Introduction to quantitative analysis with R

Sophie Lee

Table of contents

Welcome!

Welcome to the Introduction to Quantitative Analysis with R course, designed for and with the Ministry of Housing, Communities and Local Government (MHCLG). 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.

How to use these materials

This e-book provides a combination of written explanations, code examples, and practical exercises to allow you to practice what you have learned.

```
1 + 1
```

Code within these blocks can be copied and pasted into your R session to save time when coding (I recommend typing the code yourself to familiarise yourself with the coding process and use the copy option if you are really stuck!).

Throughout the materials, you will see colour-coded boxes which are used to highlight important points, give warnings, or give tips such as keyboard shortcuts.

> **Note**
>
> These boxes will be used to highlight important messages, supplementing the main text.

Hint

These boxes will contain useful hints, such as keyboard shortcuts, that can make your coding life a little easier!

Style tip

These boxes contain style tips to ensure that your code follows the Tidyverse style guide, making it as consistent and readable as possible.

Warning

These boxes will contain warnings and highlight areas where you need to be more cautious in your coding or analysis.

To make these notes as accessible as possible, they are available to view in dark mode by toggling the  button. They are also available to download as a PDF file using the  button.

All exercise solutions are available in the appendices. Please attempt the exercises yourself first, making full use of R's built in help files, cheatsheets and example R code in this book. Going straight to the solutions to copy and paste the code without thinking will not help you after the course!

Some exercises contain expandable hints, such as functions required to complete them, that can be viewed when needed. For example:

Exercise hint

The functions you will need for this exercise are `filter` and `count`.

Data for the course

This course uses data from the English Housing Survey (EHS) from 2021 and the Office for Budget Responsibility (OBR) 2024 economic and fiscal outlook. All data used in the course can be downloaded from the course repository.

Before beginning the course, please save these files into a folder called 'data' within the folder you will be working from during the course.

Part I

Introduction to R and RStudio

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:

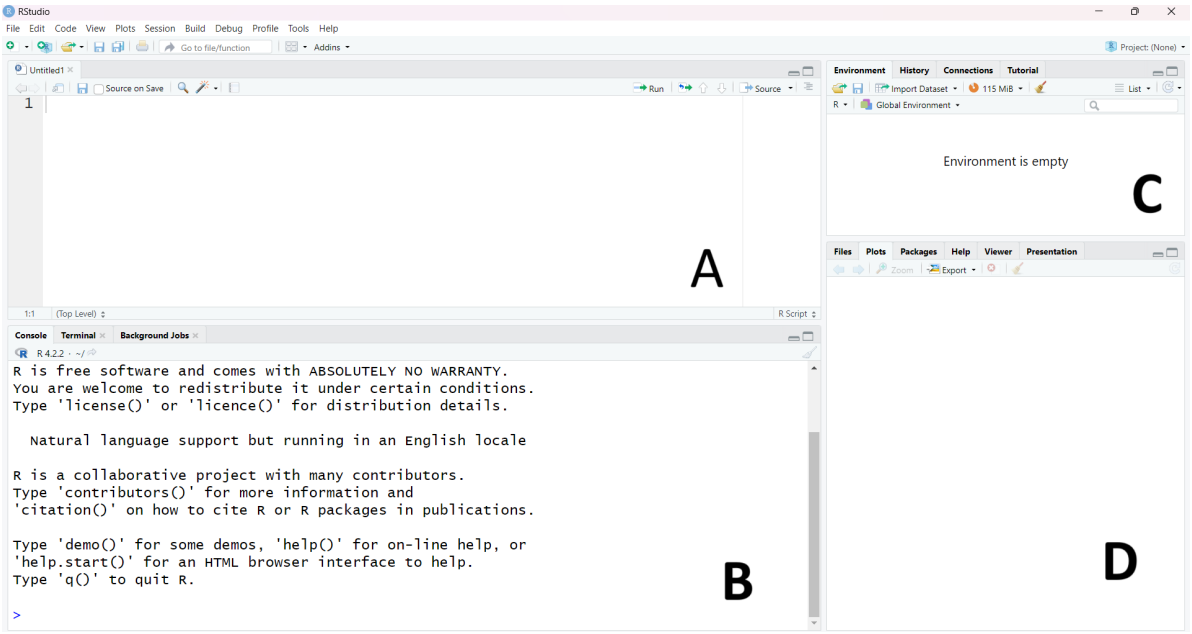


Figure 1.1: RStudio console window

1.1.1 Window A: R script files

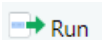
All analysis and actions in R are carried out using the R syntax language. R script files allow us to write and edit code before running it in the console window.

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.

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 are 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 ----------
```

> 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 spaces 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
```

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 sums:

a. 64^2
b. $3432 \div 8$
c. 96×72

For each part of question 2, copy the result from the console and paste them onto the same line of the script file as the code. Do this in a way that ensures there are no error messages if you were to run the entire script file.

> **Hint**
>
> To run a script file from beginning to end, select all code using `ctrl + a`/`Cmnd + a`, then run the selected code as usual.

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). They 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 Section 1.1.3).

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 Section 1.1.4.

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

> **Hint**
>
> If you know the arguments required in a function are different to default settings, but are not totally sure the settings you require, consider running example code from the helpfile in the console and seeing how these arguments change the process.

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')
```

Part II

Tidyverse and data wrangling

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-separated 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 respondents' 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:

```r
names(ehs_general)                                                    ①

head(ehs_general)                                                     ②

tail(ehs_general)                                                     ③

str(ehs_general)                                                      ④
```

① Return variable names
② Returns the first 6 rows
③ Returns the last 6 rows
④ Gives information about the structure of an object (including variable types)

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

> Hint
>
> To remove any of the attributes imported from an SPSS file, we can use `haven` packages `zap` functions. For example, `zap_widths(x)` would remove the width attributes from an object `x`, and `zap_labels(x)` would remove value labels. For more information, see this article.

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`):

```
select(ehs_general, serialanon, aagfh21, tenure4x, gorehs)                    ①

select(ehs_general, 1, 2, 5, 7)                                               ②
```

① Variables can be selected using their names
② Or their column number

```
# 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
# 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 = "saved_data/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 values 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:

- `<=` less than or equal to,
- `<` less than,
- `>=` greater than or equal to,
- `>` greater than,
- `==` is equal to,
- `!=` is not equal to,

or can be based on conditional functions:

- `is.na(variable)` where variable is missing,
- `between(variable, a, b)` where variable 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 `,` where we want all conditions met, or the `|` in place of or. 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 tip

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.

> **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):

```r
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")))
```

> **Warning**
>
> The example converting our weighting variable into a categorical alternative is **purely for deomstrative purposes**. In reality, we would not treat a weighting variable this way.

> **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:

```r
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):

```r
vignette("dplyr")
```

Or look at the `dplyr` web page and cheatsheet.

4.9 A smooth process to the analysis dataset

Our EHS analysis dataset has been created haphazardly through this chapter to demonstrate 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:

```
ehs_general_tidy <- read_spss(file = "Data/generalfs21_EUL.sav") %>%          ①
                    mutate(tenure_type = as_factor(tenure4x),                 ②
                           region = as_factor(gorehs)) %>%
                    rename(id = serialanon,                                   ③
                           weighting = aagfh21) %>%
                    select(id, weighting, tenure_type, region)                ④

str(ehs_general_tidy)                                                         ⑤

write_csv(ehs_general_tidy, file = "saved_data/ehs_general_tidy.csv")         ⑥
```

① Step 1: load the dataset into R and attach as an object
② Step 2: convert grouping variables into factors
③ Step 3: rename other variables to avoid confusion
④ Step 4: keep only the necessary variables
⑤ Check the new data looks correct
⑥ Step 5: save this tidy data as a new file in a saved_data folder

```
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 o
```

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`

> Exercise hint
>
> The functions required for this exercise are `read_spss`, `str`, `mutate`, `as_factor`, `rename`.

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?

> Exercise hint
>
> This exercise requires you to create a subgroup which just contains observations that match the condition given (see Section 4.5), and `count` the number of rows in the subgroup.
> For the final part, use the `max` function to return the maximum recorded value. Check that this value actually represents an income.

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

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 tip

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`: a measure of centre when data are normally distributed
- `median`: a measure of centre, whatever the distribution
- `range`: the minimum and maximum values
- `min`: minimum
- `max`: maximum

- `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`: 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.

> **Warning**
>
> 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 results.

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 weighted data

The English Housing Survey is weighted to take account of the over-sampling of the less prevalent tenure groups and differential non-response, in order to provide unbiased national estimates. To account for this weighting (given by the `weighting` variable), we can use summary functions such as `wtd.mean` and `wtd.quantile` from the `Hmisc` R package:

```
library(Hmisc)

summarise(ehs_tidy,
          mean_income_unwtd = mean(gross_income),
          mean_income_wtd = wtd.mean(gross_income, weights = weighting),
          median_income_unwtd = median(gross_income),
          median_income_wtd = wtd.quantile(gross_income,
                                           weights = weighting,
                                           probs = .5))
```

```
# A tibble: 1 x 4
  mean_income_unwtd mean_income_wtd median_income_unwtd median_income_wtd
              <dbl>           <dbl>               <dbl>             <dbl>
1            40786.          41868.              34016.            35368.
```

The weighted total income is calculated by summing the product of each household income and its weight. As this variable does not currently exist within the dataset, we would need to create this first before summarising the sample:

```
library(Hmisc)

ehs_tidy %>%
  mutate(wtd_income = gross_income * weighting) %>%
  summarise(mean_income_unwtd = mean(gross_income),
            mean_income_wtd = wtd.mean(gross_income, weights = weighting),
            median_income_unwtd = median(gross_income),
            median_income_wtd = wtd.quantile(gross_income,
                                             weights = weighting,
                                             probs = .5),
            total_income_unwtd = sum(gross_income),
            total_income_wtd = sum(wtd_income))
```

```
# A tibble: 1 x 6
  mean_income_unwtd mean_income_wtd median_income_unwtd median_income_wtd
              <dbl>           <dbl>                <dbl>             <dbl>
1            40786.          41868.               34016.            35368.
# i 2 more variables: total_income_unwtd <dbl>, total_income_wtd <dbl>
```

5.2.2 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:

```
perc_tenure_type <- ehs_tidy %>%
  group_by(tenure_type) %>%                                          ①
  summarise(n_tenancy = sum(weighting/1000)) %>%                     ②
  ungroup() %>%                                                      ③
  mutate(n_responses = sum(n_tenancy),                               ④
         prop_tenure = n_tenancy / n_responses)

perc_tenure_type
```

① First group the data by tenure type
② Then count the number of households in each tenure type. Note that as this data are weighted, the total number of households is given by the sum of the weighting variable. This is divided by 1000 to present the value in 1000s of households.
③ Be sure to ungroup to remove this grouping structure!
④ Now calculate the total number of respondents overall and divide the group total by the overall total

```
# A tibble: 4 x 4
  tenure_type         n_tenancy n_responses prop_tenure
  <fct>                   <dbl>       <dbl>       <dbl>
1 housing association     2456.      24200.      0.101
2 local authority         1572.      24200.      0.0650
3 owner occupied         15562.      24200.      0.643
4 private rented          4611.      24200.      0.191
```

From this summary table, the proportion of responders that lived in privately rented properties was 0.191. To convert this into a percentage, we multiple the proportions by 100%:

```
perc_tenure_type <- perc_tenure_type %>%
  mutate(perc_tenure = round(prop_tenure * 100, 2)) %>%                    ①
  select(tenure_type, n_tenancy, perc_tenure)                              ②

perc_tenure_type
```

① Round the percentage to 2 decimal places to make it easier to interpret.
② Only keep the variables required for the table.

