

Introduction to R with Tidyverse

Sophie Lee

2024-08-08

Table of contents

Welcome!	5
Data used for the course	5
License	5
1 Introduction to R and RStudio	6
1.1 The RStudio interface	6
Exercise 1	8
1.2 R syntax	9
Exercise 2	10
1.3 R objects and functions	10
1.3.1 Objects	10
1.3.2 Functions	11
1.3.3 Help files	11
1.3.4 Error and warning messages	12
1.3.5 Cleaning the environment	13
1.3.6 R packages	13
2 Introduction to tidyverse and data wrangling	15
2.1 Opening and exploring data	15
2.1.1 Styles of R coding	15
2.1.2 The working directory	16
2.2 Data input	17
2.2.1 Selecting variables	20
2.2.2 Filtering data	23
2.2.3 Pipes	24
2.2.4 Creating new variables	26
Exercise 3	27
3 Data preparation and manipulation	29
3.1 Data wrangling and summarising	29
3.1.1 Combining two datasets	29
3.1.2 Joining multiple datasets	30
3.1.3 Transforming data	31
3.1.4 Summary tables	33
Exercise 4	35

4	Data visualisation with ggplot2	36
4.1	Data visualisation with ggplot2	36
4.1.1	Choosing the most appropriate visualisation	36
4.1.2	The ggplot2 package	37
	Exercise 5	40
4.1.3	Customising visualisations	40
	Exercise 6	43
4.1.4	Scale functions	43
4.1.5	Other labelling functions	47
4.1.6	Theme functions	48
4.1.7	Facet functions	50
	Exercise 7	51
5	Reproducible research with RMarkdown	52
5.1	Introduction to RMarkdown	52
5.1.1	Creating an RMarkdown files	52
5.1.2	Rmarkdown content	53
5.1.3	Compiling RMarkdown documents	56
5.1.4	Data visualisation in RMarkdown	57
	Exercise 8	61
	Appendices	62
	Data description	62
	What is ‘CSP’?	62
	Descriptions of variables	62
	Identifier variables	62
	Regions of England	62
	Settlement Funding Assessment (SFA)	63
	Under-indexing business rate multipliers	63
	Council tax	63
	New Homes Bonus	63
	Rural Services Delivery Grant	63
	Exercise solutions	64
	Exercise 1	64
	Solutions	64
	Exercise 2	64
	Solutions	64
	Exercise 3	65
	Solutions	65
	Exercise 4	67

Solutions	67
Exercise 5	69
Solutions	69
Exercise 6	70
Solutions	71
Exercise 7	73
Solution	73
Exercise 8	74
Solutions	75

Welcome!

Welcome to the course materials for the **Introduction to R with Tidyverse** course.

This course is designed to equip you with the essential skills to leverage the power of R and Tidyverse for their work. The course begins with a gentle introduction to the user-friendly RStudio interface and the basics of the R coding language, or syntax. This makes it ideal for anyone with little or no prior coding experience, or those looking for a refresher of the basics.

Through this course, you will learn how to manipulate, transform, and clean data efficiently, and how to create compelling visualisations to communicate your findings effectively. Throughout the course, we will discuss best practices for reproducible coding.

Throughout these notes, you will also see boxes like this containing ‘style tips’. These ensure that your code follows the [Tidyverse style guide](#), making it as consistent and readable as possible.

Data used for the course

The examples and exercises in these materials are based on real world data. This data relates to the Core Spending Power (CSP) of English local authorities between 2015 and 2020.

Data for this course can be downloaded from the **data** folder of this course’s [repository](#).

For more information about this data, including variable descriptions and sources, see [the appendix](#).

License

I believe that science should not be behind a paywall, that is why these materials are available for free online, licensed under a [CC BY-SA licence](#).

1 Introduction to R and RStudio

1.1 The RStudio interface

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 via the [CRAN](#) website.

Rstudio is also free to download from the [Posit](#) website.

