure8x"   "tenure4x"
[6] "tenure2x"   "gorehs"     "region3x"   "govreg1"
```

```
# Returns the first 6 rows
head(ehs_general)
```

```
# A tibble: 6 x 9
  serialanon  aagfh21   paired      tenure8x tenure4x tenure2x gorehs   region3x
  <dbl+lbl>   <dbl+lbl> <dbl+lbl>   <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb>
1 20220000001 3934.     1 [Paired]  1 [owne~ 1 [owne~ 1 [Priv~  7 [Eas~ 3 [Rest~
2 20220000005 1580.     1 [Paired]  1 [owne~ 1 [owne~ 1 [Priv~ 10 [Sou~ 3 [Rest~
3 20220000006 3360.     1 [Paired]  1 [owne~ 1 [owne~ 1 [Priv~  6 [Wes~ 3 [Rest~
4 20220000012 1368.     0 [Not pai~ 1 [owne~ 1 [owne~ 1 [Priv~  5 [Eas~ 3 [Rest~
```

```
5 20220000013 9847.     1 [Paired]  1 [owne~ 1 [owne~ 1 [Priv~ 10 [Sou~ 3 [Rest~
6 20220000017 3262.     0 [Not pai~ 1 [owne~ 1 [owne~ 1 [Priv~  6 [Wes~ 3 [Rest~
# i 1 more variable: govreg1 <dbl+lbl>
```

```
# Returns the last 6 rows
tail(ehs_general)
```

```
# A tibble: 6 x 9
  serialanon  aagfh21   paired      tenure8x tenure4x tenure2x gorehs   region3x
  <dbl+lbl>   <dbl+lbl> <dbl+lbl>   <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb>
1 20220031393  262.     0 [Not pai~ 3 [loca~ 3 [loca~ 2 [Soci~  7 [Eas~ 3 [Rest~
2 20220031396 5741.     1 [Paired]  1 [owne~ 1 [owne~ 1 [Priv~  6 [Wes~ 3 [Rest~
3 20220031401 1579.     0 [Not pai~ 1 [owne~ 1 [owne~ 1 [Priv~  6 [Wes~ 3 [Rest~
4 20220031402 2178.     0 [Not pai~ 1 [owne~ 1 [owne~ 1 [Priv~  9 [Sou~ 2 [Lond~
5 20220031408 1133.     1 [Paired]  1 [owne~ 1 [owne~ 1 [Priv~ 10 [Sou~ 3 [Rest~
6 20220031409 1879.     1 [Paired]  1 [owne~ 1 [owne~ 1 [Priv~ 10 [Sou~ 3 [Rest~
# i 1 more variable: govreg1 <dbl+lbl>
```

```
# Gives information about the structure of an object (including variable types)
str(ehs_general)
```

```
tibble [9,752 x 9] (S3: tbl_df/tbl/data.frame)
 $ serialanon: dbl+lbl [1:9752] 2.02e+10, 2.02e+10, 2.02e+10, 2.02e+10, 2.02e+10, 2.0...
   ..@ label        : chr "Key variable: unique archived identifier"
   ..@ format.spss  : chr "F11.0"
   ..@ display_width: int 13
   ..@ labels       : Named num [1:2] -9 -8
   .. ..- attr(*, "names")= chr [1:2] "Does not apply" "No Answer"
 $ aagfh21   : dbl+lbl [1:9752] 3934, 1580, 3360, 1368, 9847, 3262, 6584,  389, 4193,...
   ..@ label        : chr "Household weight 2021"
   ..@ format.spss  : chr "F8.2"
   ..@ display_width: int 10
   ..@ labels       : Named num [1:2] -9 -8
   .. ..- attr(*, "names")= chr [1:2] "Does not apply" "No answer"
 $ paired    : dbl+lbl [1:9752] 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1,...
   ..@ label      : chr "Whether paired sample case"
   ..@ format.spss: chr "F3.0"
   ..@ labels     : Named num [1:4] -9 -8 0 1
   .. ..- attr(*, "names")= chr [1:4] "Does not apply" "No Answer" "Not paired" "Paired"
 $ tenure8x  : dbl+lbl [1:9752] 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 4, 1, 4, 1, 1, 2, 4,...
   ..@ label        : chr "Tenure with vacancy"
```

```
  ..@ format.spss  : chr "F2.0"
  ..@ display_width: int 10
  ..@ labels       : Named num [1:10] -9 -8 1 2 3 4 5 6 7 8
  .. ..- attr(*, "names")= chr [1:10] "Does not apply" "No Answer" "owner occupied - occupie
$ tenure4x  : dbl+lbl [1:9752] 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 4, 1, 4, 1, 1, 2, 4,...
  ..@ label        : chr "Tenure"
  ..@ format.spss  : chr "F2.0"
  ..@ display_width: int 10
  ..@ labels       : Named num [1:6] -9 -8 1 2 3 4
  .. ..- attr(*, "names")= chr [1:6] "Does not apply" "No Answer" "owner occupied" "private
$ tenure2x  : dbl+lbl [1:9752] 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 2,...
  ..@ label        : chr "Tenure"
  ..@ format.spss  : chr "F2.0"
  ..@ display_width: int 10
  ..@ labels       : Named num [1:4] -9 -8 1 2
  .. ..- attr(*, "names")= chr [1:4] "Does not apply" "No Answer" "Private" "Social"
$ gorehs    : dbl+lbl [1:9752]  7, 10,  6,  5, 10,  6,  4, 10,  7,  2, 10,  2,  2,  ...
  ..@ label      : chr "Government Office Region EHS version"
  ..@ format.spss: chr "F2.0"
  ..@ labels     : Named num [1:11] -9 -8 1 2 4 5 6 7 8 9 ...
  .. ..- attr(*, "names")= chr [1:11] "Does not apply" "No answer" "North East" "North West
$ region3x  : dbl+lbl [1:9752] 3, 3, 3, 3, 3, 3, 1, 3, 3, 1, 3, 1, 1, 3, 1, 2, 3, 1,...
  ..@ label        : chr "Overall region of England"
  ..@ format.spss  : chr "F2.0"
  ..@ display_width: int 10
  ..@ labels       : Named num [1:5] -9 -8 1 2 3
  .. ..- attr(*, "names")= chr [1:5] "Does not apply" "No Answer" "Northern regions" "Londo
$ govreg1   : dbl+lbl [1:9752] 4, 4, 2, 2, 4, 2, 1, 4, 4, 1, 4, 1, 1, 2, 1, 4, 4, 1,...
  ..@ label        : chr "Government office Region, grouped"
  ..@ format.spss  : chr "F2.0"
  ..@ display_width: int 9
  ..@ labels       : Named num [1:6] -9 -8 1 2 3 4
  .. ..- attr(*, "names")= chr [1:6] "Does not apply" "No Answer" "North" "Midlands" ...
 - attr(*, "notes")= chr [1:6] "This third delivery file (interviewfs21.sav)" "was updated o
```

The `str` function tells us that this object is a `tibble`. This is tidyverse language for a data set (in base R, it is known as a `data.frame`). All variables are recognised as `dbl + lbl`, or labelled double variables. Double is tidyverse language for numeric data, and labels are taken from the original SPSS data.

It is important to check that R has correctly recognised variable type when data are loaded, before generating any visualisations or analysis. If variables are incorrectly specified, this could

either lead to errors or invalid analyses. We will see how to change variables types later in this chapter.

The variables in this tibble contain additional information, stored as `attributes`. Data imported from other sources do not typically include these attributes by default, but these are able to uphold any information that was stored in the 'Variable view' window of SPSS.

4.4 Selecting variables

Often, we will not need every variable in a downloaded dataset to carry out an analysis, and we may wish to create a smaller analysis tibble. We may also wish to select individual variables from the tibble to apply functions to them without including the entire dataset.

To select one or more variable and return them as a new tibble, we can use the `select` function from tidyverse's `dplyr` package.

For example, we do not need all the variables contained in the EHS general dataset. The variables we are interested in keeping are the unique identifier variables (`serialanon`), the survey weights (`aagfh21`), the tenure type response with 4 options (`tenure4x`), and the Government office region (`govreg1`):

```
# Variables can be selected using their names
select(ehs_general, serialanon, aagfh21, tenure4x, gorehs)
```

```
# A tibble: 9,752 x 4
   serialanon  aagfh21   tenure4x           gorehs
   <dbl+lbl>   <dbl+lbl> <dbl+lbl>          <dbl+lbl>
 1 20220000001 3934.     1 [owner occupied]  7 [East]
 2 20220000005 1580.     1 [owner occupied] 10 [South West]
 3 20220000006 3360.     1 [owner occupied]  6 [West Midlands]
 4 20220000012 1368.     1 [owner occupied]  5 [East Midlands]
 5 20220000013 9847.     1 [owner occupied] 10 [South West]
 6 20220000017 3262.     1 [owner occupied]  6 [West Midlands]
 7 20220000022 6584.     1 [owner occupied]  4 [Yorkshire and the Humber]
 8 20220000025  389.     1 [owner occupied] 10 [South West]
 9 20220000026 4193.     2 [private rented]  7 [East]
10 20220000027 3589.     1 [owner occupied]  2 [North West]
# i 9,742 more rows
```

```
# Or their column number
select(ehs_general, 1, 2, 5, 7)
```

```
# A tibble: 9,752 x 4
   serialanon  aagfh21   tenure4x           gorehs
   <dbl+lbl>   <dbl+lbl> <dbl+lbl>          <dbl+lbl>
 1 20220000001 3934.     1 [owner occupied]  7 [East]
 2 20220000005 1580.     1 [owner occupied] 10 [South West]
 3 20220000006 3360.     1 [owner occupied]  6 [West Midlands]
 4 20220000012 1368.     1 [owner occupied]  5 [East Midlands]
 5 20220000013 9847.     1 [owner occupied] 10 [South West]
 6 20220000017 3262.     1 [owner occupied]  6 [West Midlands]
 7 20220000022 6584.     1 [owner occupied]  4 [Yorkshire and the Humber]
 8 20220000025  389.     1 [owner occupied] 10 [South West]
 9 20220000026 4193.     2 [private rented]  7 [East]
10 20220000027 3589.     1 [owner occupied]  2 [North West]
# i 9,742 more rows
```

The `select` function can also be combined with a number of 'selection helper' functions that help us select variables based on naming conventions:

- `starts_with("xyz")` returns all variables with names beginning `xyz`
- `ends_with("xyz")` returns all variables with names ending `xyz`
- `contains("xyz")` returns all variables that have `xyz` within their name

Or based on whether they match a condition:

- `where(is.numeric)` returns all variables that are classed as numeric

For a full list of these selection helpers, access the helpfile using `?tidyr_tidy_select`.

The `select` function can also be used to remove variables from a tibble by adding a `-` before the variable name or number. For example, to return the EHS general dataset without the unique identifier variable, we use:

```
select(ehs_general, -serialanon)
```

```
# A tibble: 9,752 x 8
   aagfh21   paired         tenure8x tenure4x tenure2x gorehs   region3x govreg1
   <dbl+lbl> <dbl+lbl>      <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb> <dbl+l>
 1 3934.     1 [Paired]     1 [owne~ 1 [owne~ 1 [Priv~  7 [Eas~ 3 [Rest~ 4 [Res~
 2 1580.     1 [Paired]     1 [owne~ 1 [owne~ 1 [Priv~ 10 [Sou~ 3 [Rest~ 4 [Res~
 3 3360.     1 [Paired]     1 [owne~ 1 [owne~ 1 [Priv~  6 [Wes~ 3 [Rest~ 2 [Mid~
 4 1368.     0 [Not paired] 1 [owne~ 1 [owne~ 1 [Priv~  5 [Eas~ 3 [Rest~ 2 [Mid~
 5 9847.     1 [Paired]     1 [owne~ 1 [owne~ 1 [Priv~ 10 [Sou~ 3 [Rest~ 4 [Res~
 6 3262.     0 [Not paired] 1 [owne~ 1 [owne~ 1 [Priv~  6 [Wes~ 3 [Rest~ 2 [Mid~
```

```
 7 6584.     0 [Not paired] 1 [owne~ 1 [owne~ 1 [Priv~  4 [Yor~ 1 [Nort~ 1 [Nor~
 8  389.     0 [Not paired] 1 [owne~ 1 [owne~ 1 [Priv~ 10 [Sou~ 3 [Rest~ 4 [Res~
 9 4193.     1 [Paired]     2 [priv~ 2 [priv~ 1 [Priv~  7 [Eas~ 3 [Rest~ 4 [Res~
10 3589.     1 [Paired]     1 [owne~ 1 [owne~ 1 [Priv~  2 [Nor~ 1 [Nort~ 1 [Nor~
# i 9,742 more rows
```

After making changes to the analysis dataset, it is useful to save this data separately to the original raw data. This can be done using the `write_csv` function.

```
ehs_general_reduced <- select(ehs_general, serialanon, aagfh21, tenure4x, gorehs)
```

```
write_csv(ehs_general_reduced, file = "ehs_general_reduced.csv")
```

> **Warning**
>
> When saving updated tibbles as files, use a different file name to the original raw data. Using the same name will overwrite the original file. We always want a copy of the original in case of any errors or issues.

The `select` function returns variables as a tibble object. However, some functions, for example summary functions from base R, require data in the form of a `vector`. Vectors are lists of numbers with no formal structure, unlike a tibble which is structured to have rows and columns. To return a single variable as a vector, we can use the `$` symbol between the data name and the variable to return:

```
ehs_general_reduced$aagfh21
```

4.5 Filtering data

The `filter` function, from tidyverse's `dplyr` package allows us to return subgroups of the data based on conditional statements. These conditional statements can include mathematical operators, e.g. `<=` (less than or equal to), `==` (is equal to), and `!=` (is not equal to), or can be based on conditional functions, e.g. `is.na(variable)` (is missing), `between(a, b)` (number lies between a and b). A list of these conditional statements can be found in the help file using `?filter`.

For example, we may wish to return rows from the EHS dataset that were privately rented. To check which number refers to privately rented, we can check the `labels` attribute of the `tenure4x` variable which shows the labels from the SPSS file:

```
attributes(ehs_general_reduced$tenure4x)$labels
```

```
     Does not apply           No Answer      owner occupied      private rented
                 -9                  -8                   1                   2
    local authority housing association
                  3                   4
```

Private rented is given by a 2 in the current dataset, therefore our conditional statement will return rows where the `tenure4x` variable takes the value 2.

```
filter(ehs_general_reduced, tenure4x == 2)
```

```
# A tibble: 1,735 x 4
   serialanon  aagfh21   tenure4x           gorehs
   <dbl+lbl>   <dbl+lbl> <dbl+lbl>          <dbl+lbl>
 1 20220000026 4193.     2 [private rented]  7 [East]
 2 20220000039  423.     2 [private rented]  7 [East]
 3 20220000075 1017.     2 [private rented]  7 [East]
 4 20220000092 1554.     2 [private rented]  7 [East]
 5 20220000113 5254.     2 [private rented]  8 [London]
 6 20220000132 7609.     2 [private rented]  4 [Yorkshire and the Humber]
 7 20220000134  775.     2 [private rented]  5 [East Midlands]
 8 20220000135 1313.     2 [private rented]  1 [North East]
 9 20220000187 1528.     2 [private rented]  8 [London]
10 20220000198  557.     2 [private rented] 10 [South West]
# i 1,725 more rows
```

Multiple conditional statements can be added to the same function by separating them with a comma `,`. To return all respondents that lived in privately rented accommodation in the North East, we can extend the previous filter statement:

```
attributes(ehs_general_reduced$gorehs)$labels
```

```
          Does not apply                No answer               North East
                      -9                       -8                        1
              North West Yorkshire and the Humber           East Midlands
                       2                        4                        5
           West Midlands                     East                   London
                       6                        7                        8
              South East               South West
                       9                       10
```

```
# North East is region 1
filter(ehs_general_reduced, tenure4x == 2, gorehs == 1)
```

```
# A tibble: 76 x 4
   serialanon  aagfh21   tenure4x           gorehs
   <dbl+lbl>   <dbl+lbl> <dbl+lbl>          <dbl+lbl>
 1 20220000135  1313.    2 [private rented] 1 [North East]
 2 20220001633  3849.    2 [private rented] 1 [North East]
 3 20220002101  1016.    2 [private rented] 1 [North East]
 4 20220002817   323.    2 [private rented] 1 [North East]
 5 20220003959  2800.    2 [private rented] 1 [North East]
 6 20220004183   636.    2 [private rented] 1 [North East]
 7 20220005151 10707.    2 [private rented] 1 [North East]
 8 20220005393  2189.    2 [private rented] 1 [North East]
 9 20220005697  2059.    2 [private rented] 1 [North East]
10 20220005754   625.    2 [private rented] 1 [North East]
# i 66 more rows
```

4.6 Pipes

When creating an analysis-ready dataset, we often want to combine functions such as `select` and `filter`. Previously, these would need to be carried out separately and a new object would need to be created or overwritten at each step, clogging up the environment.

In tidyverse, we combine functions within a single process using the 'pipe' symbol **%>%**, which is read as 'and then' within the code. For example, if we wanted to just select the unique identifiers of respondents that were privately renting in the North East, we could do this in a single process:

```
ehs_general_reduced %>%
  filter(tenure4x == 2, gorehs == 1) %>%
  select(serialanon)
```

```
# A tibble: 76 x 1
   serialanon
   <dbl+lbl>
 1 20220000135
 2 20220001633
 3 20220002101
 4 20220002817
```

```
 5 20220003959
 6 20220004183
 7 20220005151
 8 20220005393
 9 20220005697
10 20220005754
# i 66 more rows
```

> **Style tips**
>
> When combining multiple functions within a process using pipes, it is good practice to start the code with the data and pipe that into the functions, rather than including it in the function itself.

> **Helpful hint**
>
> Rather than typing out pipes every time, use the keyboard shortcut *ctrl + shift + m* for Windows and *Command + shift + m* for Mac.

4.7 Creating new variables

The function `mutate` from tidyverse's `dplyr` package allows us to add new variables to a dataset. We can add multiple variables within the same function, separating each with a comma `,`.

The `mutate` function is helpful when variable types are not correctly specified by R when they are read in. For example, the region and tenancy type variables in the `ehs_general_reduced` tibble are categorical variables but are currently recognised as numeric.

Categorical variables in R are known as **factors**. These factors can be ordered and can have labels assigned to different levels. To convert an existing variable to a factor, we can use the `factor` or `as_factor` functions. Here, we can combine the `mutate` and `as_factor` functions to convert tenancy type and region to factors:

```
ehs_general_reduced <- mutate(ehs_general_reduced,
                              tenancy_type = as_factor(tenure4x),
                              region = as_factor(gorehs))
```

> **Note**
>
> As this data was taken from an SPSS file that had labels attached to the grouped variables, we do not need to specify these within the `as_factor` function.
> When the variables do not have this labelling structure already, they will need to be added using the `label` argument of the `factor` function (see `?factor` for more information).

The `mutate` function can also be used to convert numeric variables into an ordered categorical variable, and can be used to transform variables using mathematical functions. For example, we can create two new variables, first giving the square root of the weighting variable, and second grouping the weighting variable into three categories (low: < 1000, medium: $1000 \leq$ `aagfh21` < 5000, high: ≥ 5000):

```
ehs_general_reduced <- mutate(ehs_general_reduced,
                              weighting_sqrt = sqrt(aagfh21),
                              weighting_fct = cut(aagfh21,
                                                  breaks = c(0, 1000, 5000, Inf),
                                                  right = TRUE,
                                                  labels = c("Low", "Medium",
                                                             "High")))
```

> **Helpful hint**
>
> The `c` function takes a list of values separated by commas and returns them as a `vector`. This is useful when a function argument requires multiple values (and we don't want R to move onto the next argument, which is what a comma inside functions usually means).

4.8 Other useful `dplyr` functions

To ensure our code follows the tidyverse style guide, variable names should be concise, *informative*, and contain no special charaters (other than _). The original variable names given in the original EHS data were definitely not stylish! To change names in a dataset, we can use the `rename` function:

```
ehs_general_reduced <- rename(ehs_general_reduced,
                              id = serialanon,
                              weighting = aagfh21)
```

For more useful data exploration and manipulation functions from the `dplyr` package, I would recommending taking a look at the **vignette** associated with the package (a long-form version of a help file):

`vignette("dplyr")`

Or look at the `dplyr` cheatsheet.

4.9 A smooth process to the analysis dataset

Our EHS analysis dataset has been created haphazardly through this chapter to demostrate each step separately. In reality, we would load this data and manipulate it in one process, separating steps by pipes %>%.

The code below takes the data from its saw form (the .sav file) and transforms it into a clean dataset that we will be using for the rest of the course:

```
# Step 1: load the dataset into R and attach as an object
ehs_general_tidy <- read_spss(file = "Data/generalfs21_EUL.sav") %>%
# Step 2: convert grouping variables into factors
                    mutate(tenure_type = as_factor(tenure4x),
                           region = as_factor(gorehs)) %>%
# Step 3: rename other variables to avoid confusion
                    rename(id = serialanon,
                           weighting = aagfh21) %>%
# Step 4: keep only the necessary variables
                    select(id, weighting, tenure_type, region)

# Check the new data looks correct
str(ehs_general_tidy)
```

```
tibble [9,752 x 4] (S3: tbl_df/tbl/data.frame)
 $ id         : dbl+lbl [1:9752] 2.02e+10, 2.02e+10, 2.02e+10, 2.02e+10, 2.02e+10, 2.0...
   ..@ label        : chr "Key variable: unique archived identifier"
   ..@ format.spss  : chr "F11.0"
   ..@ display_width: int 13
   ..@ labels       : Named num [1:2] -9 -8
   .. ..- attr(*, "names")= chr [1:2] "Does not apply" "No Answer"
 $ weighting  : dbl+lbl [1:9752] 3934, 1580, 3360, 1368, 9847, 3262, 6584,  389, 4193,...
   ..@ label        : chr "Household weight 2021"
   ..@ format.spss  : chr "F8.2"
   ..@ display_width: int 10
   ..@ labels       : Named num [1:2] -9 -8
   .. ..- attr(*, "names")= chr [1:2] "Does not apply" "No answer"
 $ tenure_type: Factor w/ 6 levels "Does not apply",..: 3 3 3 3 3 3 3 3 4 3 ...
  ..- attr(*, "label")= chr "Tenure"
```

```
 $ region     : Factor w/ 11 levels "Does not apply",..: 8 11 7 6 11 7 5 11 8 4 ...
  ..- attr(*, "label")= chr "Government Office Region EHS version"
 - attr(*, "notes")= chr [1:6] "This third delivery file (interviewfs21.sav)" "was updated on
```

```
# Step 5: save this tidy data as a new file in a saved_data folder
write_csv(ehs_general_tidy, file = "saved_data/ehs_general_tidy.csv")
```

4.10 Exercise 3

1. You have been provided with another .sav file which contains the interview responses from the EHS. Create and save a tidy version of this dataset, ensuring variables are classified as the correct type and names follow the style conventions (if you cannot remember these, check here for a reminder.

The variables we need in the tidy dataset are:

- The unique identifier `serialanon`
- The gross household income `HYEARGRx`
- The length of residence `lenresb`
- The weekly rent `rentwkx` and mortgage `mortwkx` payments
- Whether the property is freehold or leasehold `freeLeas`

2. Save the tidy interview dataset as a csv file with an appropriate file name.
3. Using the new, tidy dataset, answer the following questions:

- How many respondents paid weekly rent of between £150 and £300?
- How many respondents did not give a response to either the weekly rent or weekly mortgage question?
- What is the highest household gross income of these responders?

5 Combining and summarising datasets

5.1 Combining multiple datasets

Both the SPSS datasets we have been working with so far have contained different information about the English Housing Survey (EHS). We will need to join these together to create a single analysis dataset with all the information we need.

First we need to reload the tidy datasets we saved previously (now using the `read_csv` function):

```
ehs_general_tidy <- read_csv("saved_data/ehs_general_tidy.csv")
```

```
Rows: 9752 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): tenure_type, region
dbl (2): id, weighting

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
ehs_interview_tidy <- read_csv("saved_data/ehs_interview_tidy.csv")
```

```
Rows: 9752 Columns: 6
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): length_residence, freehold_leasehold
dbl (4): id, gross_income, weekly_rent, weekly_mortgage

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Notice that by default, the variables that were classed as factors have been recognised by R as `chr` (character). This is because CSV files are unable to store the grouping attributes that were created in R. Therefore, when we load in CSV files, we need to use the `mutate` function to re-classify these variables.

When we need to apply the same function to a group of variables within a dataset, the `mutate` function can be combined with `across`, which uses selection helpers (see `?dplyr_tidy_select`) and retains the original variable names

```
ehs_general_tidy <- read_csv("saved_data/ehs_general_tidy.csv") %>%
  mutate(across(where(is.character), factor))


ehs_interview_tidy <- read_csv("saved_data/ehs_interview_tidy.csv") %>%
  mutate(across(where(is.character), factor))
```

> Style tips
>
> When writing script files, we want our code to be as concise and efficient as possible. Although we could use mutate to apply the `factor` function to each of the categorical variables, using the wrapper `across` reduces the amount of code needed and, consequently, the risk of errors.

Joining datasets can be carried out using `join` functions. There are 4 options we can choose from depending on which observations we want to keep if not all of them are matched (see `?full_join` for a full list of options).

In this example, we want to keep all observations, even if they are missing from one of the datasets. This requires the `full_join` function. Both datasets contain a unique identifier which can be included in the `full_join` function to ensure we are joining like-for-like:

```
ehs_tidy <- full_join(ehs_general_tidy, ehs_interview_tidy,
                      by = "id")

head(ehs_tidy)
```

```
# A tibble: 6 x 9
       id weighting tenure_type region gross_income length_residence weekly_rent
    <dbl>     <dbl> <fct>       <fct>         <dbl> <fct>                  <dbl>
1 2.02e10     3934. owner occu~ East         38378. 20-29 years               NA
2 2.02e10     1580. owner occu~ South~       26525  30+ years                 NA
3 2.02e10     3360. owner occu~ West ~       25272. 10-19 years               NA
4 2.02e10     1368. owner occu~ East ~       51280. 3-4 years                 NA
```

```
5 2.02e10     9847. owner occu~ South~       14365  30+ years                NA
6 2.02e10     3262. owner occu~ West ~       38955  30+ years                NA
# i 2 more variables: weekly_mortgage <dbl>, freehold_leasehold <fct>
```

5.2 Summarising data

Summary tables can be created using the `summarise` function. This returns tables in a tibble format, meaning they can easily be customised and exported as CSV files (using the `write_csv` function).

The `summarise` function is set up similarly to the `mutate` function: summaries are listed and given variable names, separated by a comma. The difference between these functions is that `summarise` collapses the tibble into a single summary, and the new variables must be created using a summary function.

Common examples of summary functions include:

- `mean`
- `median`
- `range` (gives the minimum and maximum values)
- `min`
- `max`
- `IQR` (interquartile range, gives the range of the middle 50% of the sample)
- `sd` (standard deviation, a measure of the spread when data are normally distributed)
- `sum`
- `n` (counts the number of rows the summary is calculated from)

For example, if we want to generate summaries of the gross household income using the entire dataset:

```
summarise(ehs_tidy,
          total_income = sum(gross_income),
          median_income = median(gross_income),
          n_rows = n())
```

```
# A tibble: 1 x 3
  total_income median_income n_rows
         <dbl>         <dbl>  <int>
1   397744132.        34016.   9752
```

The `summarise` function can be used to produce grouped summaries. This is done by first grouping the data with the `group_by` function.

Whenever using `group_by`, make sure to `ungroup` the data before proceeding. The grouping structure can be large and slow analysis, or may interact with other functions to produce unexpected analyses.

For example, we can expand the gross income summary table to show these summaries separated by region:

```
ehs_tidy %>%
  group_by(region) %>%
  summarise(total_income = sum(gross_income),
            median_income = median(gross_income),
            n_rows = n()) %>%
  ungroup()
```

```
# A tibble: 9 x 4
  region                    total_income median_income n_rows
  <fct>                            <dbl>         <dbl>  <int>
1 East                          55188890.        37225   1275
2 East Midlands                 31706518.        32453    806
3 London                        59369895.        42278.  1199
4 North East                    15381138.        26324.   474
5 North West                    51011810.        29642   1410
6 South East                    72646076.        39087.  1600
7 South West                    44236539.        34958.  1105
8 West Midlands                 32522051.        30984.   878
9 Yorkshire and the Humber      35681216.        29900   1005
```

Before creating summary tables, it is important to consider the most appropriate choice of summary statistics for your data.

5.2.1 Summarising categorical data

To summarise a single categorical variable, we simply need to quantify the distribution of observations lying in each group. The simplest way to do this is to count the number of observations that lie in each group. However, a simple count can be difficult to interpret without proper context. Often, we wish to present these counts relative to the total sample that they are taken from.

The proportion of observations in a given group is estimated as the number in the group divided by the total sample size. This gives a value between 0 and 1. Multiplying the proportion by 100 will give the percentage in each group, taking the value between 0 and 100%.