```
# A tibble: 4 x 3
  tenure_type         n_tenancy perc_tenure
  <fct>                   <dbl>       <dbl>
1 housing association     2456.        10.2
2 local authority         1572.         6.5
3 owner occupied         15562.        64.3
4 private rented          4611.        19.0
```

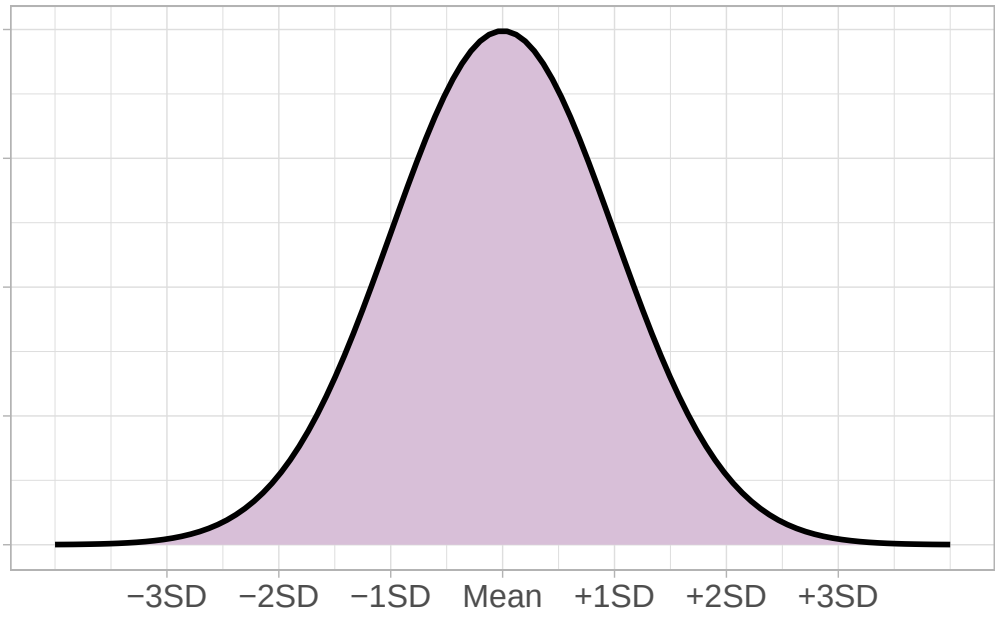
Therefore, 19.05% of responders lived in privately rented properties.

5.2.3 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.

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. Where data are not normally distributed, the mean will not be the centre and the value will be invalid. When 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.

Measures of spread

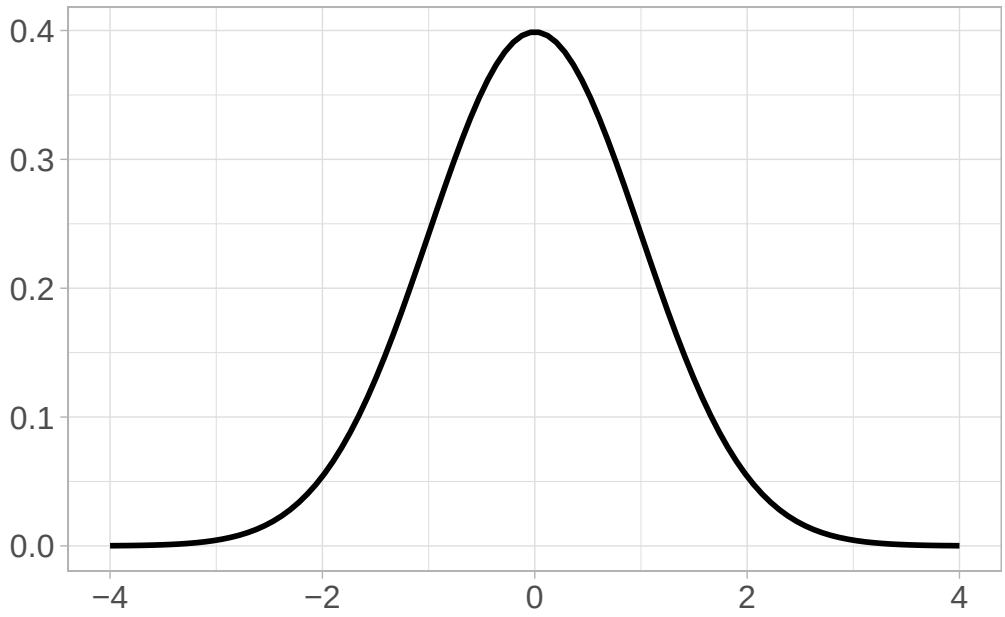
Generally the measure of the spread of a numeric variable is presented with a measure of spread, or how wide/narrow the distribution is. As with the apread, the most appropriate values will depend on whether the sample is normally distributed or not.

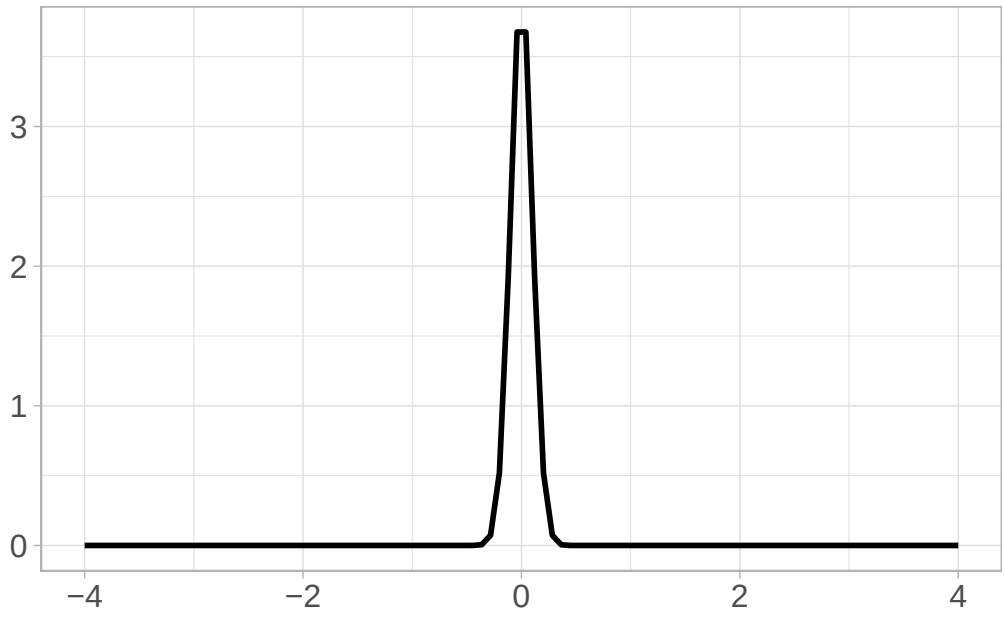
The most simple measure of spread is the **range** of a sample. In R, this is given as two values: the minimum and the maximum.

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:



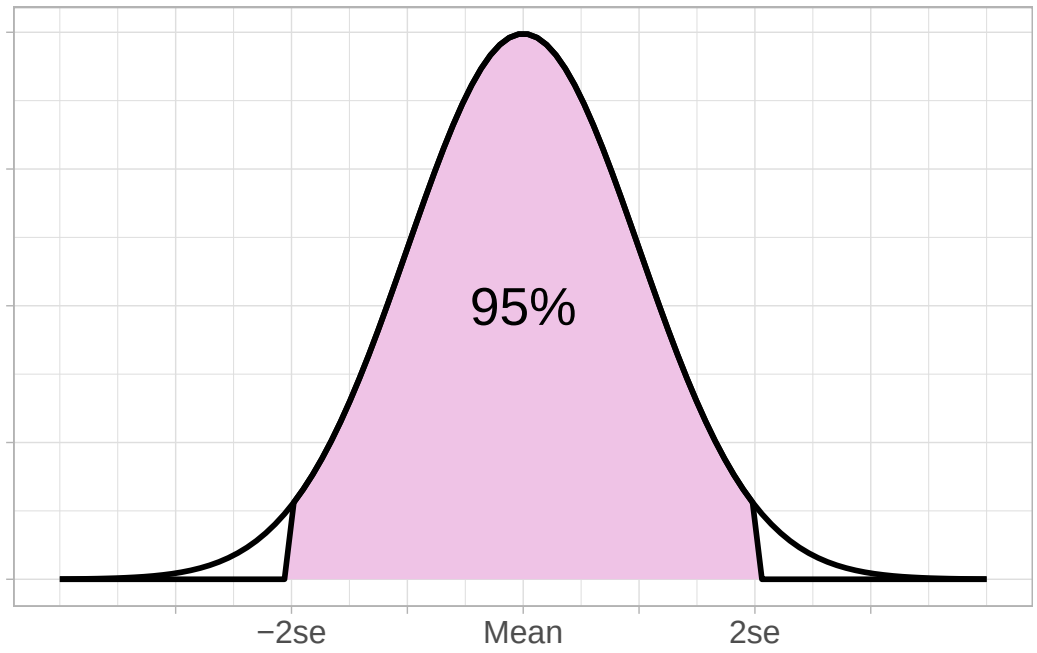


The standard deviation is only appropriate where a numeric variable has a normal distribution, otherwise this value is meaningless.

Properties of the normal distribution

If a sample is normally distributed, then it can be completely described 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.

Exercise 4

1. How many respondents had both weekly rent and mortgage payments given? What are the potential reasons for this?

> Exercise hint
>
> `filter` the data to return a subgroup of respondents with both weekly rent and mortgage payments, then `count` the number of rows in this subgroup.

2. Combine the weekly rent and mortgage variables into a single weekly payment variable.

> **Exercise hint**
>
> This exercise requires a new variable to be created that contains the sum of weekly rent and mortgage payments `if` both are observed, `else` it will take the observed value if just one is observed, or will remain missing if both are missing.
> You will need a function that creates variables based on `if` and `else` conditions, and one that will take the first non-NA value. Both of these can be found in the `dplyr` cheatsheet or as useful functions listed in the `mutate` help file.

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?

> **Exercise hint**
>
> Remember to estimate weighted summary statistics when dealing with the EHS data. Although there is not a weighted standard deviation function in the `Hmisc` package, consider that standard deviation is the square root of the variance.

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 trickier 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:

- 1.13 Household disposable income
- 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:

```
list.files(path = "data")                                                    ①

housing_market <-
  read_xlsx("data/Detailed_forecast_tables_Economy_March_2024.xlsx",
                            sheet = "1.17",range = "B4:J88",                 ②
                            col_names = FALSE) %>%                           ③
  setNames(c("period", "hpi_comp15", "hpi_prev_year",                        ④
             "residential_property_transactions",
             "private_enterprise_housing_starts",
             "private_enterprise_housing_comp",
             "housing_stock", "net_additions_housing_stock",
             "turnover_rate"))
```

① Return file names from the data folder
② Specify the sheet and range of cells to keep
③ Remove column names (too messy)
④ Add variable names manually

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

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") %>%
  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 %>%
  mutate(year =  as.numeric(str_sub(period, start = 1, end = 4)),          ①
         quarter = as.numeric(str_sub(period,
                              start = -1L, end = -1L))) %>%                ②
  select(-period)                                                          ③
```

① Don't forget to convert the string to numberic
② Use - to work from the end of the string
③ Remove the original period variable

Exercise 5

1. Load in the OBR's household disposable income data (sheet 1.13). Split the period data into separate year and quarter variables, ensure that all variable names follow Tidyverse's style guide. Name this object `disposable_income`.
2. Load the OBR's national living wage data (sheet 1.14), keep as an object named `living_wage`.

> Exercise hint
>
> The original variables take the name of the year they represent. However, variable names cannot begin with a number. Therefore, we must add a prefix to the original names. One way to add a prefix to a string is through the `paste0` function which combines arguments in the function separated by commas into a single string. For example,
>
> ```
> paste0("abc", 1, 10, "_2010")
> ```
>
> ```
> [1] "abc110_2010"
> ```
>
> Although the `rename` function requires each individual variable to be listed, look at the `dplyr` cheatsheet to find an alternative that renames variables using a function.

6.3 Transforming data

The disposable income 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 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:

```r
living_wage_long <- living_wage %>%
  pivot_longer(cols = everything(),                    ①
               names_to = "year",                      ②
               names_prefix = "year_",                 ③
               names_transform = as.numeric,           ④
               values_to = "living_wage")              ⑤
```

① First, select the columns that we wish to pivot (all of them). See the `?tidyr_tidy_select` help file for a list of options.
② Move the old variable names to a new year variable.
③ Remove the prefix from the old variable names.
④ Convert the new year variable to numeric.
⑤ Take the old cell values and create a new `living_wage` variable.

Exercise 6

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

Part III

Data visualisation

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 depends first and foremost on the **goal**, the **context**, and the **audience** of the visualisation. This choice will also be influenced (or restricted) by the type of variable(s) we wish to display and the number of variables. Common plots used to display combinations of different types of data are given in following table:

Table 7.1: Common data visualisations, classified by type and number of variables, presented with the `geom` function used to generate them.

Number of variables	Type of variables	Name of visualisation	R function
One variable	Categorical	Frequency table	table
		Bar chart	geom_bar
	Numerical	Histogram	geom_histogram
	Spatial	Map	geom_sf
	Temporal	Line plot	geom_line
Two variables	Two categorical	Frequency table	table
		Stacked/side-by-side bar chart	geom_bar
	One numeric, one categorical	Dot plot	geom_point
		Box plot	geom_boxplot
	Two numerical	Scatterplot	geom_point
>2 variables	>2 categorical	Table	table

Table 7.1: Common data visualisations, classified by type and number of variables, presented with the `geom` function used to generate them.

Number of variables	Type of variables	Name of visualisation	R function
	2 numeric, one categorical, or > 2 numeric	Scatterplot with different colours/symbols/sizes	geom_point

For a more comprehensive list (including some non-standard graphs), visit the From data to viz website.

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

TThe `ggplot2` package implements a 'grammar of graphics' approach, in which graphs are composed of multiple layers. According to the grammar of graphics, all visualisations must contain three elements: the data, the information we wish to display, and some mapping, describing how to visualise the information.

To illustrate this, we can create a relatively simple visualisation which investigates the distribution of tenure types in the responses of the English Housing Survey (EHS). The most appropriate visualisation for this data would be a bar chart, where each bar represents the number of responses in each tenure type.

The first element required for a `ggplot` is the data. As the other two layers are missing, this will just produce a blank plot area:

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

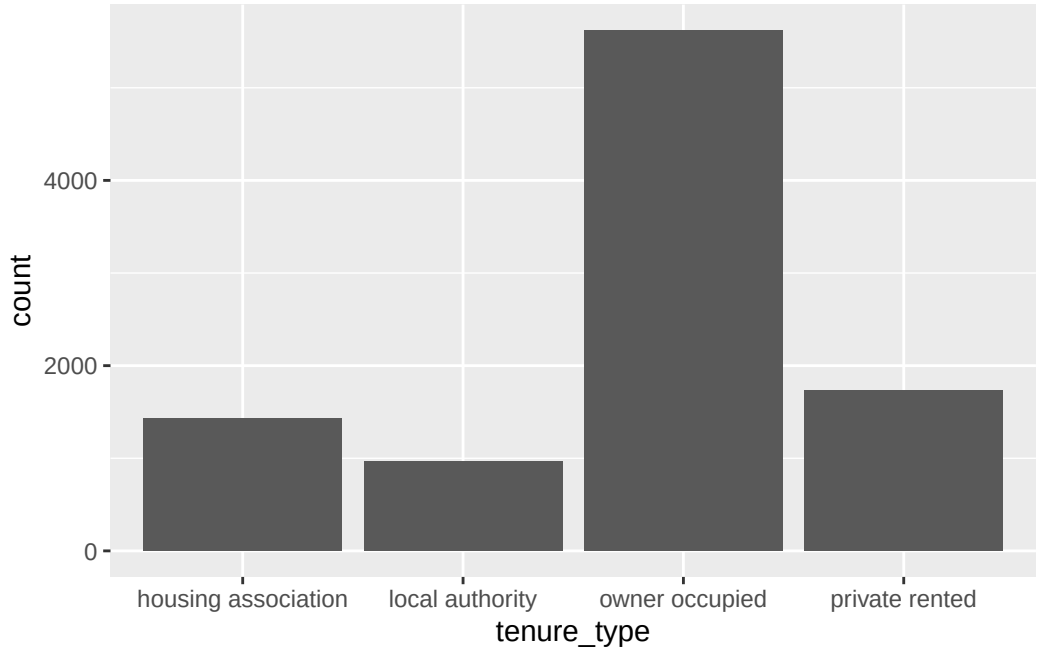
The second element that is required for a `ggplot` is the information we wish to present. In this case, we want to show the `tenure_type` variable:

```
ggplot(data = ehs_tidy,
       aes(x = tenure_type))
```

> **Note**
>
> Any aspect of the visualisation that is determined by a variable in the data must be surrounded by the `aes()` wrapper. For example, the x-axis of the visualisation is determined by the `tenure_type` variable which is given by the argument `aes(x = tenure_type)`. This will be explained in more detail later in this chapter.