1.1.0.1 The RStudio console window

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

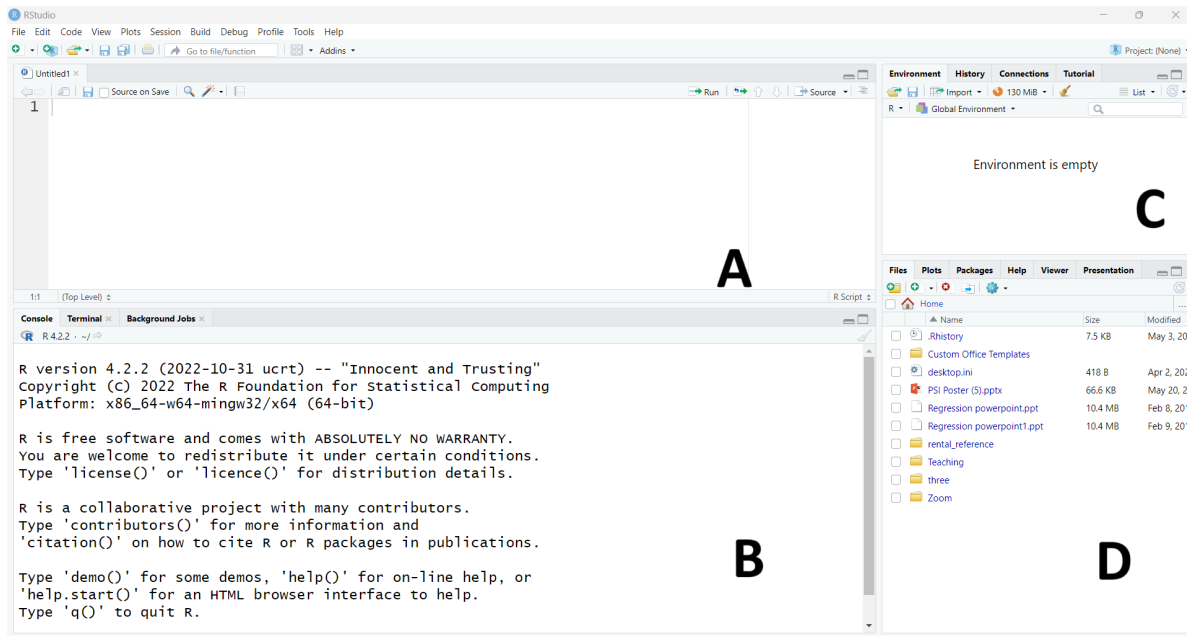



Figure 1.1: RStudio interface


Window A: R script files

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


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* 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 from the script, highlight the code 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). More than one command can be run at the same time by highlighting all of them.


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, or click the  icon at the top of the window. It is important to save the script files at regular intervals to avoid losing work. Once the script file has been saved, we can also use the keyboard shortcuts **Ctrl + s** on Windows and **Command + s** on Mac to save the latest script file.

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 or by clicking the  icon and selecting a `.R` file. The keyboard shortcut to open an existing script file is **Ctrl + o** on Windows, and **Command + o** on Macs.

Window B: The 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 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.

Window C: Environment and history

This window lists all data and objects currently loaded into R. More details on the types of objects and how to use the Environment window are given in later sections.

Window D: Files, plots, packages and help

This window has many potential uses: graphics 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.

1.2 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.

Combining `#` and `-` creates sections within a script file, making them easier to navigate and organise.

For example:

```
# Load data -----  
  
# Tidy data -----
```

Helpful hint

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

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

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

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

Although R will work whether a space is added before/after a mathematical operator, the [style guide](#) recommends to add them surrounding most mathematical operations (+, -, *, /), but not around ^.

For example:

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

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

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 calculate the following calculations. Add the result to the same line of the script file in a way that ensures there are no errors in the code.
 - a. 64^2
 - b. $3432 \div 8$
 - c. 96×72

When you have finished this exercise, select the entire script file (using **ctrl + a** on windows or **Command + a** on Mac) and run it to ensure there are no errors in the code.

1.3 R objects and functions

1.3.1 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.

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

Objects are defined in R using the `<-` or `=` symbols. 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 earlier).

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

object_1 * 2
## [1] 162
```

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

```
pi
## [1] 3.141593
```

The `[1]` that appears at the beginning of each output line indicates that this is the first element in the object. If there were two lines then the second line would start with the number of that element in square brackets.

For example, if we had an object with 6 elements and when called the first line contained the first 5 elements, each line would begin with `[1]` and `[6]` respectively.

1.3.2 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**.

1.3.3 Help files

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

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

Help files contain the following information:

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

1.3.4 Error and warning messages

Where a function or object has not been correctly specified, or there 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(object_1)
## Error in eval(expr, envir, enclos): object 'object_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.

When working within the R console, if an incomplete command is run, a `+` symbol will appear in the console, rather than the usual `>`. This indicates that R expects you to keep writing

the previous code. To overcome this issue, either finish the command on the next line of the console, or press the **esc** button on your keyboard to start from scratch.

One of the benefits of using RStudio rather than the basic R package is that it will suggest object or function names after typing the first few letters. This avoids spelling mistakes and accidental errors when running code. To accept the suggestion, either click the correct choice with your mouse or use the **tab** button on your keyboard.

1.3.5 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.

1.3.6 R 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.

1.3.6.1 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.

1.3.6.2 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.

Add your `library` function at the beginning of your script file. This reminds you to reload packages when opening a new R session, and reduces the chance of error messages from functions requiring these packages.

2 Introduction to tidyverse and data wrangling

2.1 Opening and exploring data

2.1.1 Styles of R coding

Up to this point, 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.

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:

```
library(tidyverse)
```

Style tip

Begin every script file with the `library` command, loading packages in before any data. This avoids any potential errors arising where functions are called before the necessary package has been loaded into the current session.

2.1.2 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 these issues is to create an R project.

2.1.2.1 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  **Project: (None)** 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 want to create a new directory or use 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.

2.2 Data input

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

Before loading files in R, it is essential to check that they are correctly formatted. Data files should only contain one sheet with no pictures or graphics, each row should correspond to a case or observation and each column should correspond to a variable.

To avoid any errors arising from spelling mistakes, we can use the `list.files` function to return 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 must add the argument `path =` to specify the folder we want to list files from.

```
list.files(path = "data")
## [1] "CSP_2015.csv"      "CSP_2016.csv"      "CSP_2017.csv"
## [4] "CSP_2018.csv"      "CSP_2019.csv"      "CSP_2020.csv"
## [7] "CSP_long_201520.csv" "data_description.pdf"
```

This list should contain 6 CSV files with the core spending power in local authorities in England between 2015 and 2020. We will first load and explore the 2020 data using the `read_csv` function and attaching the data to an object. Remember to add the `data` folder to the file name.

```
csp_2020 <- read_csv("data/CSP_2020.csv")
```

Imported datasets will appear in the Environment window of the console once they are saved as objects. This Environment also displays the number of variables and observations in each object. To preview the contents of an object, click on its name in the Environment window or use the function `View(data)`.

Other useful functions that help explore a dataset include:

```
# Return variable names from a dataset object
names(csp_2020)
## [1] "ons_code"          "authority"          "region"             "sfa_2020"
## [5] "under_index_2020" "ct_total_2020"      "nhb_2020"           "nhb_return_2020"
## [9] "rsdg_2020"
```

Style tip

Variable names should follow the same style rules as object names: only contain lower case letters, numbers, and use `_` to separate words. They should be meaningful and concise.

```
# Display information about the structure of an object
str(csp_2020)
## spc_tbl_ [396 x 9] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ ons_code      : chr [1:396] "E07000223" "E07000026" "E07000032" "E07000224" ...
## $ authority     : chr [1:396] "Adur" "Allerdale" "Amber Valley" "Arun" ...
## $ region        : chr [1:396] "SE" "NW" "EM" "SE" ...
## $ sfa_2020      : num [1:396] 1.77 3.85 3.23 3.67 4.08 ...
## $ under_index_2020: num [1:396] 0.0708 0.1465 0.1292 0.147 0.1557 ...
## $ ct_total_2020  : num [1:396] 6.53 5.4 6.85 11.61 6.42 ...
## $ nhb_2020      : num [1:396] 0.0881 0.6061 1.5786 2.2949 1.1547 ...
## $ nhb_return_2020: num [1:396] 0 0 0 0 0 0 0 0 0 ...
## $ rsdg_2020     : num [1:396] 0 0.326 0 0 0 ...
## - attr(*, "spec")=
## .. cols(
## ..   ons_code = col_character(),
## ..   authority = col_character(),
## ..   region = col_character(),
## ..   sfa_2020 = col_double(),
## ..   under_index_2020 = col_double(),
## ..   ct_total_2020 = col_double(),
## ..   nhb_2020 = col_double(),
```

```
## .. nhb_return_2020 = col_double(),
## .. rsdg_2020 = col_double()
## .. )
## - attr(*, "problems")=<externalptr>
```

Output from the `str` function differs depending on the type of object it is applied to. For example, this object is a `tibble` (`tbl`, Tidyverse's name for a dataset). The information given about tibbles includes the object dimensions (396 x 9, or 396 rows and 9 columns), variable names, and variable types.

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 session.

```
# Return the first 6 rows of the tibble
head(csp_2020)
## # A tibble: 6 x 9
##   ons_code authority region sfa_2020 under_index_2020 ct_total_2020 nhb_2020
##   <chr>      <chr>   <chr>   <dbl>         <dbl>         <dbl>     <dbl>
## 1 E07000223 Adur      SE      1.77         0.0708         6.53     0.0881
## 2 E07000026 Allerdale NW       3.85         0.147         5.40     0.606
## 3 E07000032 Amber Valley EM       3.23         0.129         6.85     1.58
## 4 E07000224 Arun      SE      3.67         0.147        11.6     2.29
## 5 E07000170 Ashfield  EM      4.08         0.156         6.42     1.15
## 6 E07000105 Ashford   SE      2.88         0.115         7.92     3.05
## # i 2 more variables: nhb_return_2020 <dbl>, rsdg_2020 <dbl>
```