For example, to calculate the proportion of respondents that live in privately rented properties, we divide the total number in that group by the total number of respondents:

```
ehs_tidy %>%
  # First group the data by tenure type
  group_by(tenure_type) %>%
  # Then count the number of rows in each of these group (types)
  summarise(n_tenancy = n()) %>%
  # Be sure to ungroup to remove this structure!
  ungroup() %>%
  # Now calculate the total number of respondents overall
  mutate(n_responses = sum(n_tenancy),
  # and divide the group total by the overall total
         prop_tenure = n_tenancy / n_responses)
```

```
# A tibble: 4 x 4
  tenure_type         n_tenancy n_responses prop_tenure
  <fct>                   <int>       <int>       <dbl>
1 housing association      1429        9752      0.147
2 local authority           971        9752      0.0996
3 owner occupied           5617        9752      0.576
4 private rented           1735        9752      0.178
```

From this summary table, the proportion of responders that lived in privately rented properties was 0.1779. To convert this into a percentage, we multiple the proportions by 100%:

```
ehs_tidy %>%
  # First group the data by tenure type
  group_by(tenure_type) %>%
  # Then count the number of rows in each of these group (types)
  summarise(n_tenancy = n()) %>%
  # Be sure to ungroup to remove this structure!
  ungroup() %>%
  # Now calculate the total number of respondents overall
  mutate(n_responses = sum(n_tenancy),
  # and divide the group total by the overall total
         prop_tenure = n_tenancy / n_responses,
         perc_tenure = prop_tenure * 100)
```

```
# A tibble: 4 x 5
  tenure_type         n_tenancy n_responses prop_tenure perc_tenure
  <fct>                   <int>       <int>       <dbl>       <dbl>
1 housing association      1429        9752      0.147        14.7
2 local authority           971        9752      0.0996        9.96
3 owner occupied           5617        9752      0.576        57.6
4 private rented           1735        9752      0.178        17.8
```

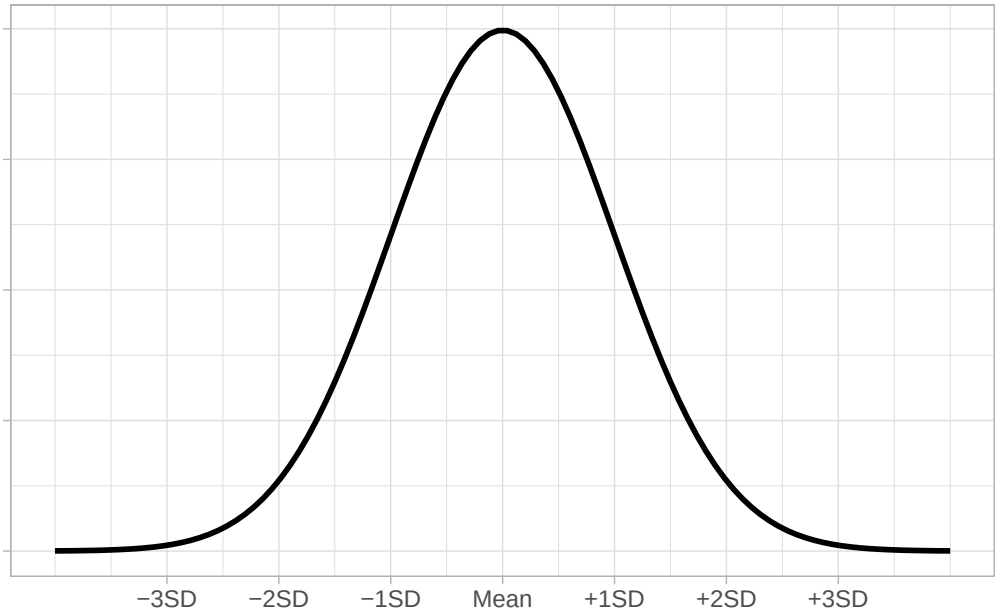
Therefore, 17.79% of responders lived in privately rented properties.

5.2.2 Summarising numeric variables

Numeric variables are typically summarised using the centre of the variable, also known as the average, and a measure of the spread of the variable. The most appropriate choice of summary statistics will depend on the distribution of the variable. More specifically, whether the numeric variable is normally distributed or not. The shape/distribution of a variable is typically investigated by plotting data in a histogram.

5.2.2.1 Measures of centre

The average of a numeric variable is another way of saying the centre of its distribution. Often, people will think of the **mean** when trying to calculate an average, however this may not always be the case.



When data are normally distributed, the mean is the central peak of the distribution. This is calculated by adding together all numbers in the sample and dividing it by the sample size.

However, when the sample is not normally distributed and the peak does not lie in the middle, extreme values or a longer tail will pull the mean towards it. This means that where data are not normally distributed, the mean will not be the centre and the value will be invalid. Where this is the case, the **median** should be used instead. The median is calculated by ordering the numeric values from smallest to largest and selecting the middle value.

When data are normally distributed, the mean and median will give the same, or very similar, values. This is because both are measuring the centre. However, when the data are skewed, the mean and median will differ. We prefer to use the mean where possible as it is the more powerful measure. This means that it uses more of the data than the median and is therefore more sensitive to changes in the sample.

5.2.2.2 Measures of spread

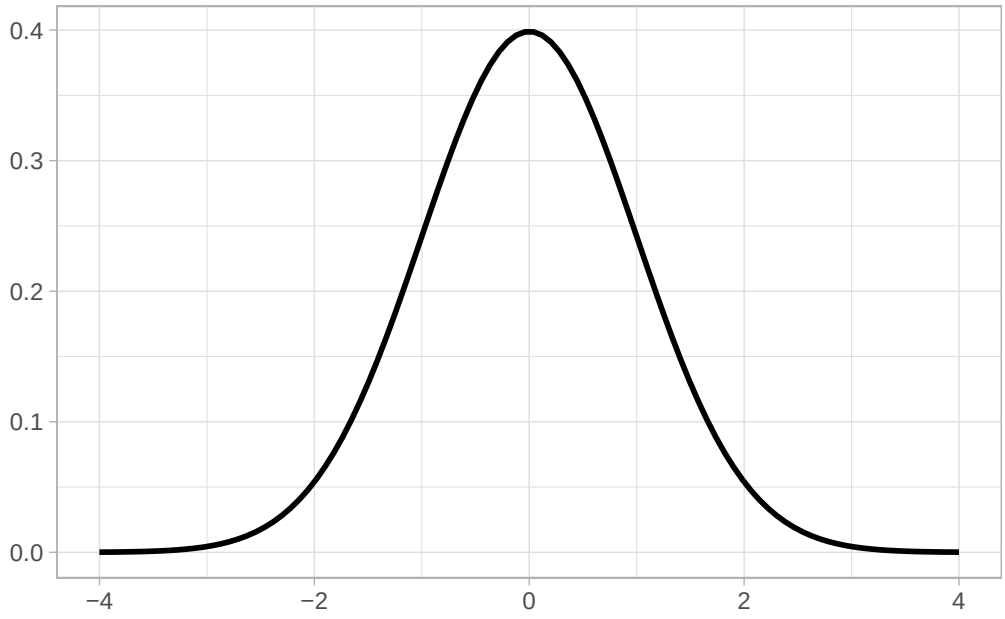
Generally the measure of the spread of a numeric variable is presented with a measure of spread, or how wide/narrow the distribution is. As with the apread, the most appropriate values will depend on whether the sample is normally distributed or not.

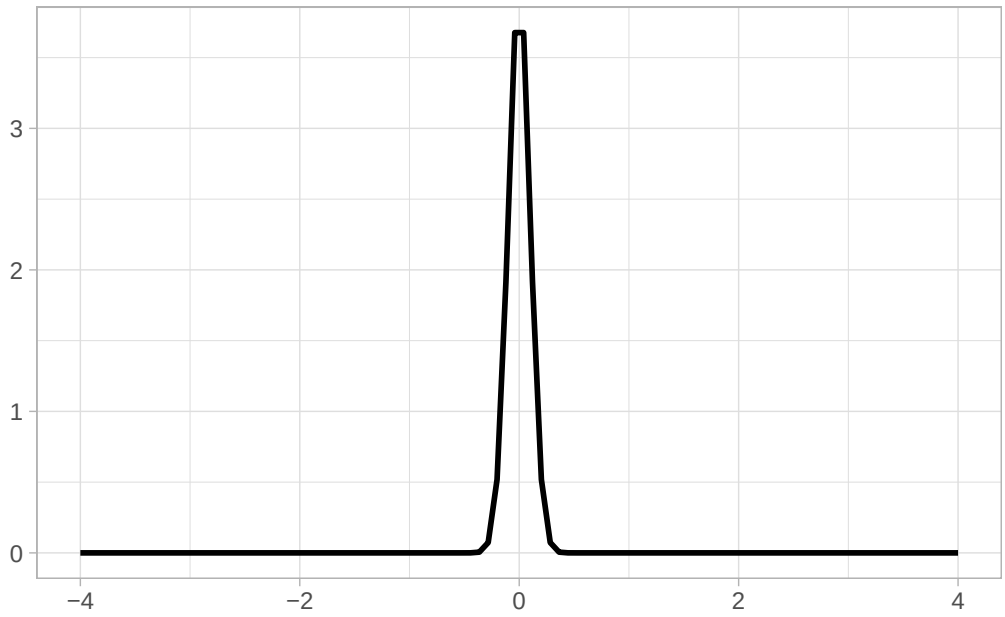
The most simple measure of spread is the **range** of a sample. In R, this is given as two values: the minimum and the maximum.

The issue with using the range is that it is entirely defined by the most extreme values in the sample and does not give any information about the rest of it. An alternative to this would be to give the range of the middle 50%, also known as the **interquartile range** (IQR).

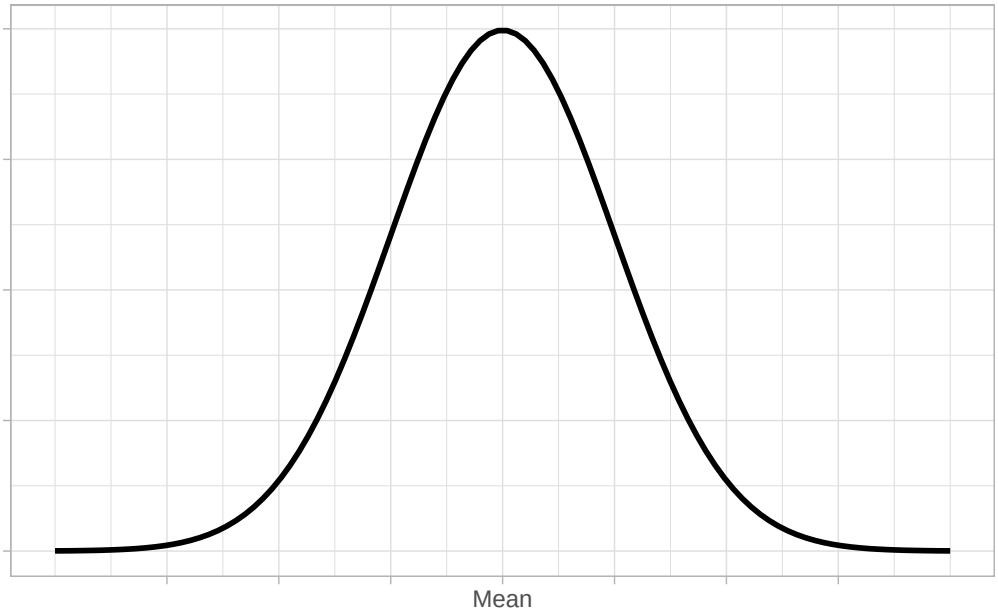
The IQR is the difference between the 75th percentile, or upper quartile, and the 25th percentile, or lower quartile. As with the median, this is calculated by ordering the sample from smallest to largest. The sample is then cut into 4 and the quartiles are calculated. In R, the IQR is given as the difference between the upper and lower quartiles. To calculate these values separately, we can use the `quantile` function.