The third and final element required to generate a visualisation in `ggplot2` is some physical markings of the data. Most visual markings within `ggplot` are determined by a `geom` object. In this case, the visual representation of the data will be given by bars:

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



> **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.

The plot may not be pretty, but it contains all three elements required by graphics. Additional layers will be introduced throughout this chapter to improve the design of this visualisation.

7.3 Exporting visualisations

Graphs appear in the plot tab in the bottom-right of the RStudio interface 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,
                     aes(x = tenure_type, weight = weighting)) +
  geom_line()

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

The `ggsave` function can be customised to change the file type, `height`, `width` and resolution (using the `dpi` argument).

> **Hint**
>
> `ggsave` is compatible with a range of file types, including `png`, `jpg`, `pdf` and `svg`. Saving these visualisations in a vectorised format, such as `svg` allows graph elements to be edited outside of R. For example, after pasting an exported `svg` file into Microsoft Word, ungroup the image. This allows customisation of axes text, legends, background colours, etc.
> External editing is not recommended as output would no longer be reproducible via R.

Exercise 7

1. Choose an appropriate visualisation to investigate the change in household disposable income between 2012 and 2024. Comment on your findings.

> **Caution**
>
> This exercise requires a visualisation that is appropriate for temporal numeric data. For inspiration, check Table 7.1.
> This visualisation will require a continuous time variable on the x-axis that incorporates both the year and the quarter. If this does not currently exist, create it using the `mutate` function.

7.4 Aesthetic values

Any information that we are presenting that is taken from the data must be given within the `aes` wrapper. The argument each variable takes within this wrapper depends on the element of the graph which it defines. Additional variables can be added to a visualisation by using them to customise other elements of a graph, such as:

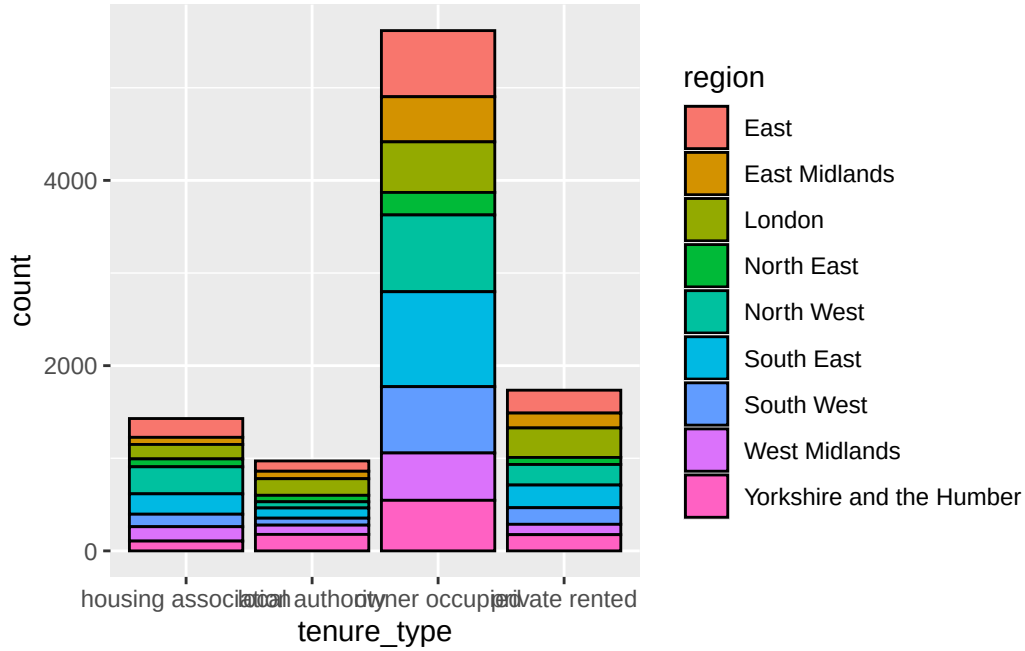
- `colour`: determines the colour of points (for dot and scatterplots), lines (for line graphs), or borders (for bar charts, histograms and pie charts)
- `fill`: determines the colour of bars or segments
- `shape`: changes the symbols presented on dot and scatterplots
- `linetype`: customises the type of line displayed (solid by default, but can be used to show dashed lines, etc)
- `size`: determines the size of points
- `linewidth`: changes the line width
- `alpha`: controls the transparency of graph elements

These options can be set manually or 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 `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) +
  geom_bar(aes(x = tenure_type, fill = region),                    ①
           colour = "black")                                       ②
```

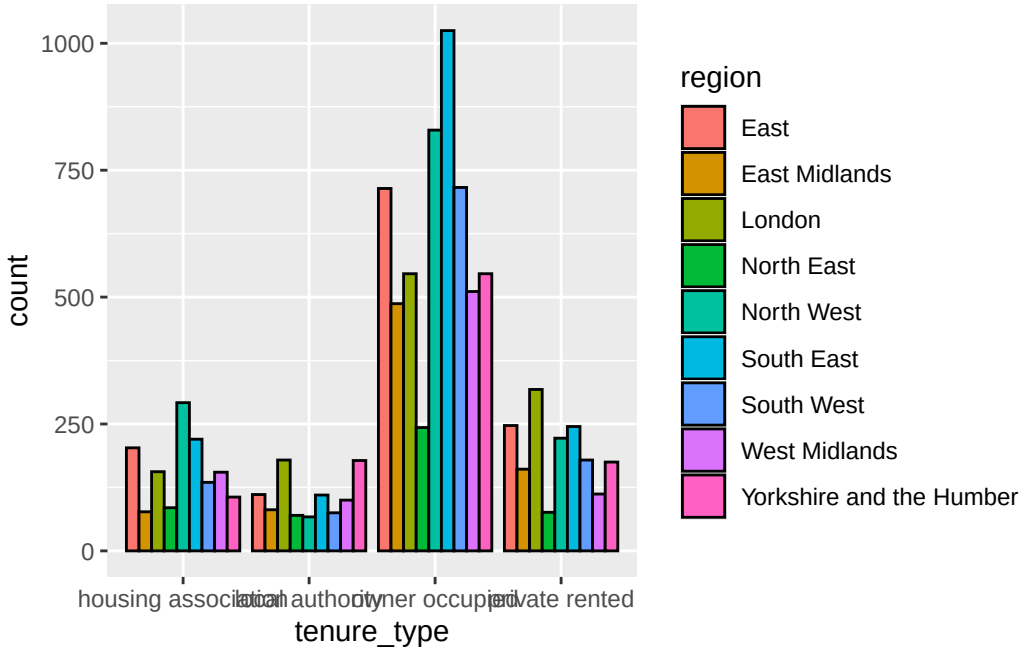
① Define the x axis and fill inside aes
② Manually define colour outside aes



> **Note**
>
> Aesthetic markings from the data specified in the `ggplot` function are applied to each `geom` in the visualisation. These aesthetics can also be specified in each `geom` separately. Where aesthetics are consistent over multiple `geom` layers, it is better to specify them in the `ggplot` function to avoid repetitive coding.

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 help file (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.

Exercise 8

1. Disposable income is the sum of labour income, non-labour income, and net taxes and benefits. Decompose the time series created in Exercise 7 to show how the contribution of these elements to total disposable income varied across time.

> **Exercise hint**
>
> Consider visualising this data as a stacked area chart rather than a line graph. The time variable will remain on the x-axis, the y-axis will be the total income, but the `fill` will be determined by the source of this income (labour, non-labour, or taxes and benefits). If this `fill` variable does not currently exist in the current **wide format**, consider how

this could be `pivoted` to convert it into an alternative, more appropriate format.

7.5 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'`.

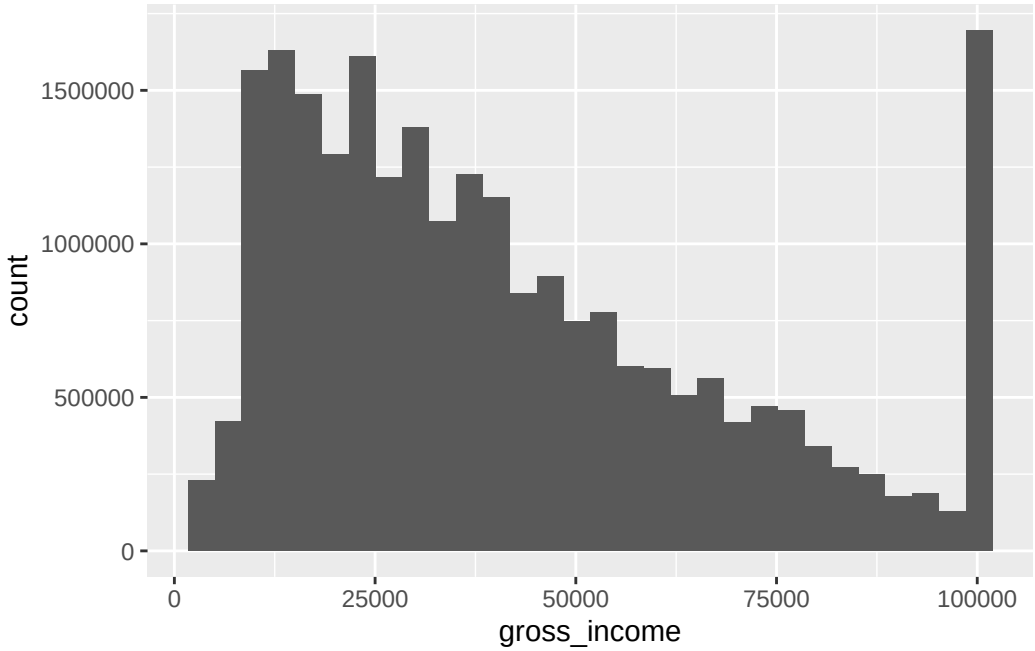
7.5.1 Customising axes

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 investigate the distribution of gross incomes in the EHS data, we could generate a **weighted** histogram. The `geom` required for this visualisation would be `geom_histogram`, which also contains an optional `aes` argument `weights`:

> ⚠ Warning
>
> Remember that the value `100000` is actually a grouped variable containing all household with a gross income of £100,000 or over.

```
ggplot(data = ehs_tidy) +
  geom_histogram(aes(x = gross_income, weight = weighting))
```



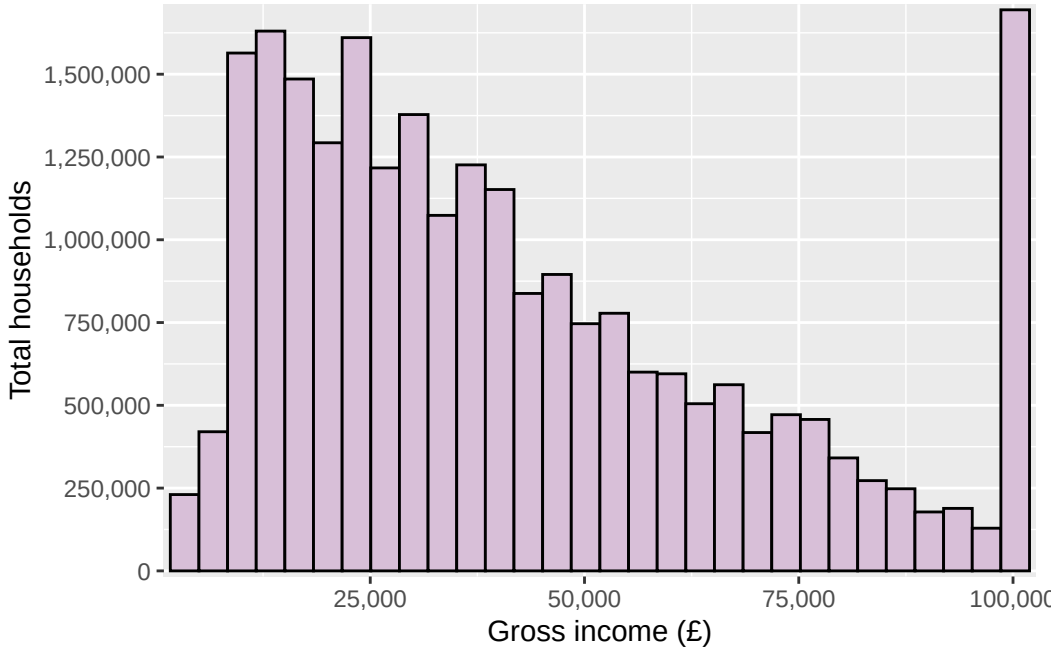
We could improve the appearance of this histogram by changing the axis label, the colour scheme of the bars, and could remove the blank space between the graph area and the axes. As the gross income is a numberic variable, we could use the `scale_x_continuous` function to customise its appearance. Arguments 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. This can be combined with a function from the `scales` package (which is automatically loaded with `ggplot2`) to apply common formats to labels (see the package reference website for examples).
- `expand = expansion(0)` removes the default blank space around the axis limits (this can also be used to add space by replacing `0` with either `add =` or `mult =` depending if this change is additive or multiplicative)
- `transform =` transform the scale the axis is shown on. Transformations include `reverse`, `sqrt`, `log`, etc. For a full list, view the appropriate help file