```
# Return the final 6 rows of the tibble.
tail(csp_2020)
## # A tibble: 6 x 9
##   ons_code authority region sfa_2020 under_index_2020 ct_total_2020 nhb_2020
##   <chr>      <chr>   <chr>   <dbl>         <dbl>         <dbl>     <dbl>
## 1 E07000229 Worthing SE      2.69         0.108         9.52     0.961
## 2 E07000238 Wychavon WM       2.65         0.106         6.29     4.73
## 3 E07000007 Wycombe SE       0          0            0         0
## 4 E07000128 Wyre      NW      3.41         0.137         7.64     1.28
## 5 E07000239 Wyre Forest WM      2.84         0.114         7.45     0.262
## 6 E06000014 York      YH      27.1         1.06        93.8     2.68
## # i 2 more variables: nhb_return_2020 <dbl>, rsdg_2020 <dbl>
```

2.2.1 Selecting variables

Often, our analysis will not require every variable in a downloaded dataset, 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, if we wanted to return the new homes bonus (`nhb`) for each local authority (the seventh column of the dataset), we can either `select` this based on the variable name or its location in the object:

```
# Return the nhb_2020 variable from the csp_2020 object
select(csp_2020, nhb_2020)
## # A tibble: 396 x 1
##   nhb_2020
##   <dbl>
## 1  0.0881
## 2  0.606
## 3  1.58
## 4  2.29
## 5  1.15
## 6  3.05
## 7  0
## 8  0
## 9  1.05
## 10 1.85
## # i 386 more rows

# Return the 7th variable of the csp_2020 object
select(csp_2020, 7)
## # A tibble: 396 x 1
##   nhb_2020
##   <dbl>
## 1  0.0881
## 2  0.606
## 3  1.58
## 4  2.29
## 5  1.15
## 6  3.05
## 7  0
## 8  0
```

```
## 9 1.05
## 10 1.85
## # i 386 more rows
```

We can select multiple variables and return them as a tibble by separating the variable names or numbers with commas:

```
# Return three variables from the csp_2020 object
select(csp_2020, ons_code, authority, region)
## # A tibble: 396 x 3
##   ons_code authority      region
##   <chr>    <chr>      <chr>
## 1 E07000223 Adur        SE
## 2 E07000026 Allerdale   NW
## 3 E07000032 Amber Valley EM
## 4 E07000224 Arun        SE
## 5 E07000170 Ashfield    EM
## 6 E07000105 Ashford     SE
## 7 E31000001 Avon Fire   SW
## 8 E07000004 Aylesbury Vale SE
## 9 E07000200 Babergh     EE
## 10 E09000002 Barking And Dagenham L
## # i 386 more rows
```

When selecting consecutive variables, a shortcut can be used that gives the first and last variable in the list, separated by a colon, `:`. The previous example can be carried out using the following code:

```
# Return variables from ons_code up to and including region
select(csp_2020, ons_code:region)
## # A tibble: 396 x 3
##   ons_code authority      region
##   <chr>    <chr>      <chr>
## 1 E07000223 Adur        SE
## 2 E07000026 Allerdale   NW
## 3 E07000032 Amber Valley EM
## 4 E07000224 Arun        SE
## 5 E07000170 Ashfield    EM
## 6 E07000105 Ashford     SE
## 7 E31000001 Avon Fire   SW
## 8 E07000004 Aylesbury Vale SE
## 9 E07000200 Babergh     EE
```

```
## 10 E09000002 Barking And Dagenham L
## # i 386 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:

```
# Remove the ons_code variable
select(csp_2020, -ons_code)
## # A tibble: 396 x 8
##   authority      region sfa_2020 under_index_2020 ct_total_2020 nhb_2020
##   <chr>         <chr>    <dbl>         <dbl>         <dbl>    <dbl>
## 1 Adur          SE        1.77         0.0708         6.53    0.0881
## 2 Allerdale     NW        3.85         0.147          5.40    0.606
## 3 Amber Valley  EM        3.23         0.129          6.85    1.58
## 4 Arun          SE        3.67         0.147          11.6    2.29
## 5 Ashfield      EM        4.08         0.156          6.42    1.15
## 6 Ashford       SE        2.88         0.115          7.92    3.05
## 7 Avon Fire     SW       16.0         0.437          27.8    0
## 8 Aylesbury Vale SE         0           0              0        0
## 9 Babergh       EE        2.14         0.0857         5.77    1.05
## 10 Barking And Dagenham L 75.7        2.31         65.8    1.85
## # i 386 more rows
## # i 2 more variables: nhb_return_2020 <dbl>, rsdg_2020 <dbl>
```