Both the range and IQR only use 2 values from the sample. As with the median, these measures discard a lot of information from the summaries. Where the sample is normally distributed, the **standard deviation** (SD) can be used which measures the average distance between each observation and the mean. The larger the SD, the wider and flatter the normal curve will be; the smaller the SD, the narrower and taller the curve will be:





The standard deviation is only appropriate where a numeric variable has a normal distribution, otherwise this value is meaningless. If a sample is normally distributed, then the entire sample can be completely described just using the mean and standard deviation, even when the sample values are not given. As the distribution is symmetrical, the mean and standard deviation can be used to estimate ranges of values. For example, it is known that approximately 68% of a sample will lie one standard deviation from the mean, approximately 95% within 2 standard deviations from the mean, and around 99.7% within 3 standard deviations:



This knowledge can also be used to check the mean and standard deviation were appropriate summary statistics, even if we have no other information.

5.3 Exercise 4

1. How many responders had both weekly rent and mortgage payments given? What are the potential reasons for this?
2. Combine the weekly rent and mortgage variables into a single weekly payment variable.
3. Create a summary table containing the mean, median, standard deviation, and the upper and lower quartiles of the weekly payment (rent and mortgage combined) for each region. What, if anything, can you infer about the distribution of this variable based on the table?

6 Loading and tidying Excel data

In the first part of the course, we saw how SPSS data (`.sav`) can be loaded into R using the `haven` package. Another common format of data that cannot be loaded into base R is excel (`.xlsx`). The package required to read Excel data is the `readxl` package.

If this is the first time using the `readxl` package, remember to install this to your machine using the `install.packages` function:

```
install.packages("readxl")
```

Once this package has been installed, add it to the list of packages to install at the top of your script file:

```
pacman::p_load(tidyverse, haven, readxl)
```

6.1 Loading an Excel sheet into R

As with any file format, we must ensure data are in the correct form before loading them into R, ensuring each column represents a variable, each row represents an observation, and there are no tables or graphics.

Excel files can be a little tricker to manipulate than SPSS and CSV files as they often contain multiple sheets. This is the case for the data that we will be using for this part of the course.

The Excel file that we will be loading contains the Office for Budget Responsibility (OBR) economic and fiscal outlook. This contains many sheets of data, but for this course we will just be focusing on three:

- 1.6 Labour market
- 1.14 National living wage
- 1.17 Housing market

Let's begin with housing market data, stored in the 18th sheet, labelled "1.17". This sheet can be selected in the `read_xlsx` function using the `sheet` argument.

The housing market sheet shows information over different time scales: first by quarters, then years, and then across pairs of years. For this example, we will extract information measured quarterly (rows 4 - 88). The argument `range` allows us to define the range of cells (by columns and rows) to extract.

Finally, we can see that the column headers are not in an appropriate format for R: they contain spaces, brackets, and are very long! There are two approaches we will consider to overcome this.

The first is to remove the column names completely (by not including them in the `range` argument and setting `col_names = FALSE` within the `read_xlsx` function) and add them manually, using the `setNames` function.

Setting names manually can take a long time and a lot of typing if there are many variables. An alternative to this manual approach is to include them in the `range` of the `read_xlsx` function, and use an R function to 'clean' them, making them follow the style guide.

> Style tip
>
> The `janitor` package has been designed to format inputed data to ensure it follows the Tidyverse style guide. The `clean_names` function can be applied to a data frame or tibble to adapt variable names in this way.

The following code loads the housing market sheet and manually sets the variable names:

```
# Return file names from the data folder
list.files(path = "data")
```

```
[1] "Detailed_forecast_tables_Economy_March_2024.xlsx"
[2] "generalfs21_EUL.sav"
[3] "interviewfs21_EUL.sav"
```

```
housing_market <-
  read_xlsx("data/Detailed_forecast_tables_Economy_March_2024.xlsx",
                            # Specify the sheet and range of cells to keep
                            sheet = "1.17",range = "B4:J88",
                            # Remove column names (too messy)
                            col_names = FALSE) %>%
  # Add variable names manually
  setNames(c("period", "hpi_comp15", "hpi_prev_year",
             "residential_property_transactions",
```

```
              "private_enterprise_housing_starts",
              "private_enterprise_housing_comp",
              "housing_stock", "net_additions_housing_stock",
              "turnover_rate"))
```

The following code loads the same data but uses the `janitor` package.

> **Warning**
>
> Do not run this code without installing and loading the `janitor` package first. We will not run this during the course, but it is included for future reference.

```
housing_market_alt <-
  read_xlsx("data/Detailed_forecast_tables_Economy_March_2024.xlsx",
                                # Specify the sheet and range of cells to keep
                                sheet = "1.17",range = "B3:J88") %>%
  # Removes spaces, special characters, all lower case, etc.
  clean_names()
```

6.2 Splitting variables

In the current dataset, the time variable is given as a character and so is not recognised as ordered or temporal by R. To overcome this, we can split the variable to create separate year and quarter variables.

The `str_sub` function from tidyverse's `stringr` package extracts elements based on their position in a string of characters. This can be used to return the first 4 digits to a new `year` variable, and the final digit to a new `quarter` variable:

```
housing_market <- housing_market %>%
  # Don't forget to convert the string to numberic
  mutate(year =  as.numeric(str_sub(period, start = 1, end = 4)),
         quarter = as.numeric(str_sub(period,
                              # Use - to work from the end of the string
                              start = -1L, end = -1L))) %>%
  # Remove the original period variable
  select(-period)
```

6.3 Exercise 5

1. Load in the OBR's quarterly labour market data (sheet 1.6), keeping the following variables:

- Period
- Employment rate (%)
- Average earning growth (%)
- Average earning index
- Productivity per hour index
- Real wage product
- Real consumption wage

Split the period data into separate year and quarter variables, ensure that all variable names follow Tidyverse's style guide. Name this object `labour_market`.

2. Load the OBR's national living wage data (sheet 1.14), keep as an object named `living_wage`.

6.4 Tranforming data

The labour market and housing market data are currently considered in what is known as **long format**, with many rows and fewer variables. The alternative to this format, **wide format**, can be seen in the living wage data, which has many variables and very few (only one!) row. Sometimes we may wish to convert between these variables, either to join them to other datasets (as is the case here), or to carry out an analysis or visualisation that requires a certain format. These conversions are carried out using the `pivot_longer` or `pivot_wider` functions.

There are many ways to pivot data within R (see the helpfile `?pivot_longer` for a full list of arguments), and the setup of this function tends to differ for every situation. For worked examples and a more detailed explanation of the function's capabilities, enter `vignette("pivot")` into the console. For our data, we will need to convert the wide-format living wage data to a long-format so we are able to join it to the other data. This will create a new dataset with 2 variables: year and living wage, with a row per year. The `year` variable will be taken from the wide data names, and the `living_wage` variable will come from the wide data values:

```
living_wage_long <- living_wage %>%
  # First, select the columns that we wish to pivot (all of them)
  pivot_longer(cols = everything(),
               # Move the old variable names to a new year variable
               names_to = "year",
```

```
                # Remove the prefix from the old variable names
                names_prefix = "year_",
                # Convert the new year variable to numeric
                names_transform = as.numeric,
                # Take the old values and create a new living_wage variable
                values_to = "living_wage")
```

6.5 Exercise 6

1. Combine all three OBR datasets (housing market, labour market and living wage) together to create one complete dataset, `obr_data`.

7 Data visualisation

Data visualisation is a powerful tool with many important uses. First, visualisations allow us to explore the data, identify potential outliers and errors, or check that the variables behave in the way we would expect them to if they had been recorded correctly. Visualisations can also be used as an analysis tool, allowing us to identify trends in the data or differences between groups. Finally, visualisations can help to convey messages to an audience in a clear, concise way that is often more powerful than presenting them using numbers or text. In some cases, data visualisations can show results so clearly that further analysis is arguably unnecessary.

7.1 Choosing the most appropriate visualisation

The most appropriate choice of visualisation will depend on the type of variable(s) we wish to display, the number of variables and the message we are trying to disseminate. Common plots used to display combinations of different types of data are given in following table:

Number of variables	Type of variables	Visualisation
One variable	Categorical	Frequency table
		Bar chart
	Numerical	Histogram
	Spatial	Map
	Temporal	Line plot
Two variables	Two categorical	Frequency table
		Stacked/side-by-side bar chart
	One numeric, one categorical	Dot plot
		Box plot
	Two numerical	Scatterplot
	> 2 categorical	Table

Number of variables	Type of variables	Visualisation
> 2 variables	2 numeric, one categorical or > 2 numeric	Scatterplot with different colours/symbols/sizes

R is very flexible when it comes to visualising data and contains a wide variety of options to customise graphs. This section will focus on the Tidyverse package `ggplot2` and introduce some of the more commonly used graphical functions and parameters.

7.2 The ggplot2 package

The `ggplot2` package implements the 'grammar of graphics', a system that aims to describe all statistical graphics in terms of their components or layers. All graphics can be broken down into the same components: the data, a coordinate system (or plot area) and some visual markings of the data. More complex plots may have additional layers but all must contain these three.