```
ggplot(data = ehs_tidy) +
  geom_histogram(aes(x = gross_income, weight = weighting),
                 colour = "black", fill = "thistle") +                    ①
  scale_x_continuous(name = "Gross income (£)",                           ②
                     expand = expansion(add = 750),                       ③
                     labels = scales::label_comma()) +                    ④
```

```
  scale_y_continuous(name = "Total households",
                     expand = expansion(mult = .01),
                     breaks = seq(0, 1750000, by = 250000),                    ⑤
                     labels = scales::label_comma())
```

① Manually change the outline and fill of bars to improve the appearance
② Change the axis title to remove underscore, add capitalisation, and add scale
③ Control the blank space between the axis and bars (add 750 to either side)
④ Format axis labels to have a comma every 3 digits (e.g. convert `100000` to `100,000`) to improve readability
⑤ Add axis breaks every 250,000 households (look up the `?seq` help file to learn more about how this can be customised)



7.5.2 Customising colour scales

There is a wide range of options available for customising `colour` and `fill` aesthetics within `ggplot2`. The choice will depend on the type of variable determining colours (whether it is numeric or categorical) and whether we want to use a pre-defined colour palette or manually specify our own.

Warning

When choosing a colour palette, be sure that all colours are distinct to everyone, including those with colour-vision deficiencies. To help check this is the case, use a colour blindness simulator to see what a visualisation looks like under different types of colour blindness. Avoid potentially harmful stereotypes when choosing colours to represent groups, and avoid cyclical palettes, such as the rainbow palette, to avoid confusion between high and low values.

7.5.2.1 Pre-built colour palettes

There are thousands of colour palettes that are available within R. Some of them are included within the `ggplot2` package, but there are many others that require additional package installation. This website gives a list and preview of all palettes currently available.

Colour palettes included within the `ggplot2` package (and therefore don't require any additional packages) are the viridis and colorbrewer scales. Both contain palettes that are colour-blind friendly and can be used for either continuous or discrete scales.

For continuous data, use `scale_colour_viridis_c` or `scale_colour_distiller` to select one of the in-built colour palettes (replace `colour` with `fill` when dealing with bars). For discrete or categorical variables, use `scale_colour_viridis_d` or `scale_colour_brewer` instead.

7.5.2.2 Manually defining a colour palette

There are various way of creating your own colour palette if you (or the department you are part of) have preferred colours.

For discrete or categorical variables, the `scale_colour_manual` (or `scale_fill_manual`) function allows colours to be specified using the `values` argument.

Style tip

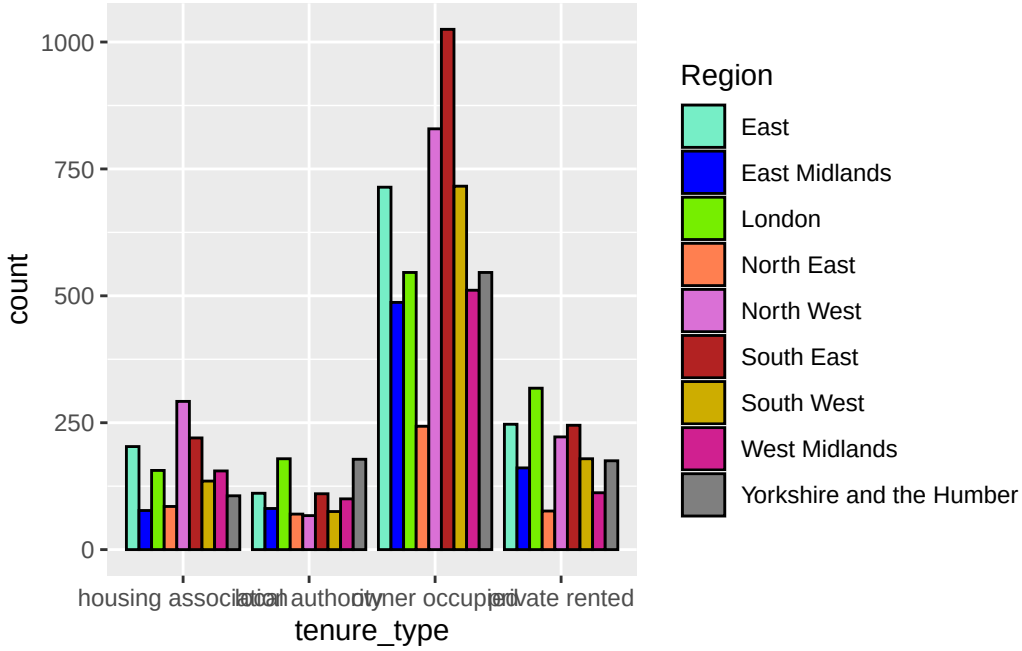
R contains a list of 657 pre-programmed colours that can be used to create palettes (run `colours()` in the console for a full list).

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).

> **Hint**
>
> 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 repetitive coding.

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

ggplot(data = ehs_tidy) +
  geom_bar(aes(x = tenure_type, fill = region),
           colour = "black",
           position = "dodge") +
  scale_fill_manual(name = "Region", values = region_palette)
```



When including a continuous variable, palettes can be created using gradients. The choice of function depends on the number of gradients required:

- `scale_colour_gradient` / `scale_fill_gradient`: specifies a two colour gradient based on a `low` and `high` value
- `scale_colour_gradient2` / `scale_fill_gradient2`: specifies a three colour gradient based on a `low`, `mid` (defined by the `midpoint` argument), and `high` value
- `scale_colour_gradientn` / `scale_fill_gradientn`: specifies a palette with more than three colours, customised by setting `colours` and corresponding `values`.

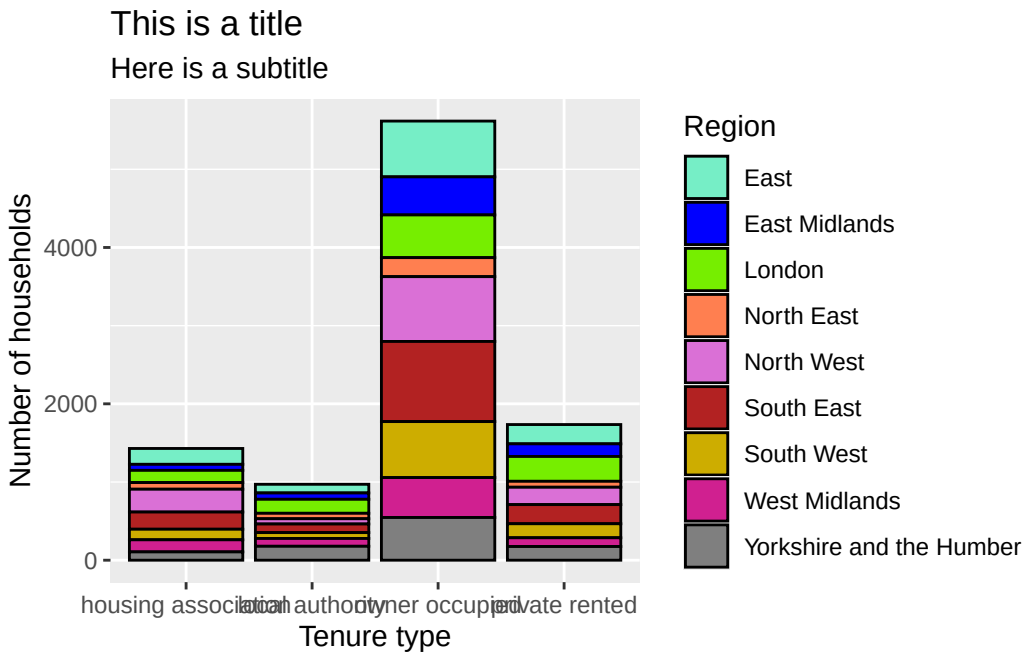
7.6 Annotations and titles

Besides the layers required to generate a visualisation, additional layers can be added enhance the messages given by the data, drawing readers' attention to interesting findings and the story you are trying to tell.

7.6.1 Plot, axes and legend titles

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, subtitles and footnotes to visualisations:

```
ggplot(data = ehs_tidy) +
  geom_bar(aes(x = tenure_type, fill = region),
           colour = "black") +
  scale_fill_manual(values = region_palette) +
  labs(title = "This is a title",
       subtitle = "Here is a subtitle",
       x = "Tenure type", y = "Number of households",
       fill = "Region")
```



Style tip

`\n` can be used to specify line breaks within the `labs` arguments.

Specifying any of the arguments in `labs` as `NULL` (no speech marks) removes the title from the visualisation.

Mathematical equations can be added into `labs` arguments by surrounding the text with the `quote()` function. Check `?plotmath` for examples of equation syntax.

7.6.2 Annotations

Annotations can be useful to include context to visualisations and draw attention to important messages. Annotations can include text labels, reference lines and shading, amongst others. `ggplot2` contains a number of `geom` objects that can be used to add annotation layers to a visualisation. As these annotations are added within `geom`s, they can be specified using values from the data (when wrapped in the `aes` function) or manually. This section will cover some common annotations but there are many others available (see the ggplot ebook for a more comprehensive list).

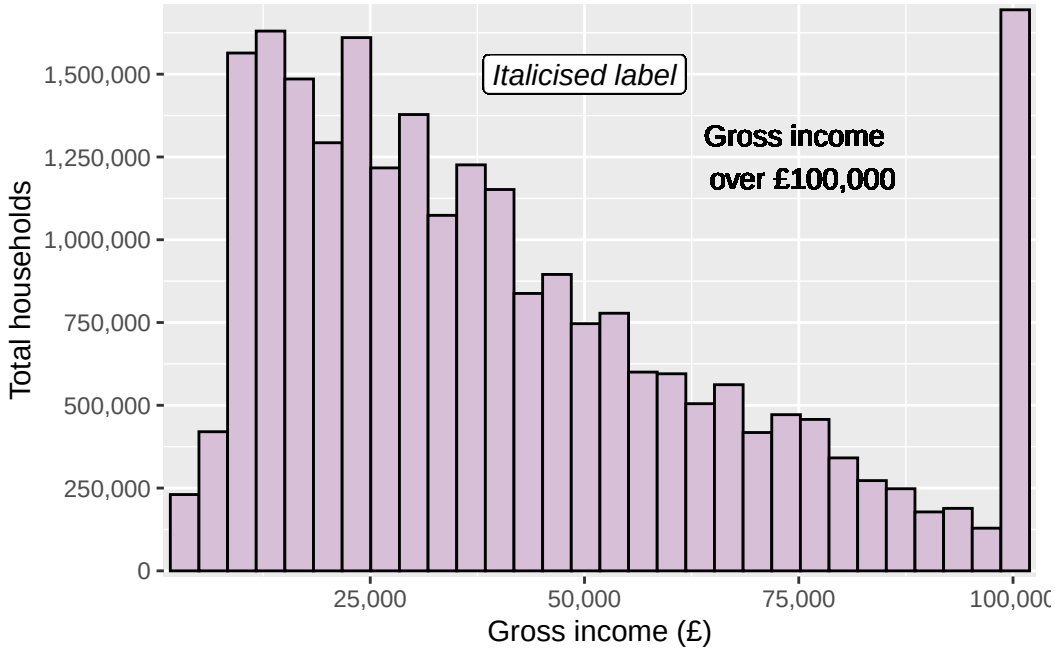
7.6.2.1 Text labels

Text labels can either be added using `geom_text` or `geom_label` (which adds text surrounded by a rectangular box, making it easier to read on busy backgrounds). Aesthetics such as `x`, `y` and `colour` can be used to customise text labels (either manually or from the data). Other aesthetics that can be added include:

- `label` defines the text displayed
- `angle` rotates the text
- `family` defines the font
- `fontface` can be changed to make text `"bold"` or `"italic"`

```
ggplot(data = ehs_tidy) +
  geom_histogram(aes(x = gross_income, weight = weighting),
                 colour = "black", fill = "thistle") +
  scale_x_continuous(name = "Gross income (£)",
                     expand = expansion(add = 750),
                     labels = scales::label_comma()) +
  scale_y_continuous(name = "Total households",
                     expand = expansion(mult = .01),
                     breaks = seq(0, 1750000, by = 250000),
                     labels = scales::label_comma()) +
  geom_text(x = 75000, y = 1250000,
            label = "Gross income \n over £100,000") +
```

```
  geom_label(x = 50000, y = 1500000, label = "Italicised label",
             fontface = "italic")
```



Adding text through `geoms` will work but notice that the annotations looks a little blurry on the text. This is because `geom` layers take the data into account and assume that you want the same number of layers/markings as observations in the data. This means that rather than adding a single text or label, `ggplot` is actually adding 9752. To overcome this, we can use the `annotate` function instead.