The `select` function returns the variable(s) in the form of a tibble (or dataset). However, some functions, such as basic summary functions, require data to be entered as a **vector** (a list of values). Tibbles with a single variable can be converted into a vector using the `as.vector` function, or we can use the base R approach to selecting a single variable. To return a single variable as a vector in base R, we can use the `$` symbol between the data name and the variable to return:

```
csp_2020$nhb_2020
```

It is important to save any changes made to the existing dataset. This can be done using the `write_csv` function:

```
write_csv(csp_2020, file = "data/csp_2020_new.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 raw data in case of any errors or issues.

2.2.2 Filtering data

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

A more comprehensive list of conditional statements can be found in the help file using `?filter`.

For example, to return the core spending power for local authorities in the North West region of England, we use the following:

```
# Return rows where region is equal to NW from the csp_2020 object
filter(csp_2020, region == "NW")
## # A tibble: 46 x 9
##   ons_code authority region sfa_2020 under_index_2020 ct_total_2020 nhb_2020
##   <chr>      <chr>   <chr>   <dbl>         <dbl>         <dbl>     <dbl>
## 1 E07000026 Allerdale NW        3.85         0.147         5.40     0.606
## 2 E07000027 Barrow-in-~ NW        4.40         0.125         4.74     0.0111
## 3 E06000008 Blackburn ~ NW       58.1         1.79         55.9     0.999
## 4 E06000009 Blackpool NW       63.3         1.94         60.1     0.266
## 5 E08000001 Bolton NW       84.2         2.73        115.     0.506
## 6 E07000117 Burnley NW       5.90         0.171         7.16     0.694
## 7 E08000002 Bury NW       42.3         1.44         89.0     0.458
## 8 E07000028 Carlisle NW       3.34         0.134         7.49     1.49
## 9 E06000049 Cheshire E~ NW       42.5         1.70        230.    11.2
```

```
## 10 E31000006 Cheshire F~ NW      13.5      0.380      30.1      0
## # i 36 more rows
## # i 2 more variables: nhb_return_2020 <dbl>, rsdg_2020 <dbl>
```

Multiple conditional statements can be added to the same function by separating them with a comma `,`. For example, to return a subgroup of local authorities in the North West region that had a settlement funding assessment (SFA) of over £40 million, we use the following:

```
filter(csp_2020, region == "NW", sfa_2020 > 40)
## # A tibble: 23 x 9
##   ons_code authority region sfa_2020 under_index_2020 ct_total_2020 nhb_2020
##   <chr>    <chr>    <chr>    <dbl>         <dbl>         <dbl>         <dbl>
## 1 E06000008 Blackburn ~ NW      58.1          1.79          55.9          0.999
## 2 E06000009 Blackpool  NW      63.3          1.94          60.1          0.266
## 3 E08000001 Bolton      NW      84.2          2.73         115.          0.506
## 4 E08000002 Bury        NW      42.3          1.44          89.0          0.458
## 5 E06000049 Cheshire E~ NW      42.5          1.70         230.          11.2
## 6 E06000050 Cheshire W~ NW      56.3          2.12         196.          10.2
## 7 E10000006 Cumbria     NW     107.          3.56         248.          0.824
## 8 E47000001 Greater Ma~ NW      50.6          1.28          50.5          0
## 9 E06000006 Halton      NW      45.6          1.45          52.2          2.21
## 10 E08000011 Knowsley   NW      84.1          2.50          56.8          2.10
## # i 13 more rows
## # i 2 more variables: nhb_return_2020 <dbl>, rsdg_2020 <dbl>
```

2.2.3 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 multiple functions into a single process by using the ‘pipe’ symbol `%>%`, which is read as ‘and then’ within the code.

Helpful hint

To save time when piping, use the keyboard shortcut `ctrl + shift + m` for Windows, and `Command + shift + m` for Mac to create a pipe.

For example, we can return a list of local authority names from the North West region:


```
# Using the csp_2020 object
csp_2020 %>%
  # Return just rows where region is equal to NW, and then
  filter(region == "NW") %>%
  # Select just the authority variable
  select(authority)
## # A tibble: 46 x 1
##   authority
##   <chr>
## 1 Allerdale
## 2 Barrow-in-Furness
## 3 Blackburn with Darwen
## 4 Blackpool
## 5 Bolton
## 6 Burnley
## 7 Bury
## 8 Carlisle
## 9 Cheshire East
## 10 Cheshire Fire
## # i 36 more rows
```

Style tips

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

The pipe can also be combined with the `filter` function to count the number of observations that lie within a subgroup:

```
# Take the csp_2020 object
csp_2020 %>%
  # Return just rows where region is equal to NW, and then
  filter(region == "NW") %>%
  # Count the number of rows
  count()
## # A tibble: 1 x 1
##       n
##   <int>
## 1    46
```

2.2.4 Creating new variables

The function `mutate` from tidyverse's `dplyr` package allows us to add new variables to a dataset, either by manually specifying them or by creating them from existing variables. We can add multiple variables within the same function, separating each with a comma ,.

For example, we can create a new variables with the squared settlement funding assessment (`sfa_2020`), and another that recodes the council tax variable (`ct_total_2020`) into a categorical variable with three levels (low: below £5 million, medium: between £5 million and £15 million, and high: above £15 million):

```
# Create a new object, csp_2020_new, starting with the object csp_2020
csp_2020_new <- csp_2020 %>%
  # Add a new variable, sfa_2020_sq, by squaring the current sfa_2020 variable
  mutate(sfa_2020_sq = sfa_2020 ^ 2,
    # Create ct_2020-cat by cutting the ct_total_2020 object
    ct_2020_cat = cut(ct_total_2020,
      # Create categories by cutting at 0, 5 and 15
      breaks = c(0, 5, 15, Inf),
      # Add labels to these new groups
      labels = c("Low", "Medium", "High"),
      # Include the lowest break point in each group
      include_lowest = TRUE))
```

Helpful hint

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

The `mutate` function is also useful for reclassifying variables when R did not correctly choose the variable type. In this example, the `region` variable is a grouping variable, but `str(csp_2020)` shows it is recognised by R as a **character**. Grouping variables in R are known as **factors**. To convert the `region` variable to a **factor**, we use the `factor` function inside `mutate`:

```
csp_2020_new <- csp_2020 %>%
  # Add a new variable, sfa_2020_sq, by squaring the current sfa_2020 variable
  mutate(sfa_2020_sq = sfa_2020 ^ 2,
    # Create ct_2020-cat by cutting the ct_total_2020 object
    ct_2020_cat = cut(ct_total_2020,
      # Create categories by cutting at 0, 5 and 15
      breaks = c(0, 5, 15, Inf),
```

```

# Add labels to these new groups
labels = c("Low", "Medium", "High"),
# Include the lowest break point in each group
include_lowest = TRUE),
region_fct = factor(region,
# To order the variable, use the levels argument
levels = c("L", "NW", "NE", "YH", "WM",
"EM", "EE", "SW", "SE")))

# Check variables are correctly classified
str(csp_2020_new)
## tibble [396 x 12] (S3: tbl_df/tbl/data.frame)
## $ ons_code      : chr [1:396] "E07000223" "E07000026" "E07000032" "E07000224" ...
## $ authority     : chr [1:396] "Adur" "Allerdale" "Amber Valley" "Arun" ...
## $ region        : chr [1:396] "SE" "NW" "EM" "SE" ...
## $ sfa_2020      : num [1:396] 1.77 3.85 3.23 3.67 4.08 ...
## $ under_index_2020: num [1:396] 0.0708 0.1465 0.1292 0.147 0.1557 ...
## $ ct_total_2020  : num [1:396] 6.53 5.4 6.85 11.61 6.42 ...
## $ nhb_2020      : num [1:396] 0.0881 0.6061 1.5786 2.2949 1.1547 ...
## $ nhb_return_2020: num [1:396] 0 0 0 0 0 0 0 0 0 0 ...
## $ rsdg_2020     : num [1:396] 0 0.326 0 0 0 ...
## $ sfa_2020_sq    : num [1:396] 3.12 14.86 10.41 13.46 16.66 ...
## $ ct_2020_cat    : Factor w/ 3 levels "Low","Medium",...: 2 2 2 2 2 2 3 NA 2 3 ...
## $ region_fct     : Factor w/ 9 levels "L","NW","NE",...: 9 2 6 9 6 9 8 9 7 1 ...

```

Although there is no real ordering to the regions in England, attaching this order allows us to control how they are displayed in outputs. By default, character variables are displayed in alphabetical order. By adding the order to this variable, we will produce output where the reference region (London) will be displayed first, followed by regions from north to south.

Exercise 3

1. How many local authorities were included in the London region?
2. Give three different ways that it would be possible to select all spend variables (`sfa_2020`, `nhb_2020`, etc.) from the `CSP_2020` dataset.
3. Create a new tibble, `em_2020`, that just includes local authorities from the East Midlands (EM) region.
 - How many local authorities in the East Midlands had an SFA of between £5 and 10 million?

- Create a new variable with the total overall spend in 2020 for local authorities in the East Midlands.

3 Data preparation and manipulation

3.1 Data wrangling and summarising

3.1.1 Combining two datasets

We may need to combine data from different files within R to perform an analysis. For example, in our case we have the core spending power for each year between 2015 and 2020. If our analysis required comparing this spending over the time period, we would need to combine these files together.

Before the data can be combined, it must be loaded into R. We will begin combining data from 2015 and 2016, then extend this to the entire period.