For example, if we want to investigate the distribution of tenure types between responses of the English Housing Survey (EHS), we could use a **bar chart**. The visual markings for a bar chart is a bar per group (in this case, tenure type), where the length of each bar represents the number of observations within that group.

For any visualisation created using `ggplot2`, we first use the `ggplot` function to create a coordinate system (a blank plot space) that we can add layers and objects to. Within this function, we specify the data that we wish to display on the coordinate system:

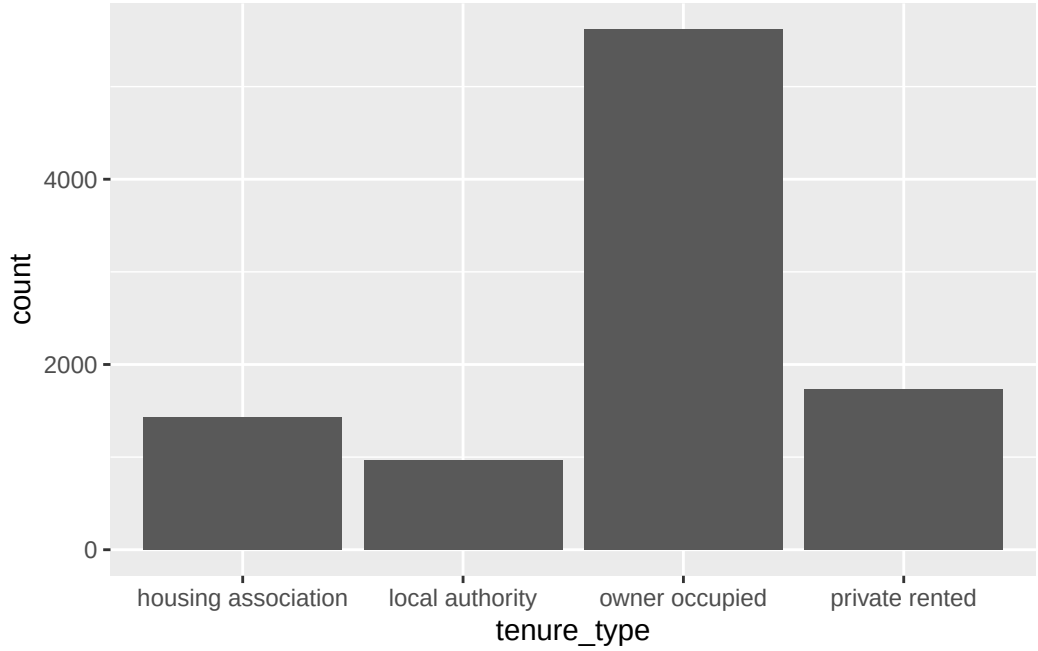
```
ggplot(data = ehs_tidy)
```

To add information to this graph, we add a **geom** layer: a visual representation of the data. There are many different geom objects built into the ggplot2 package (begin typing `?geom` into the console to see a list). The `geom_bar` function is used to create bar charts

Each geom object must contain a mapping argument, coupled with the `aes` function which defines how the variables in the dataset are visualised. In this case, we use the `aes` function to specify the grouping variable on the x axis, but it can also be used to set the colour, size or symbol based on variable values.

> **Warning**
>
> Although `ggplot2` is part of the tidverse package, it uses a `+` symbol to add layers to visualisations rather than the pipe `%>%` we have been using in other packages.

```
ggplot(data = ehs_tidy) +
  geom_bar(mapping = aes(x = tenure_type))
```



Graphs appear in the plot window in RStudio and can be opened in a new window using the Zoom icon. Graphs in this window can also be copied and pasted into other documents using the Export icon and selecting *Copy to clipboard*.

New graphs will replace existing ones in this window but all graphs created in the current session of R can be explored using the icons.

Graphs can be stored as objects using the `<-` symbol. These objects can then be saved as picture or PDF files using the `ggsave` function:

```
tenure_bar <- ggplot(data = ehs_tidy) +
  geom_line(aes(x = tenure_type))

ggsave(tenure_bar, filename = "tenure_bar.png")
```

7.3 Exercise 7

1. Choose an appropriate visualisation to check the distribution of the gross income variable from respondents of the English Housing survey. Comment on your findings.
2. Based on the output from question 1, generate a summary table giving the minimum, maximum gross income, and an appropriate measure of the centre and spread of this variable.

7.4 Customising visualisations

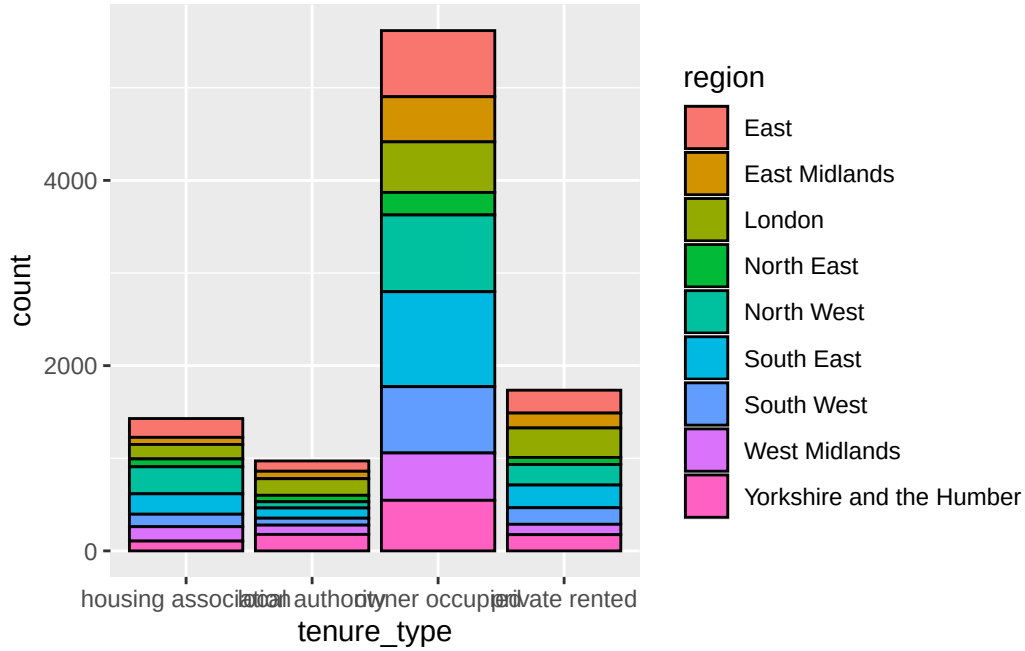
Visual markings of a `ggplot` object can be customised as part of the `geom` function. Arguments that can be adjusted within these geoms include:

- `colour`: change the colour (if point or line) or outline (if bar or histogram) of the geom
- `size`: change the size of the markings (if point used)
- `shape`: change the shape of markings (for points)
- `fill`: Change the colour of bars in bar charts or histograms
- `linewidth`: Change the line width
- `linetype`: Choose the type of line (e.g. `dotted`)
- `alpha`: Change the transparency of a visualisation

These options can be used to add variables to a visualisation. For example, the distribution of tenure types could be compared between regions by changing the `fill` of these bars, converting the bar chart into a **stacked bar chart**. When these options are determined by a variable in the data, they should be added inside the `aes` wrapper. Options can also be adjusted manually when the arguments are added outside of the `aes` wrapper.

To convert the previous bar chart into a stacked bar chart, we define the `fill` by the `region` variable. To make these distinctions easier to see, we can also add a black outline to the bars by manually setting `colour`:

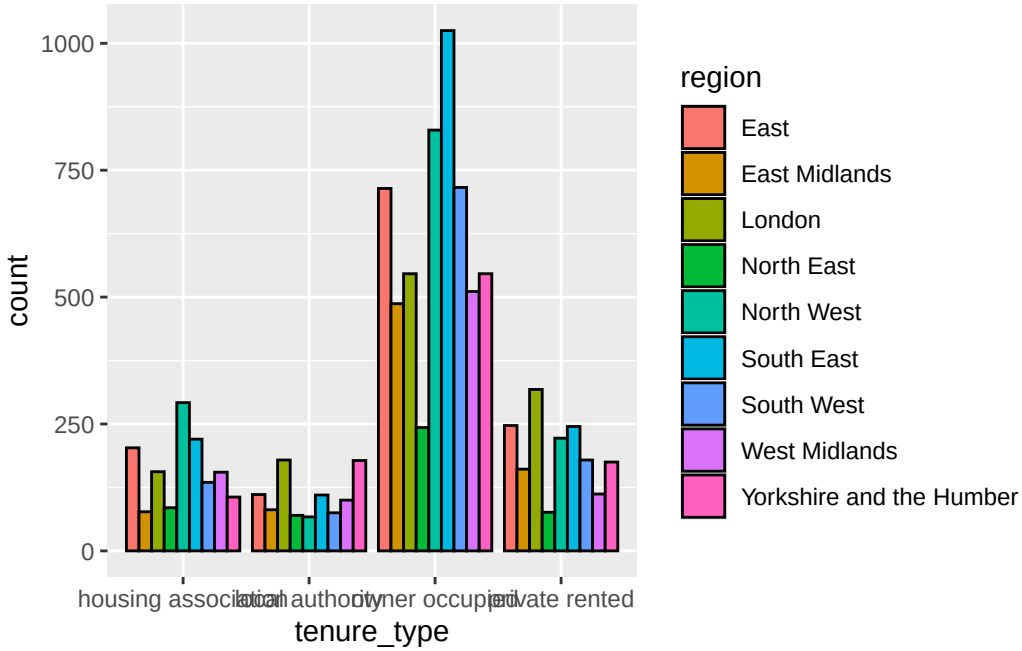
```
ggplot(data = ehs_tidy) +
  # Define the x axis and fill inside aes
  geom_bar(aes(x = tenure_type, fill = region),
           # Manually define colour outside aes
           colour = "black")
```



> **Style tip**
>
> R contains a list of 657 pre-programmed colours that can be used to create palettes (run `colours()` in the console for a full list). Hexadecimal codes can also be included instead in the form `#rrggbb` (where rr (red), gg (green), and bb (blue) are numbers between 00 and 99 giving the level of intensity of each colour).