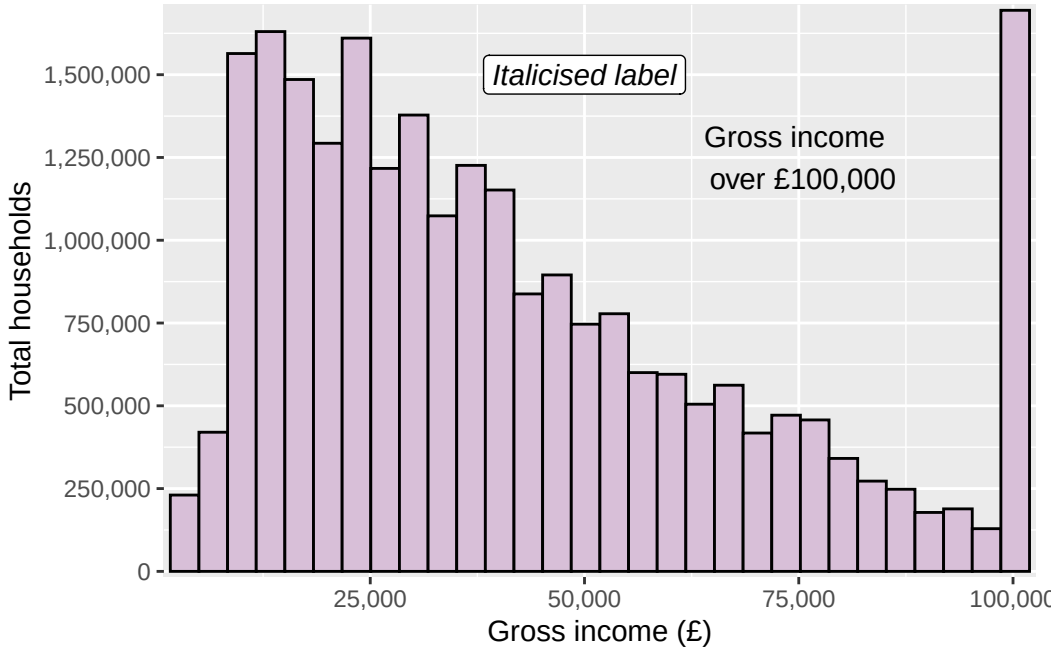
7.6.3 Annotate function

The `annotate` function will add single `geom` layers to a visualisation while disregarding the rest of the data. This is useful when adding annotations such as text, labels, shapes or arrows. `annotate` functions require the same arguments as the corresponding `geom`, with an additional argument that specifies the `geom` we require.

For example, the blurry labels on our previous plot can be fixed by replacing `geom_text` and `geom_label` with `annotate` functions:

```
ggplot(data = ehs_tidy) +
  geom_histogram(aes(x = gross_income, weight = weighting),
                 colour = "black", fill = "thistle") +
```

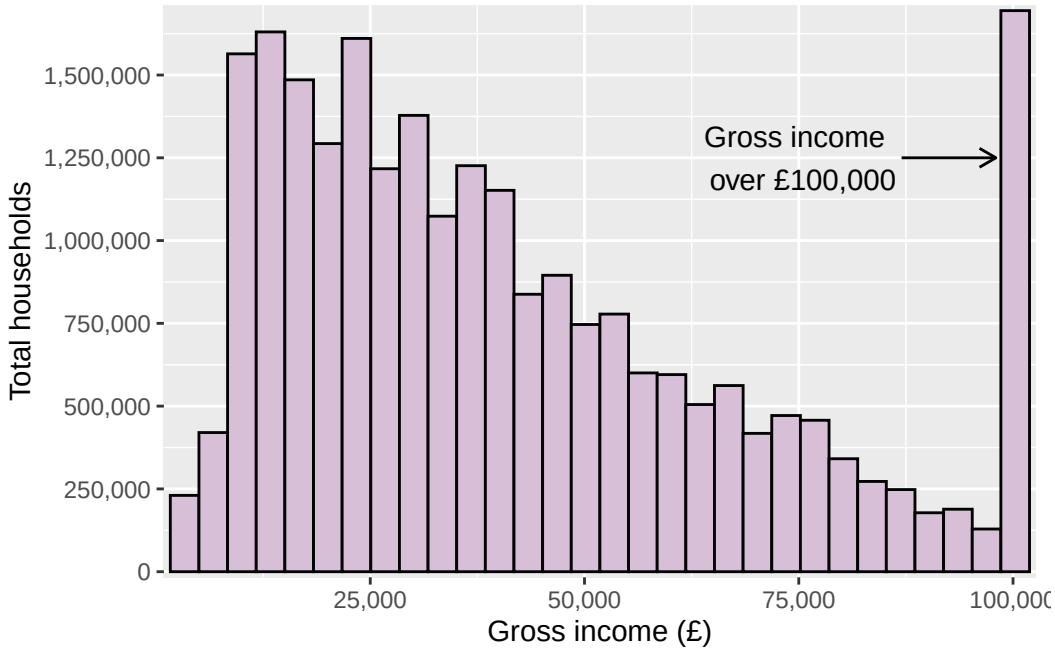
```
  scale_x_continuous(name = "Gross income (£)",
                     expand = expansion(add = 750),
                     labels = scales::label_comma()) +
  scale_y_continuous(name = "Total households",
                     expand = expansion(mult = .01),
                     breaks = seq(0, 1750000, by = 250000),
                     labels = scales::label_comma()) +
  annotate("text", x = 75000, y = 1250000,
           label = "Gross income \n over £100,000") +
  annotate("label", x = 50000, y = 1500000, label = "Italicised label",
            fontface = "italic")
```



Text labels can be combined with lines and arrows to make them clearer, using the `curve` (for curved lines) or `segment` (for straight lines) `geoms`. Both contain the optional argument `arrow` which adds an arrow to the curved line (this must be defined within the `arrow` function, which can be used to adjust the size or shape of the arrow):

```
ggplot(data = ehs_tidy) +
  geom_histogram(aes(x = gross_income, weight = weighting),
                 colour = "black", fill = "thistle") +
  scale_x_continuous(name = "Gross income (£)",
                     expand = expansion(add = 750),
                     labels = scales::label_comma()) +
```

```
  scale_y_continuous(name = "Total households",
                     expand = expansion(mult = .01),
                     breaks = seq(0, 1750000, by = 250000),
                     labels = scales::label_comma()) +
  annotate("segment", x = 87000, xend = 98000, y = 1250000,
           arrow = arrow(length = unit(.25, "cm"))) +
  annotate("text", x = 75000, y = 1250000,
           label = "Gross income \n over £100,000")
```



Other useful annotations that can enhance visualisations' messages include:

- `rect`: draws a rectangle that can be used to highlight a section of the graph
- `geom_vline`, `geom_hline` and `geom_abline`: adds a vertical, horizontal, or diagonal reference line across the entire graph area (these `geom`s are not compatible with the `annotate` function)

7.7 Theme functions

The `theme` function modifies non-data components of the visualisation. For example, the legend position, label fonts, graph background, and grid lines. There are many options that can be adjusted within the `theme` function (see the help file `?theme` for a complete list). Often, it is efficient to begin with a pre-built theme and tweak elements that do not suit our purpose.

7.7.1 Pre-built themes

There are 8 complete themes programmed in the `ggplot2` package. These are:

Style tip

Although pre-built `theme` functions do not require arguments to run, they all contain the optional argument `base_size` which set the default font size (defaulted to 11). To ensure visualisations are as accessible and inclusive as possible, ensure this is set to at least 12 for printed graphs or 36 for presentations.

7.7.2 Customising themes

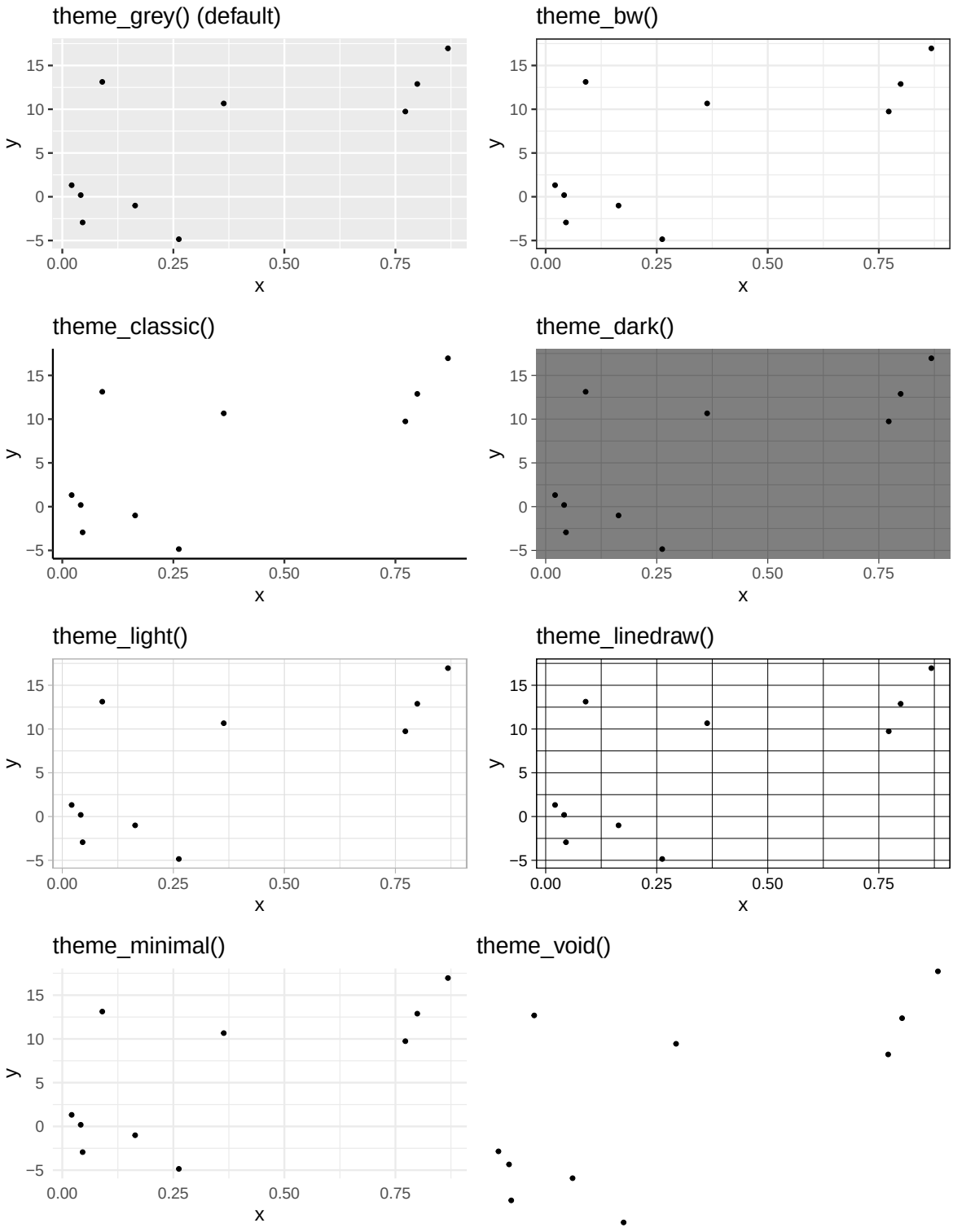
Individual elements of a visualisation's theme can be customised within the `theme` functions. Many elements that can be customised using the `theme` require an `element` wrapper. This wrapper is determined by the type of object that we are customising, the four options are:

- `element_text` when customising text, e.g. axis titles and labels
- `element_rect` when customising backgrounds, e.g. the graph area
- `element_line` when customising lines, e.g. gridlines
- `element_blank` to remove elements

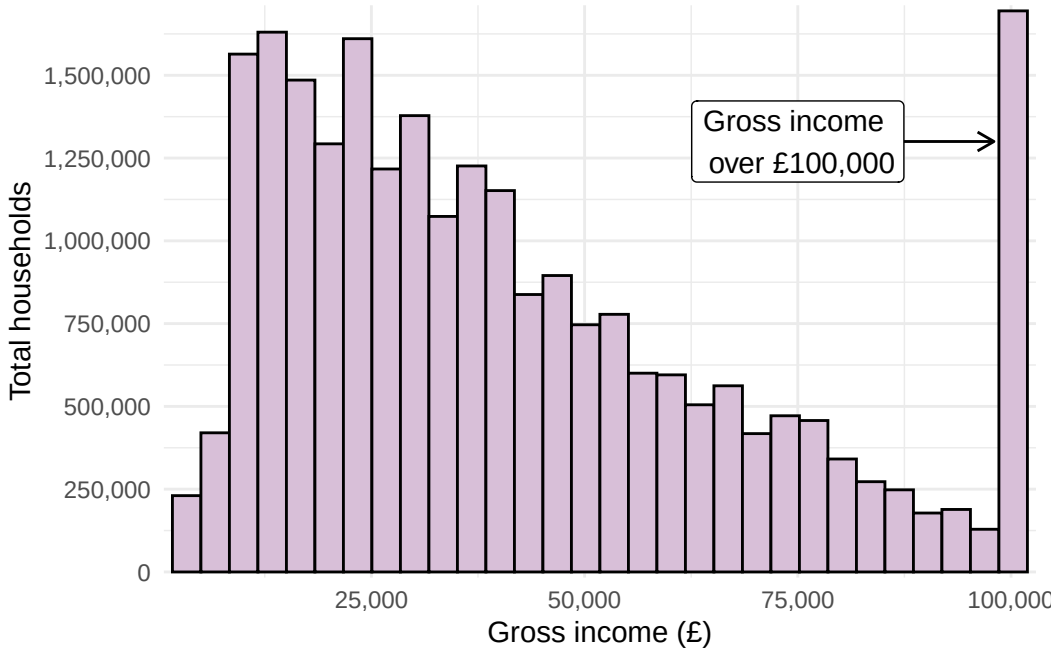
Elements that do not require these wrappers are often related to positioning. A common example of this is the `legend.position` argument which can be set to `"left"`, `"right"` (default), `"top"`, `"bottom"`, or removed using `"none"`.