```
# Return a list of files to copy from the working directory
list.files(path = "data")
```

```
[1] "CSP_2015.csv"      "CSP_2016.csv"      "CSP_2017.csv"
[4] "CSP_2018.csv"      "CSP_2019.csv"      "CSP_2020.csv"
[7] "CSP_long_201520.csv" "data_description.pdf"
```

```
# Load the 2015 data and attach as an object
CSP_2015 <- read_csv("data/CSP_2015.csv")

# Load the 2016 data and attach as an object
CSP_2016 <- read_csv("data/CSP_2016.csv")
```

Next, we will combine these datasets by joining them using key variable(s) which are shared between them. In this case, each local authority has a unique identifier code (`ons_code`) and naming variable (`authority`), they also should have the same `region` listed across both datasets.

In Tidyverse, there is a family of ‘joining’ functions that combine two datasets at a time. The choice of function depends on which observations we wish to keep where the joining variables do not match between data. In this example, we expect all local authority values to be the same across years, so will use the `full_join` function.

For more information about different joining options, check the helpfile via `?full_join`.

```
# Create a new object by joining the two datasets
csp_201516 <- full_join(CSP_2015, CSP_2016,
  # List the key joining variables (in speech marks)
  by = c("ons_code", "authority", "region"))
```

3.1.2 Joining multiple datasets

R's joining functions can only be applied to two datasets at a time. To combine all 6 core spending power datasets from 2015 to 2020 in this way would require a lot of repetitive coding (which we want to avoid where necessary).

An alternative approach would be to automate this process by using **functional programming**, implemented using tidyverse's **purrr** package.

The first step of this process requires loading all csv files into R by repeatedly applying `read_csv`. This requires a list of file names from the working directory. The function `list.files` introduced earlier contains an optional argument, `pattern` which can be used to return files and folders that match a naming pattern. In this case, all csv files begin "CSP_20", so to return this list of names from the *data* folder, we use the function:

```
csp_201520 <- list.files(path = "data", pattern = "CSP_20")
```

Next, we apply `read_csv` to each element of the list of file names. The function `map` allows us to do this and return a list of tibbles. As the data lies in a folder in the working directory, we must add this file path to the file names:

```
# Return a list of files in the data folder containing CSP_20
csp_201520 <- list.files(path = "data", pattern = "CSP_20") %>%
  # Add "data/" to each of these file names
  paste0("data/", .) %>%
  # Apply read_csv to every element of the list (of file names)
  map(read_csv)
```

Finally, we require a function that apply `full_join` iteratively to the list of tibbles and reduce it to a single tibble containing core spending powers for all years. The function that does this is `reduce`:

```
# Return a list of files in the data folder containing CSP_20
csp_201520 <- list.files(path = "data", pattern = "CSP_20") %>%
  # Add "data/" to each of these file names
  paste0("data/", .) %>%
  # Apply read_csv to every element of the list (of file names)
  map(read_csv) %>%
  # Reduce the list of tibbles to a single object by iteratively joining
  reduce(full_join, by = c("ons_code", "authority", "region"))
```

3.1.3 Transforming data

The dataset containing core spending power in England between 2015 and 2020 is currently in what is known as **wide format**. This means there is a variable per measure per year, making the object very wide.

Some analyses and visualisations, particularly those used for temporal data, require a time variable in the dataset (for example, year). This requires the data to be in a different format, known as **long format**. Long format is where each row contains an observation per year (making the data much longer and narrower).

To convert data between wide and long formats, we can use the tidyverse functions `pivot_longer` and `pivot_wider`.

The first argument required by `pivot_longer` is the data we wish to transform. This is followed by the columns we wish to pivot (in this case, all variable other than the local authority codes, names, and regions). The next steps will depend on the format of data we wish to transform, format of the data we would like to generate, the values we need to include in the long dataset, and where this information will be extracted from.

For worked examples and a detailed explanation of different approaches that can be used to pivot data, access the vignette for these function by entering `vignette("pivot")` into the R console.

In the core spending power example, the new dataset will contain a row per local authority per year. A new **year** variable will be created using the suffix of the original variable names, and the prefix of the original names (e.g. **sfa**) will be retained for the new variable names.

Using a combination of the helpfile (`?pivot_longer`) and vignette, the arguments required to convert this data are **names_to**, to specify the old variable names will be used in the new data, and **names_pattern** to define how the old variable names will be separated.