Each geom has different arguments that can be customised to adapt visualisations. For example, `geom_bar` has the `position` argument which controls how additional groups are displayed. By default, this argument is set to `"stack"` which created a stacked bar chart as we saw in the last example. An alternative would be to set this to `position = "dodge"` which creates a **side-by-side bar chart**. Here, the tenure type bars are separated into smaller bars per region, but are displayed next to one another, rather than on top of each other:

```
ggplot(data = ehs_tidy) +
  # Define the x axis and fill inside aes
  geom_bar(aes(x = tenure_type, fill = region),
           # Manually define colour outside aes
           colour = "black",
           # Show bars side-by-side instead of stacked
           position = "dodge")
```



For a more comprehensive list of the options available for the geom you are interested in, check the helpfile (e.g. `?geom_bar`).

> **Warning**
>
> Although it may be tempting to add many variables to the same visualisation, be sure that you are not overcomplicating the graph and losing important messages. It is better to have multiple clear (but simpler) visualisations than fewer confusing ones.

7.5 Exercise 8

Choose an appropriate visualisation to investigate the change in employment rate between 2008 and 2024. Generate this visualisation and comment on your findings.

7.6 Scale functions

Scale functions allow us to customise aesthetics defined in geom objects, such as colours and axes labels. They take the form `scale_'aesthetic to customise'_'scale of variable'`.

7.6.1 Customising axes

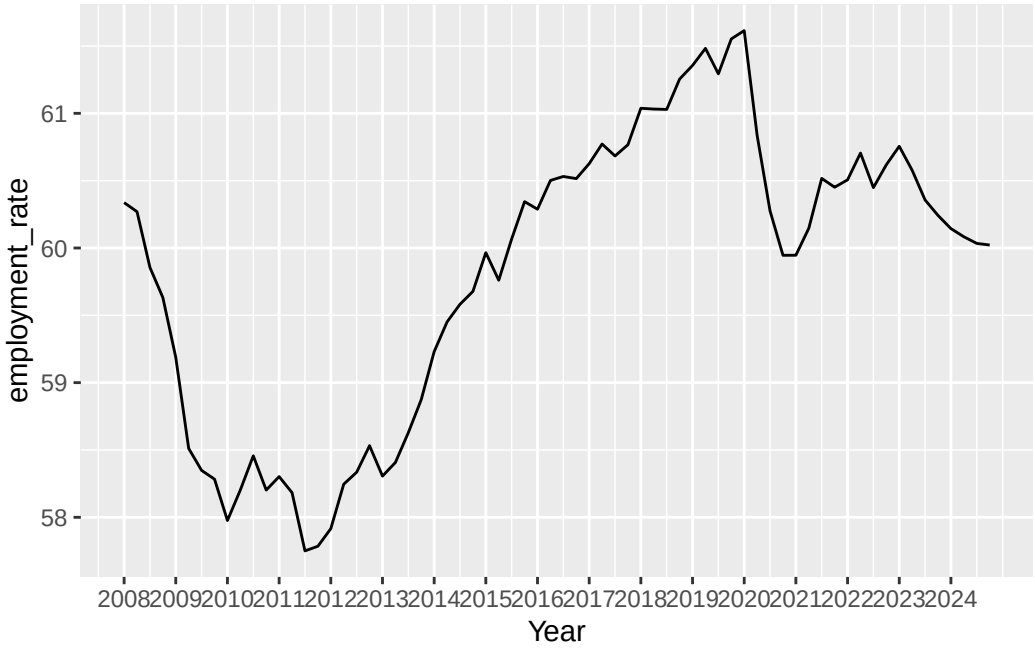
Scale functions can be used to customise axis titles, limits, breaks, and labels. The choice of `scale` function is determined by the type of variable displayed on the axis.

For example, if we wanted to customise the x axis of the line graph generated in the previous exercise, showing the employment rate over time, we would use the `scale_x_continuous` function. Arguments to customise the x or y axes include:

- `name =` to change the axis title
- `limits = c(...)` sets the axis limits
- `breaks = c(...)` defines tick marks
- `labels = c(...)` attaches labels to break values
- `trans =` transforms the scale that the axis is shown on.

In this example, we can add labels to the x axis that shows which year the time variable represents, making it easier to interpret:

```
ggplot(data = obr_data) +
  geom_line(aes(x = time, y = employment_rate)) +
  # Add axis title
  scale_x_continuous(name = "Year",
                     # Add breaks for each year
                     breaks = seq(1, 65, by = 4),
                     # Add labels to breaks
                     labels = 2008:2024)
```



7.6.2 Customising colour scales

There are a wide range of options for customising the colour aesthetics of geoms. These include pre-defined colour palettes, such as `scale_colour_viridis_c` for continuous variables, or `scale_colour_viridis_d` for discrete or categorical variables. Viridis colour palettes are designed to be colourblind friendly and print well in grey scale. There are also many R packages containing colour palettes for different scenarios.

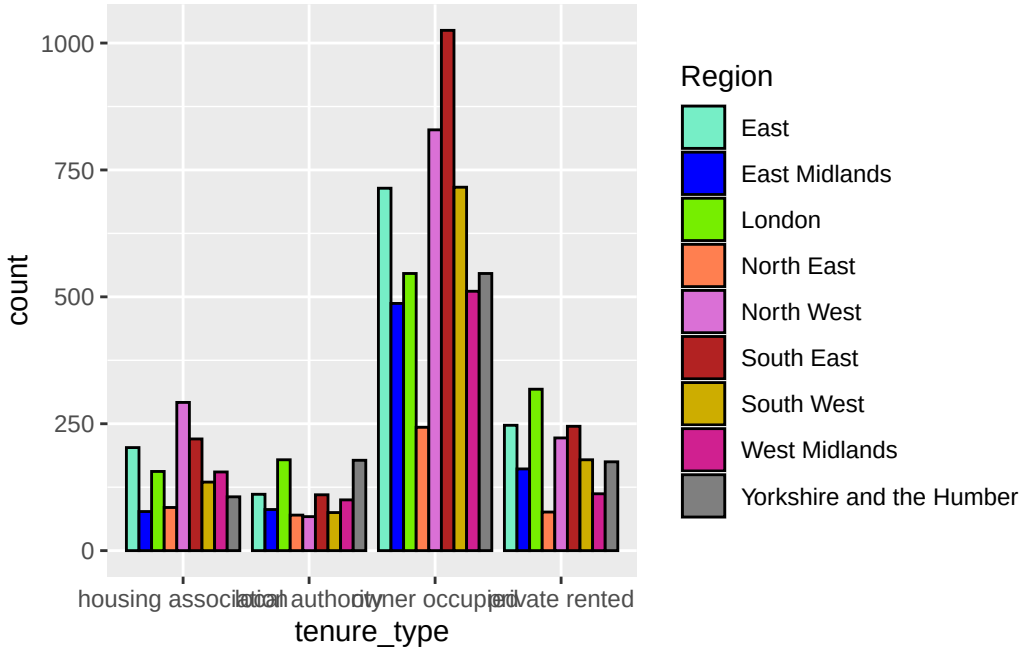
Colour palettes can be created manually for categorical variables using the `scale_colour_manual` function. Here, the argument `values` allows us to specify a colour per category.

Where a colour palette will be used across multiple plots, defining this list of colours as a vector and then entering this into `scale_fill_manual` will reduce repetition. For example, where region is used to group across multiple plots, it will be useful to create a region colour palette:

```
region_palette <- c("aquamarine2", "blue", "chartreuse2", "coral", "orchid",
                    "firebrick", "gold3", "violetred", "grey50")

ggplot(data = ehs_tidy) +
  # Define the x axis and fill inside aes
  geom_bar(aes(x = tenure_type, fill = region),
           # Manually define colour outside aes
```

```
          colour = "black",
          # Show bars side-by-side instead of stacked
          position = "dodge") +
  # Change legend title and add colour values
  scale_fill_manual(name = "Region", values = region_palette)
```



Palettes can also be created using gradients with the `scale_colour_gradient` function, that specifies a two colour gradient from low to high, `scale_colour_gradient2` that creates a diverging gradient using low, medium, and high colours, and `scale_colour_gradientn` that creates an n-colour gradient.

7.7 Other labelling functions

Although axis and legend labels can be updated within scale functions, the `labs` function exist as an alternative. This function also allows us to add titles and subtitles to visualisations:

```
labs(x = "x-axis name", y = "y-axis name",
    colour = "Grouping variable name", title = "Main title",
    subtitle = "Subtitle", caption = "Footnote")
```

The `annotate` function allows us to add text and other objects to a ggplot object. For example:

```
annotate("text", x = 50, y = 200, label = "Text label here")
```

Adds "Text label here" to a plot at the coordinates (50, 200) on a graph, and

```
annotate("rect", xmin = 0, xmax = 10, ymin = 20, ymax = 50, alpha = 0.2)
```

adds a rectangle to the graph.

7.8 Theme functions

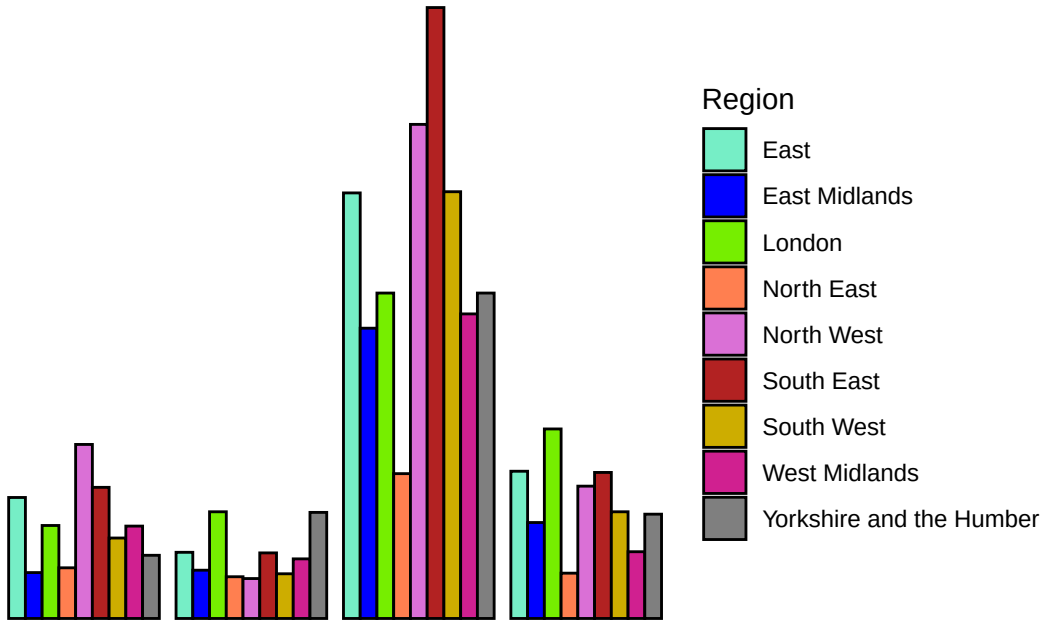
The `theme` function modifies non-data components of the visualisation. For example, the legend position, label fonts, the graph background, and gridlines. There are many options that exist within the `theme` function (use `?theme` to list them all).