For example, the histogram we have been building showing household gross income can be customised using `theme` functions:

```
histogram_minimal <- ggplot(data = ehs_tidy) +
  geom_histogram(aes(x = gross_income, weight = weighting),
                 colour = "black", fill = "thistle") +
  scale_x_continuous(name = "Gross income (£)",
                     expand = expansion(add = 1000),
                     labels = scales::label_comma()) +
  scale_y_continuous(name = "Total households",
                     expand = expansion(mult = .01),
                     breaks = seq(0, 1750000, by = 250000),
                     labels = scales::label_comma()) +
  annotate("segment", x = 87000, xend = 98000, y = 1300000,
           arrow = arrow(length = unit(.25, "cm"))) +
  annotate("label", x = 75000, y = 1300000,
           label = "Gross income \n over £100,000") +
  theme_minimal()
```

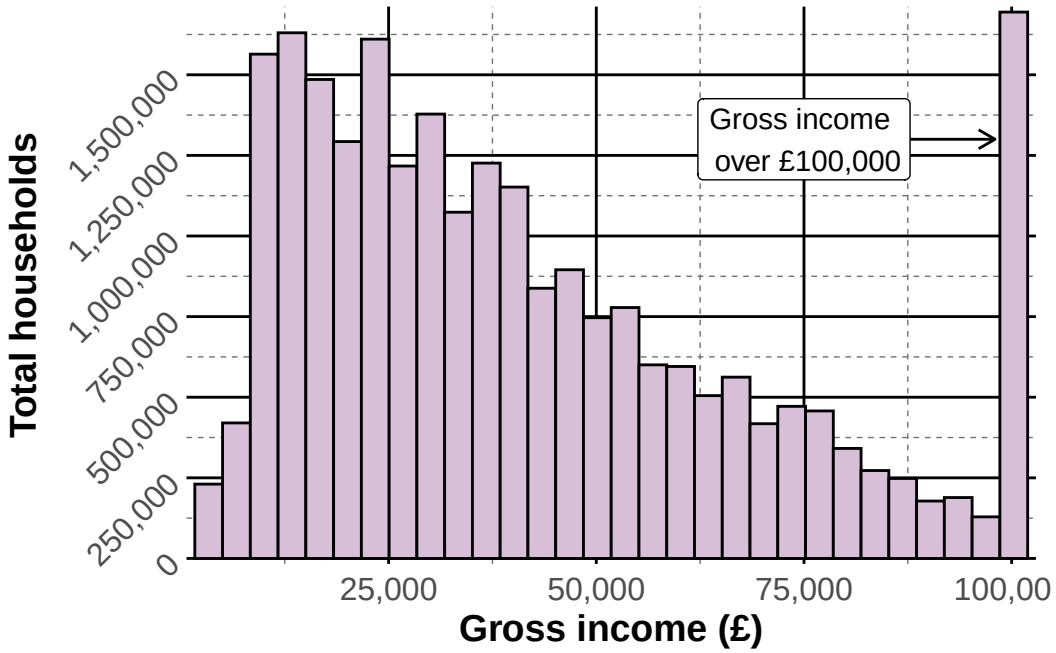

theme_grey() (default)
theme_bw()
theme_classic()
theme_dark()
theme_light()
theme_linedraw()
theme_minimal()
theme_void()
x
y
0.00
0.25
0.50
0.75
15
10
5
0
−5

```
histogram_minimal
```




```
histogram_minimal +
  theme(axis.title = element_text(size = 14, face = "bold"),                ①
        axis.text.x = element_text(size = 12),                              ②
        axis.text.y = element_text(size = 12, angle = 45),                  ③
        panel.grid.major = element_line(colour = "black"),                  ④
        panel.grid.minor = element_line(colour = "grey45",                  ⑤
                                        linetype = "dashed"))
```

① Increase the axis title size and make them bold
② Set the x-axis text size to 12
③ Set the y-axis text size to 12 and rotate them 45 degrees
④ Add major grid lines
⑤ Add dashed minor grid lines



Style tip

Good visualisations require a middle ground between overly minimal design, which can make interpretation difficult, and charts overloaded with clutter. The bold and dashed grid lines on this visualisations are a good example of where 'chart junk' can distract from the data.

Visualisations should strive to make the data are the most important part of the graphic, whilst ensuring there is sufficient context provided by non-data elements.

7.7.3 Creating a theme

One benefits of using `theme` functions is that visualisations will remain consistent in terms of their design. Custom themes can be saved as functions and added to `ggplot` objects in place of the in-built themes. For example,

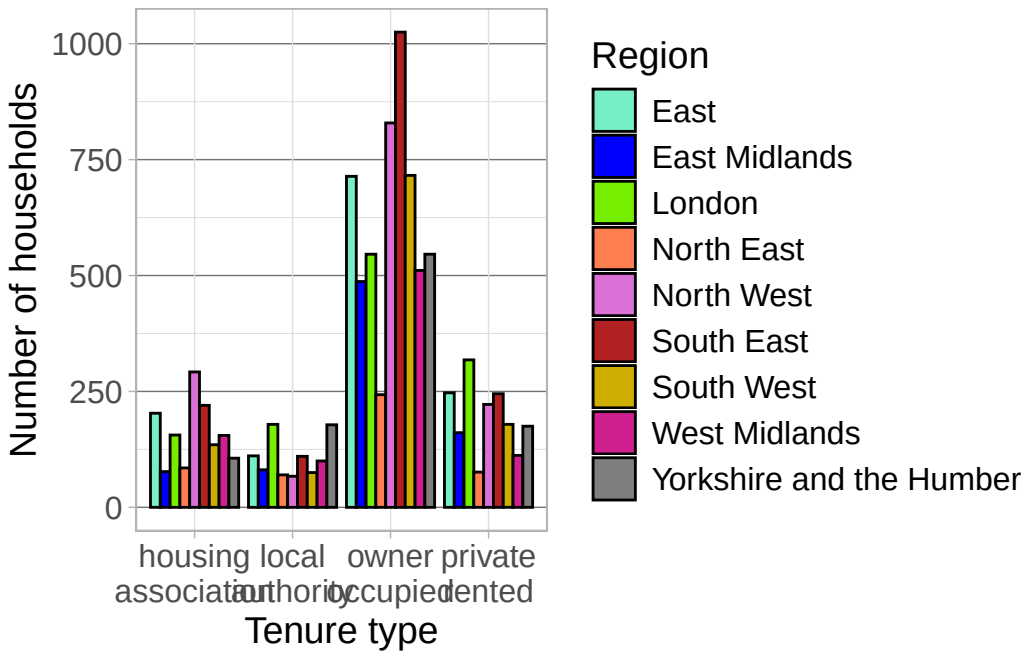
To create our own function in R, we first give it a name and attach `function()` followed by curly brackets `{}`, with the function defined inside those brackets.

For example, to create our own theme function, called `theme_dataviz`, which sets the title font size to 18, the axis and legend titles to size 14, the axis and legend text to size 12, adds just gridlines to the y-axis, and changes the background colours, we use the following:

```
theme_dataviz <- function() {
  theme_light() +
  theme(plot.title = element_text(size = 18),
        axis.title = element_text(size = 14),
        axis.text = element_text(size = 12),
        legend.text = element_text(size = 12),
        legend.title = element_text(size = 14),
        panel.grid.major.y = element_line(colour = "grey45"))
}
```

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

```
ggplot(data = ehs_tidy) +
  geom_bar(aes(x = tenure_type, fill = region),
           colour = "black",
           position = "dodge") +
  scale_fill_manual(name = "Region", values = region_palette) +
  scale_x_discrete(labels = scales::label_wrap(12)) +
  labs(x = "Tenure type", y = "Number of households") +
  theme_dataviz()
```



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

7.8 Facet functions

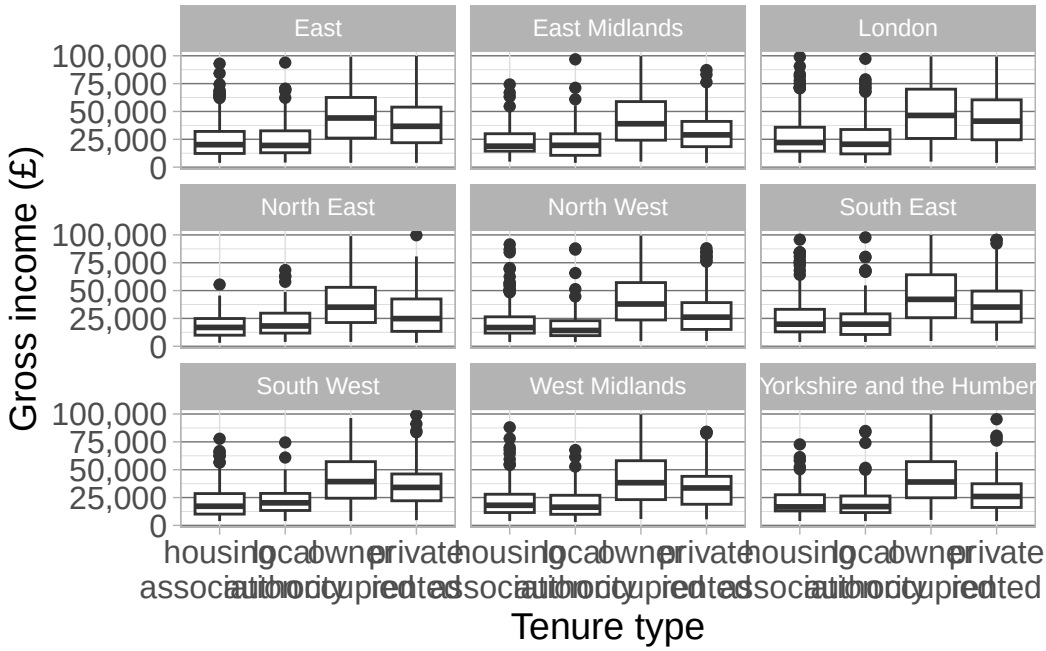
Faceting allows us to divide a plot into subplots based on some grouping variable within the data. This allows us to show multiple variables in the same visualisation without risking overloading the plot and losing the intended message.

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 %>%
  filter(gross_income != 100000) %>%                                    ①
  ggplot() +                                                            ②
  geom_boxplot(aes(x = tenure_type, y = gross_income)) +
  scale_x_discrete(labels = scales::label_wrap(8)) +                    ③
  scale_y_continuous(labels = scales::label_comma()) +
  labs(x = "Tenure type", y = "Gross income (£)") +
  facet_wrap( ~ region) +
  theme_dataviz()
```

① Remove gross income >= £100,000
② We do not need to specify data, it is already passed through the pipes
③ Wrap tenure type labels to avoid overlap



Exercise 9

Return to the visualisation showing the change in disposable income over time, decomposed into different income streams (labour, non-labour, and taxes and benefits). Adapt the visualisation to ensure it is accessible, compelling, and clear. This can include:

- Adding an appropriate title and caption with data source information
- Adapting the colour scheme to make the differences more obvious
- Adding labels to commmunicate important findings to the reader
- Adjusting the `theme` to ensure text is large enough, values are clear but not overwhelmed by 'chart junk'

Save this visualisation and add it into a Word document, along with a brief interpretation of the visualisation.

> Exercise hint
>
> Add layers to the previous `geom_area` from exercise 8. Use the `labs` function to add a `title` and `caption`, and to ensure axes and legend titles are clear. Use `scale_fill_manual` to manually enter colours or `scale_fill_brewer` to choose a pre-existing `palette`.
> Consider adding text labels to each element of the disposable income to give the percent-

age split represented by each element.
Ensure your theme sets all text and titles to size 12 or larger, adds enough grid lines to make values readable, but not so many that it distracts from the data.

A Exercise 2 solutions

A.1 Question 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)

Solution

1. Add a `#` at the beginning of the row, followed by your name and the date, for example:

```
# Sophie 18/11/2024
```

A.2 Question 2

Use R to answer to following sums:

a. 64^2

b. $3432 \div 8$

c. 96×72

For each part of question 2, copy the result from the console and paste them onto the same line of the script file as the code. Do this in a way that ensures there are no error messages if you were to run the entire script file.

Solution

For each calculation, copy the result from the console and paste it in the script file after a `#` symbol:

```
64^2   # 4092

3432 / 8  # 429

96 * 72  # 6912
```

B Exercise 3 solutions

B.1 Question 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`

Solution

The first step is to load in the data. However, before this, we need the name of the file. We can look into our documents to get this, but the `list.files` function will do it for us:

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

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

From the console, we can copy and paste the file name into the `read_spss` function:

```
ehs_interview_tidy <- read_spss("data/interviewfs21_EUL.sav")
```

Next, we can `select` just the variables we need to reduce the data size:

```
ehs_interview_tidy <- read_spss("data/interviewfs21_EUL.sav") %>%
  select(serialanon, HYEARGRx, lenresb, rentwkx, mortwkx, freeLeas)
```

We can now explore this data to see which variables need converting, and which are truly numeric:

```
str(ehs_interview_tidy)
```

```
tibble [9,752 x 6] (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"
 $ HYEARGRx  : dbl+lbl [1:9752]  38378,  26525,  25273,  51280,  14365,  38955,  2137...
   ..@ label        : chr "Household gross annual income (inc. income from all adult household
   ..@ format.spss  : chr "F8.2"
   ..@ display_width: int 10
   ..@ labels       : Named num 1e+05
   .. ..- attr(*, "names")= chr "£100,000 or more"
 $ lenresb   : dbl+lbl [1:9752] 7, 8, 6, 4, 8, 8, 6, 2, 6, 8, 6, 4, 8, 5, 4, 5, 4, 3,...
   ..@ label        : chr "Length of residence"
   ..@ format.spss  : chr "F8.0"
   ..@ 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" "less than 1 year" "one y
 $ rentwkx   : dbl+lbl [1:9752]    NA,    NA,    NA,    NA,    NA,    NA,    NA,    N...
   ..@ label        : chr "Total weekly rent payable (rent plus housing benefit)"
   ..@ format.spss  : chr "F8.2"
   ..@ display_width: int 10
   ..@ labels       : Named num -9
   .. ..- attr(*, "names")= chr "question not applicable - owner occupier and not shared owne
 $ mortwkx   : dbl+lbl [1:9752]    NA,    NA,    NA, 184.6,  88.8,    NA,    NA, 313....
   ..@ label        : chr "Weekly mortgage payments"
   ..@ format.spss  : chr "F8.2"
   ..@ display_width: int 10
   ..@ labels       : Named num [1:3] -9 -8 0
   .. ..- attr(*, "names")= chr [1:3] "not applicable - tenant" "unknown" "no payments - own
 $ freeLeas  : dbl+lbl [1:9752]  1,  1,  1,  1,  1,  1,  1,  1, NA,  1,  1, NA,  1, N...
   ..@ label        : chr "Freehold or leasehold"
```

```
   ..@ format.spss  : chr "F8.0"
   ..@ display_width: int 10
   ..@ labels       : Named num [1:4] -9 -8 1 2
   .. ..- attr(*, "names")= chr [1:4] "does not apply" "no answer" "freehold" "leasehold"
 - attr(*, "label")= chr "Aggregated File"
```

As with the general data, all variables are classified as `dbl + lbl` by R. Of these, the length of residence and freehold/leashold variables appear to by categorical. There are also labels attached to the gross annual income (for those over £100,000) which we need to be aware of when analysing this data.

Therefore, our next step will involve converting the categorical variables into factors:

```
ehs_interview_tidy <- read_spss("data/interviewfs21_EUL.sav") %>%
  select(serialanon, HYEARGRx, lenresb, rentwkx, mortwkx, freeLeas) %>%
  mutate(length_residence = as_factor(lenresb),
         freehold_leasehold = as_factor(freeLeas))

head(ehs_interview_tidy)
```

```
# A tibble: 6 x 8
  serialanon  HYEARGRx  lenresb        rentwkx mortwkx freeLeas length_residence
  <dbl+lbl>   <dbl+lbl> <dbl+lbl>      <dbl+l> <dbl+l> <dbl+lb> <fct>
1 20220000001 38378.    7 [20-29 year~ NA       NA     1 [free~ 20-29 years
2 20220000005 26525     8 [30+ years]  NA       NA     1 [free~ 30+ years
3 20220000006 25272.    6 [10-19 year~ NA       NA     1 [free~ 10-19 years
4 20220000012 51280.    4 [3-4 years]  NA      185.    1 [free~ 3-4 years
5 20220000013 14365     8 [30+ years]  NA       88.8   1 [free~ 30+ years
6 20220000017 38955     8 [30+ years]  NA       NA     1 [free~ 30+ years
# i 1 more variable: freehold_leasehold <fct>
```

Finally, we need to rename the existing variables to ensure they are informative and follow the style rules, and remove any unnecessary variables:

```
ehs_interview_tidy <- read_spss("data/interviewfs21_EUL.sav") %>%
  select(serialanon, HYEARGRx, lenresb, rentwkx, mortwkx, freeLeas) %>%
  mutate(length_residence = as_factor(lenresb),
         freehold_leasehold = as_factor(freeLeas)) %>%
  rename(id = serialanon,
         gross_income = HYEARGRx,
         weekly_rent = rentwkx,
         weekly_mortgage = mortwkx) %>%
```

```
  select(id, gross_income, length_residence, weekly_rent, weekly_mortgage,
         freehold_leasehold)
```

B.2 Question 2

Save the tidy interview dataset as a csv file with an appropriate file name.

Solution

```
write_csv(ehs_interview_tidy, file = "saved_data/ehs_interview_tidy.csv")
```

B.3 Question 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?

Solution

For the first two part, we use the `filter` function to return a subgroup matching the condition, and combine this with the `count` function that counts the number of rows in a tibble:

```
ehs_interview_tidy %>%
  filter(between(weekly_rent, 150, 300)) %>%
  count()
```

```
# A tibble: 1 x 1
      n
  <int>
1   993
```

```
ehs_interview_tidy %>%
  filter(is.na(weekly_rent), is.na(weekly_mortgage)) %>%
  count()
```

```
# A tibble: 1 x 1
      n
  <int>
1  2956
```

There were 993 respondents that paid weekly rent of between £150 and £300.

There were 2956 respondents that did not give a response to either the weekly rent or mortgage question.

The final part could usually be carried out with the base R `max` function:

```
max(ehs_interview_tidy$gross_income)
```

```
<labelled<double>[1]>: Household gross annual income (inc. income from all adult household me
[1] 1e+05

Labels:
 value            label
 1e+05 £100,000 or more
```

However, the labels attached to the SPSS file showed that a value of 100000 actually represents a group of responders earning at least £100,000. Therefore, we cannot answer this question from the available data. If we were to analyse this variable, we would need to categorise the rest of the data, losing a lot of information. Failure to do this would produce invalid results.

C Exercise 4 solutions

C.1 Question 1

How many respondents had both weekly rent and mortgage payments given? What are the potential reasons for this?

Solution

We can combine the `filter` and `count` functions to answer the first part of this question:

```
ehs_tidy %>%
  filter(!is.na(weekly_rent), !is.na(weekly_mortgage)) %>%
  count()
```

```
# A tibble: 1 x 1
      n
  <int>
1   105
```

There were 105 respondents with both weekly rent and mortgage payments.

To understand why this is, we could first view this data (or a summary of the data) to see what other characteristics these respondents share:

```
ehs_tidy %>%
  filter(!is.na(weekly_rent), !is.na(weekly_mortgage)) %>%
  summary()
```

```
       id              weighting                   tenure_type
 Min.   :2.022e+10   Min.   :  305.5   housing association:  0
 1st Qu.:2.022e+10   1st Qu.:  628.9   local authority    :  0
 Median :2.022e+10   Median : 1058.4   owner occupied     :105
 Mean   :2.022e+10   Mean   : 1859.3   private rented     :  0
```

```
3rd Qu.:2.022e+10   3rd Qu.: 2535.4
Max.   :2.022e+10   Max.   :10568.5

          region    gross_income            length_residence  weekly_rent
South East   :22   Min.   :  9880   two years       :27      Min.   :  8.30
East         :19   1st Qu.: 26595   one year        :19      1st Qu.: 49.80
London       :19   Median : 39655   3-4 years       :15      Median : 69.69
North West   :13   Mean   : 41908   5-9 years       :15      Mean   : 86.23
South West   :13   3rd Qu.: 52323   less than 1 year: 9      3rd Qu.:120.00
East Midlands: 8   Max.   :100000   10-19 years     : 8      Max.   :219.23
(Other)      :11                    (Other)         :12
weekly_mortgage    freehold_leasehold
Min.   :  0.0231   freehold :29
1st Qu.:  0.0231   leasehold:65
Median : 60.0000   NA's     :11
Mean   : 71.6271
3rd Qu.:103.8462
Max.   :343.8138
```

All respondents in this group owned and lived in their own home. Most were leasehold properties, suggesting some of the weekly rent refers to lease payments. Other potential reasons could include shared ownership (which is not given as an option for tenure type), or respondents that lived with renters in the same property.

C.2 Question 2

Combine the weekly rent and mortgage variables into a single weekly payment variable.

Solution

Where only one value has been recorded, we want to use this in the new variable. Where both have been recorded, we will need to add the values together to get a weekly total.

There are a few different ways to do this. The first is to include an `if_else` statement in the `mutate` function, changing how the variable is calculated whether either value is missing or not:

```
ehs_tidy_ex4 <- ehs_tidy %>%
  mutate(weekly_total = if_else(is.na(weekly_rent) |                    ①
                                  is.na(weekly_mortgage),
                                coalesce(weekly_rent, weekly_mortgage),  ②
                                weekly_rent + weekly_mortgage))          ③
```

① If either `weekly_rent` or `weekly_mortgage` are missing
② Then return the non-missing value
③ Or else (if both are NOT missing), return the sum of these values

C.3 Question 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?

Solution

```
ehs_tidy_ex4 %>%
  group_by(region) %>%                                                  ①
  summarise(mean_payment = wtd.mean(weekly_total, weights = weighting,  ②
                                    na.rm = TRUE),
            median_payment = wtd.quantile(weekly_total,                 ③
                                          weights = weighting,
                                          probs = .5, na.rm = TRUE),
            sd_payment = sqrt(wtd.var(weekly_total, weights = weighting,  ④
                                      na.rm = TRUE)),
            lq_payment = wtd.quantile(weekly_total, weights = weighting,  ⑤
                                      probs = .25, na.rm = TRUE),
            uq_payment = quantile(weekly_total, weights = weighting,    ⑥
                                  probs = .75, na.rm = TRUE)) %>%
  ungroup()                                                             ⑦
```

① Calculate summaries per region
② Return the weighted mean
③ Return the weighted median (the 50th percentile)
④ Return the weighted standard deviation (the square root of the weighted variance)
⑤ Return the weighted lower quartile (the 25th percentile)
⑥ Return the weighted upper quartile (the 75th percentile)

⑦ Don't forget to `ungroup`

```
# A tibble: 9 x 6
  region            mean_payment median_payment sd_payment lq_payment uq_payment
  <fct>                    <dbl>          <dbl>      <dbl>      <dbl>      <dbl>
1 East                      182.          150        154.       104.        208.
2 East Midlands             151.          125.       117.        93         162.
3 London                    283.          254.       223.       135         346.
4 North East                111.           96.9       52.6       81         115.
5 North West                124.          110.        68.7       85.2       138.
6 South East                210.          179.       149.       121.        245.
7 South West                153.          138.        92.0       97         183.
8 West Midlands             142.          122.        88.1       92         150
9 Yorkshire and th~         120.          107.        76.4       82         133.
```

There are big differences between most mean and medians across regions, indicating that the data are not normally distributed. If we use the approximate 95% range formula (mean $\pm$ (2 $\times$ sd)), we would get negative values for almost all regions. Negative payments do not make sense in this context, confirming that the data are not normally distributed.

In this case, the median and IQR should be give, not the mean and standard deviation.

D Exercise 5 solutions

D.1 Question 1

Load in the OBR's household disposable income data (sheet 1.13). Split the period data into separate year and quarter variables, ensure that all variable names follow Tidyverse's style guide. Name this object `disposable_income`.

Solution

Using the same approach as the housing market sheet, load the range of cells containing the data required, not including the variable names. Add variable names manually after selecting the variables required, then split the time variable into years and quarters.

```
disposable_income <-
  read_xlsx("data/Detailed_forecast_tables_Economy_March_2024.xlsx",
                               sheet = "1.13",range = "B4:I51",
                               col_names = FALSE) %>%
  setNames(c("period", "labour_income", "emp_comp", "mixed_income",
             "emp_social_cont", "nonlabour_inc", "net_tax_benefits",
             "disposable_income")) %>%
  mutate(year = as.numeric(str_sub(period, start = 1L, end = 4L)),
         quarter = as.numeric(str_sub(period, start = -1L, end = -1L))) %>%
  select(-period)

str(disposable_income)
```

```
tibble [48 x 9] (S3: tbl_df/tbl/data.frame)
 $ labour_income    : num [1:48] 196 197 197 198 199 ...
 $ emp_comp         : num [1:48] 208 210 210 211 212 ...
 $ mixed_income     : num [1:48] 25.4 26.2 25.8 25.6 25.8 ...
 $ emp_social_cont  : num [1:48] 38.2 38.6 38.9 38.6 38.6 ...
 $ nonlabour_inc    : num [1:48] 67 68.1 67.5 66.5 66.4 ...
 $ net_tax_benefits : num [1:48] 16.7 18.2 17.5 19 18.5 ...
 $ disposable_income: num [1:48] 279 284 282 284 284 ...
```

```
$ year              : num [1:48] 2012 2012 2012 2012 2013 ...
$ quarter           : num [1:48] 1 2 3 4 1 2 3 4 1 2 ...
```

D.2 Question 2

Load the OBR's national living wage data (sheet 1.14).

Solution

Using the same approach as earlier, load the correct sheet in, selecting cells with data included. Variable names cannot begin with numbers, so rename them either manually, or by adding a prefix. The `rename_with` function allows us to rename variables by applying a function to them, in this case `paste0` which combines elements in the function separated by commas:

```
living_wage <-
  read_xlsx("data/Detailed_forecast_tables_Economy_March_2024.xlsx",
            # Specify the sheet and range of cells to keep
            sheet = "1.14", range = c("C4:K5")) %>%
  # Rename variables by pasting the prefix "year_" to the original name
  rename_with(~paste0("year_", .x))
```

E Exercise 6 solutions

E.1 Question

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

Solution

Use `full_join` to combine the housing and disposable data by year and quarter, then pipe to apply `full_join` to the resulting data and add the living wage, joining by year. As there are multiple year rows in the housing and disposable income data, include the argument `muliple = "all"` to ensure the living wage variable is repeated for each quarter.