> **Note**
>
> Many of the elements that can be customised within the `theme` function require an `element` wrapper. This wrapper is determined by the type of object we are customising (e.g. `element_text` when customising text, `element_rect` when customising a background, `element_blank` to remove something). Check `?theme` for more information.

One of the most common theme options is `legend.position` which can be used to move the legend to the top or bottom of the graph space (`legend.position = "top"` or `legend.position = "bottom"`) or remove the legend completely (`legend.position = "none"`).

`ggplot` also contains a number of pre-defined themes which change non-data elements of the plot to a programmed default. For example `theme_void` removes all gridlines and axes, `theme_light` changes the graph background white and the gridlines and axes light grey:

```
ggplot(data = ehs_tidy) +
  # Define the x axis and fill inside aes
  geom_bar(aes(x = tenure_type, fill = region),
           # Manually define colour outside aes
           colour = "black",
           # Show bars side-by-side instead of stacked
           position = "dodge") +
  # Change legend title and add colour values
  scale_fill_manual(name = "Region", values = region_palette) +
  # Remove gridlines and axes (not recommended!!)
  theme_void()
```



One benefit of using themes is that all visualisations will be consistent in terms of colour scheme, font size and gridlines. Although there are pre-built themes, we are able to create our own and save them as functions. These can then be used in place of R's themes.

7.8.1 Creating functions

To create our own function in R, we first give it a name and attach `function()` followed by curly brackets `{}`, with the function defined inside those brackets.

For example, to create our own theme function, called `theme_dluch`, which sets the title font size to 15, the axis titles to size 12, the axis ticks to size 10, moves the legend to the bottom of the graph, and changes the background colours, we use the following:

```
theme_dluch <- function() {
  # Change plot title size
  theme(plot.title = element_text(size = 15),
        # Change axis title size
        axis.title = element_text(size = 12),
        # Change axis text size
        axis.text = element_text(size = 12),
        # Move legend to bottom of graph
        legend.position = "bottom",
        # Change background
```

```
        plot.background = element_rect(fill = "grey75"),
        # Change graph area background
        panel.background = element_rect(fill = "white",
                                        colour = "black"))
}
```

The function `theme_dluch` will now appear in the Environment window and can be added to `ggplot` objects:

```
ggplot(data = ehs_tidy) +
  # Define the x axis and fill inside aes
  geom_bar(aes(x = tenure_type, fill = region),
           # Manually define colour outside aes
           colour = "black",
           # Show bars side-by-side instead of stacked
           position = "dodge") +
  # Change legend title and add colour values
  scale_fill_manual(name = "Region", values = region_palette) +
  labs(x = "Tenure type", y = "Count") +
  # Remove gridlines and axes (not recommended!!)
  theme_dluch()
```

Creating a personalised theme ensures that visualisations are consistent, whilst keeping code concise and reducing repetition.

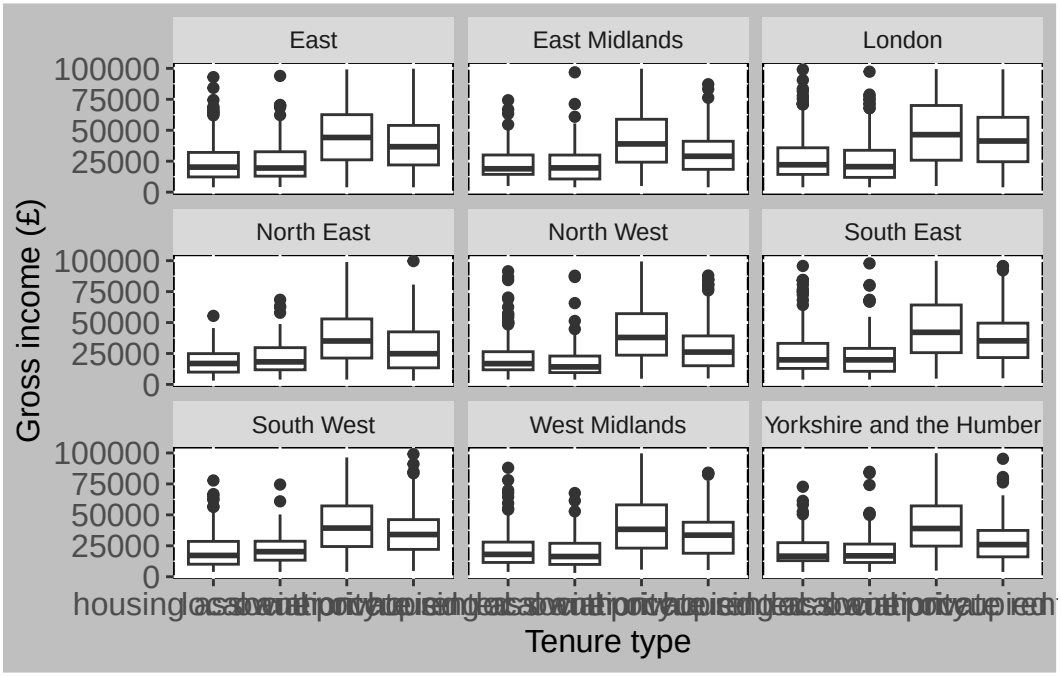
7.9 Facet functions

Faceting allows us to divide a plot into subplots based on some grouping variable within the data. This allows us to show multiple variables in the same visualisation without risking overloading the plot and losing the intended message.

For example, we could compare the relationship between gross income and tenure type (shown using a boxplot) between regions by faceting the graph by region using the `facet_wrap` function:

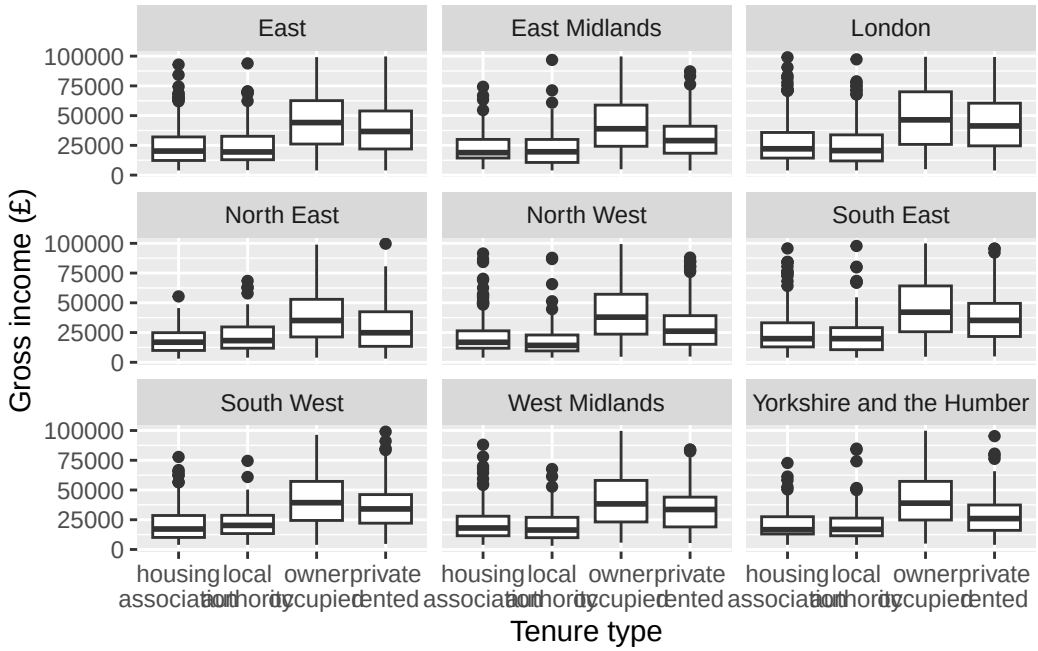
> **Warning**
>
> Remember that the value 100,000 actually represents anyone earning £100,000 or more. To avoid skewing the data, we will remove these values and investigate trends below this threshold.

```
ehs_tidy %>%
  # Remove gross income >= £100,000
  filter(gross_income != 100000) %>%
  # Do not need to specify data, it is already passed through the pipes
  ggplot() +
  geom_boxplot(aes(x = tenure_type, y = gross_income)) +
  labs(x = "Tenure type", y = "Gross income (£)") +
  facet_wrap( ~ region) +
  theme_dluch()
```



Facetting has made the tenure type labels unreadable. To overcome this, we can 'wrap' the long labels using the `label_wrap_gen` function and setting an upper limit on the width of labels:

```
ehs_tidy %>%
  # Remove gross income >= £100,000
  filter(gross_income != 100000) %>%
  # Do not need to specify data, it is already passed through the pipes
  ggplot() +
  geom_boxplot(aes(x = tenure_type, y = gross_income)) +
  scale_x_discrete(labels = label_wrap_gen(12)) +
  labs(x = "Tenure type", y = "Gross income (£)") +
  facet_wrap( ~ region)
```



7.10 Exercise 9

Use an appropriate visualisation to investigate the relationship between house prices and wages between 2008 and now. Ensure that the variables you choose are comparable and treat this as though the final product will be exported into a report (make sure it is clear and looks good!). Interpret your final graph.