```
# full join the housing and labour market data
obr_data <- full_join(housing_market, disposable_income,
                      by = c("year", "quarter")) %>%
  # join this data to the living wage
  full_join(., living_wage_long, by = "year", multiple = "all")
```

F Exercise 7 solutions

F.1 Question 1

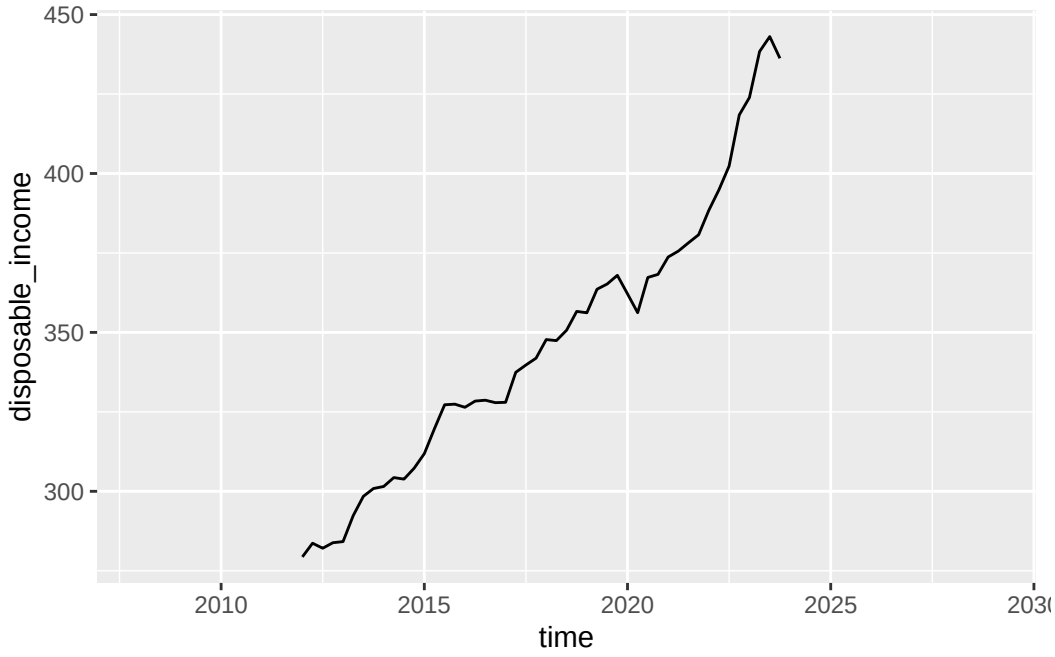
Choose an appropriate visualisation to investigate the change in household disposable income between 2012 and 2024. Comment on your findings.

Solution

An appropriate visualisation to investigate how a continuous variable changes across time is a line graph. This can be created using the `geom_line` function and requires the disposable income on the y-axis and a time variable on the x. The dataset does not currently contain an appropriate time variable that takes account of both the year and the quarter. Therefore, we will need to create one before we generate the plot:

```
obr_data %>%
  mutate(time = year + ((quarter - 1) * .25)) %>%                    ①
  ggplot() +                                                         ②
  geom_line(aes(x = time, y = disposable_income))
```

① Create a time variable that takes the value of the `year` for the first `quarter`, and then increases by .25 for each subsequent `quarter`
② The data has already been specified by piping it into the `ggplot` function



Disposable income has increased over the past 12 years, with small decreases around 2020 and in the first period of 2024.

G Exercise 8 solution

G.1 Question

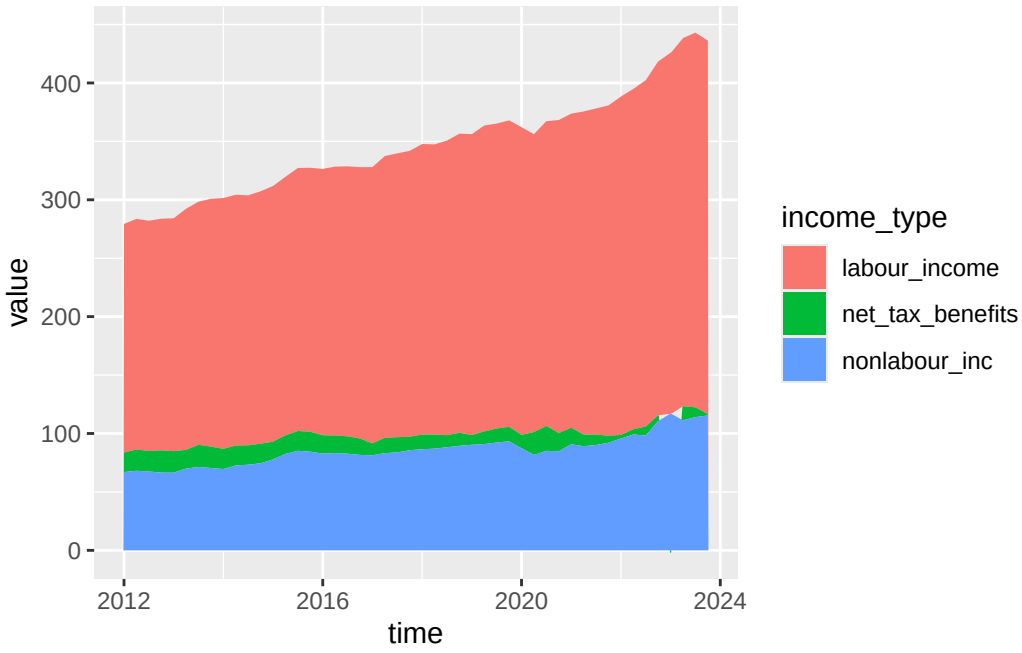
Disposable income is the sum of labour income, non-labour income, and net taxes and benefits. Decompose the time series created in Exercise 7 to show how the contribution of these elements to total disposable income varied across time.

Solution

A stacked area chart (generated using `geom_area`) would allow us to visualise the total disposable income, broken down into the different income streams. The stacked area chart is created by layering each income value on top of one another, using a categorical variable representing income type to determine the area `fill`. This layout will require us to `pivot` the data to convert it into a long format.

```
obr_data %>%
  mutate(time = year + ((quarter - 1) * .25)) %>%                    ①
  select(time, labour_income, nonlabour_inc,                         ②
         net_tax_benefits) %>%
  pivot_longer(cols = -time,                                         ③
               names_to = "income_type",                             ④
               names_transform = list(income_type = as.factor),      ⑤
               values_to = "value") %>%                              ⑥
  ggplot() +
  geom_area(aes(x = time, y = value, fill = income_type))
```

① Create a `time` variable for the x-axis that accounts for both year and quarter
② Select variables required for the visualisation
③ Pivot all variables apart from `time`
④ Use the current variable names to create a new variable named `income_type`
⑤ Convert the new `income_type` variable to `factor` (by default, R treats variable names as `character`)
⑥ Use the current income values to create a new variable named `values`


400
300
200
100
0
value
2012
2016
2020
2024
time
income_type
labour_income
net_tax_benefits
nonlabour_inc

H Exercise 9 solution

H.1 Question

Return to the visualisation showing the change in disposable income over time, decomposed into different income streams (labour, non-labour, and taxes and benefits). Adapt the visualisation to ensure it is accessible, compelling, and clear. This can include:

- Adding an appropriate title and caption with data source information
- Adapting the colour scheme to make the differences more obvious
- Adding labels to commmunicate important findings to the reader
- Adjusting the `theme` to ensure text is large enough, values are clear but not overwhelmed by 'chart junk'

Save this visualisation and add it into a Word document, along with a brief interpretation of the visualisation.

Solution

There is no correct answer to this exercise, the solution provided here is a suggestion based on data visualisation principles covered in this chapter and my personal preference. I have added a title to make the visualisation clearer and a footnote with data source information. I have added the percentage of disposable income to each segment in 2012 and the last quarter of 2023 to make the changes more obvious.

The colours I have chosen to represent each income type cone from the R colour brewer palette `Dark2` which is colour blind friendly and ensures the sections are distinct. Rather than including a legend, I have chosen to change the colour of the text in the subtitle to align with the income type. This is done using the `ggtext` package. You are not expected to know this but it is included as an idea for future work For more information and example code, visit the package website.

```
pacman::p_load(ggtext)                                                    ①

disp_income_long <- obr_data %>%                                          ②
  mutate(time = year + ((quarter - 1) * .25)) %>%
```

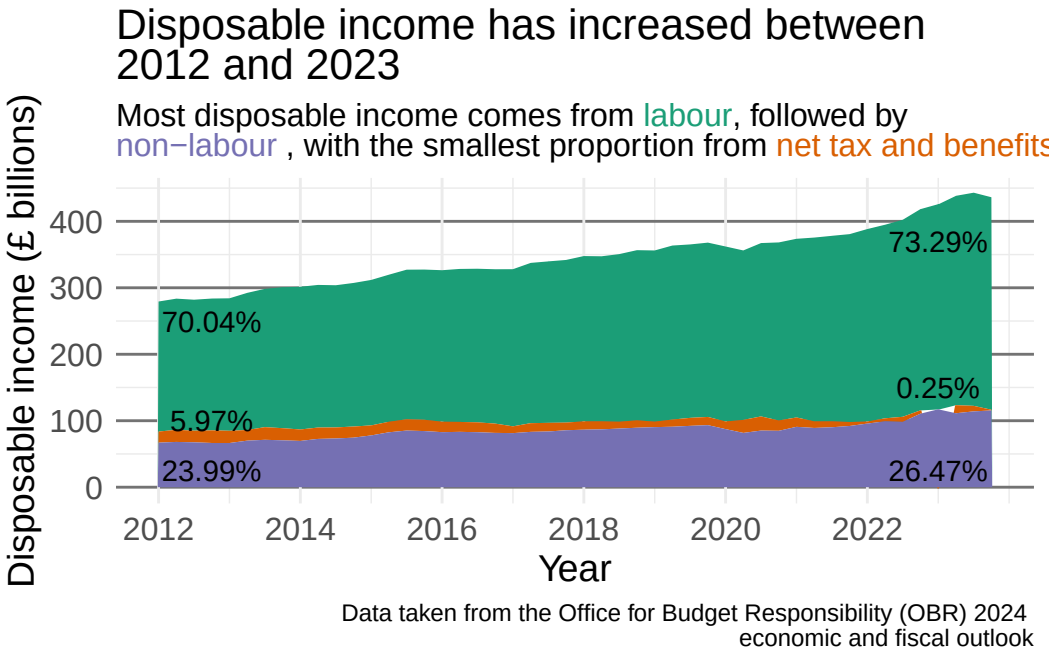
```
  select(time, labour_income, nonlabour_inc,
         net_tax_benefits) %>%
  pivot_longer(cols = -time,
               names_to = "income_type",
               names_transform = list(income_type = as.factor),
               values_to = "value")

perc_income <- disp_income_long %>%                                    ③
  filter(time == 2012 | time == 2023.75) %>%
  group_by(time) %>%
  mutate(perc_income = (value / sum(value)) * 100,
         perc_clean = paste0(round(perc_income, 2), "%")) %>%
  ungroup()

ggplot(disp_income_long) +
  geom_area(aes(x = time, y = value, fill = income_type)) +
  scale_fill_brewer(palette = "Dark2", guide = "none") +               ④
  scale_x_continuous(breaks = seq(2012, 2022, by = 2)) +               ⑤
  labs(title = "Disposable income has increased between <br> 2012 and 2023", ⑥
       subtitle = "Most disposable income comes from <span style = 'color:#1b9e77;'>labour</s
       followed by <br> <span style = 'color:#7570b3;'>non-labour</span>
       , with the smallest proportion from <span style = 'color:#d95f02;'>net tax and benefi
       x = "Year", y = "Disposable income (£ billions)",
       caption = "Data taken from the Office for Budget Responsibility (OBR) 2024 \n economi
  annotate("text", x = 2012.75, y = 250,                               ⑧
           label = filter(perc_income,
                          time == 2012,
                          income_type == "labour_income")$perc_clean) +
  annotate("text", x = 2012.75, y = 25,
           label = filter(perc_income,
                          time == 2012,
                          income_type == "nonlabour_inc")$perc_clean) +
  annotate("text", x = 2012.75, y = 100,
           label = filter(perc_income,
                          time == 2012,
                          income_type == "net_tax_benefits")$perc_clean) +
  annotate("text", x = 2023, y = 370,
           label = filter(perc_income,
                          time == 2023.75,
                          income_type == "labour_income")$perc_clean) +
  annotate("text", x = 2023, y = 25,
           label = filter(perc_income,
```

```
                        time == 2023.75,
                        income_type == "nonlabour_inc")$perc_clean) +
  annotate("text", x = 2023, y = 150,
           label = filter(perc_income,
                        time == 2023.75,
                        income_type == "net_tax_benefits")$perc_clean) +
  theme_minimal() +                                                    ⑨
  theme(plot.title = element_markdown(size = 16),
        plot.subtitle = element_markdown(size = 12),
        panel.grid.major.y = element_line(colour = "grey45"),
        axis.title = element_text(size = 14),
        axis.text = element_text(size = 12))
```

① Load in the `ggtext` package (if using)
② `pivot` the original data into the wide format required for this visualisation
③ Begin by calculating the percentage of disposable income that is made up of each different source. Create a 'tidy' version of this which can be added to the visualisation
④ Use the `Dark2` palette for the area fills, determined by income type, and remove the legend (`guide = "none"`)
⑤ Adapt the x-axis labels to show every 2 years
⑥ Add an informative title
⑦ Add a subtitle, showing each income type in the colour it is represented in the graph (this code is adapted from the `ggtext` webpage) as an alternative to a legend
⑧ Add a text label showing the percentage of disposable income made up by each income type at the start and end of the period (the label is taken from the percentage we calculated above)
⑨ Check that titles and text are large enough, make the y-axis gridlines bolder and easier to interpret.


Disposable income has increased between 2012 and 2023
Most disposable income comes from labour, followed by non−labour , with the smallest proportion from net tax and benefits
Disposable income (£ billions)
400
300
200
100
0
2012
2014
2016
2018
2020
2022
Year
73.29%
70.04%
0.25%
5.97%
23.99%
26.47%
Data taken from the Office for Budget Responsibility (OBR) 2024 economic and fiscal outlook