```
# Create an object csp_long by pivoting csp_201520
csp_long <- pivot_longer(csp_201520,
  # Pivot columns sfa_2015 up to and including rsdg_2020
  cols = sfa_2015:rsdg_2020,
  # Separate the old variable names in two,
  # keep the prefix as it was, and put the suffix
  # into a new variable, year
  names_to = c(".value", "year"),
  # The name prefix and suffix were separated by an _,
  # the prefix can take different lengths, the suffix
  # is always the final 4 characters
  names_pattern = "(.*)_(....)")

# Check the new, long dataset's structure
str(csp_long)
```

```
tibble [2,376 x 10] (S3: tbl_df/tbl/data.frame)
 $ ons_code   : chr [1:2376] "E07000223" "E07000223" "E07000223" "E07000223" ...
 $ authority  : chr [1:2376] "Adur" "Adur" "Adur" "Adur" ...
 $ region     : chr [1:2376] "SE" "SE" "SE" "SE" ...
 $ year       : chr [1:2376] "2015" "2016" "2017" "2018" ...
 $ sfa        : num [1:2376] 3.02 2.39 1.92 1.7 1.74 ...
 $ under_index: num [1:2376] 0.0234 0.0234 0.0248 0.039 0.0567 ...
 $ ct_total   : num [1:2376] 5.47 5.68 5.85 6.08 6.35 ...
 $ nhb        : num [1:2376] 0.652 0.767 0.553 0.202 0.126 ...
 $ nhb_return : num [1:2376] 0.00523 0.00374 0.00397 0 0 ...
 $ rsdg       : num [1:2376] 0 0 0 0 0 ...
```

Notice that the new `year` variable is recognised as a character, not a numeric variable as we would like. This is because these values were taken from variable names, which R treats as characters. To fix this, we can use the `mutate` function to convert the new variable into a numeric variable.

We may also wish to calculate the total core spending power for each local authority per year to compare this over time:

```
# Create a new object based on the long data
csp_long2 <- mutate(csp_long,
  # Convert year to a numeric variable
  year = as.numeric(year),
  # Create a new total spend variable
```



```
total_spend = sfa + under_index + ct_total + nhb +  
nhb_return + rsdg)
```

After manipulating and transforming the data into the format we need for analysis and visualisation, we can save this object to reload later. Tibbles and data frame objects can be saved as CSV files using the `write_csv` function. Remember to save the data with a different name than the raw data to avoid overwriting these files.

```
write_csv(csp_long2, file = "data/CSP_long_201520.csv")
```

3.1.4 Summary tables

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 row, and the new variables must be created using a summary function.

Common examples of summary functions include:

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

For a full list of compatible summary functions, view the helpfile `?summarise`.

If we wanted to summarise the total core spending power between 2015 and 2020 across all local authorities, we can apply `summarise` to the long format data from the previous section:

```
summarise(csp_long2,  
  # Return sum of the total_spend variable  
  total_spend_all = sum(total_spend),  
  # Return the mean total spend  
  mean_total_spend = mean(total_spend),
```

```

# Return the median total spend
median_total_spend = median(total_spend),
# Return the 10th percentile (the value that 10% of the sample lies below)
quantile10_total_spend = quantile(total_spend, 0.1),
# Count the number of rows that have been summarised
total_obs = n()

```

```

# A tibble: 1 x 5
  total_spend_all mean_total_spend median_total_spend quantile10_total_spend
      <dbl>          <dbl>          <dbl>          <dbl>
1    263484.         111.         17.6           8.34
# i 1 more variable: total_obs <int>

```

The `summarise` function can be used to produce grouped summaries. This is done by first grouping the data with the `group_by` function. For example, if we wished to produce a summary table with a row per local authority, summarising the total spending between 2015 and 2020, we would use the following:

```

csp_long2 %>%
# Group by the local authority's unique identifiers
group_by(ons_code, authority) %>%
# Total spend 2015 - 2020
summarise(total_spend_all = sum(total_spend),
# Mean spend 2015 - 2020
mean_total_spend = mean(total_spend),
# Median spend 2015 - 2020
median_total_spend = median(total_spend),
# 10th percentile of total spend
quantile10_total_spend = quantile(total_spend, 0.1),
# Number of rows summarised over
total_obs = n()) %>%
# Remove grouping structure
ungroup()

```

```

# A tibble: 396 x 7
  ons_code authority total_spend_all mean_total_spend median_total_spend
  <chr>    <chr>          <dbl>          <dbl>          <dbl>
1 -      Greater London~ 12416.         2069.         2022.
2 E06000001 Hartlepool    485.          80.8          80.3
3 E06000002 Middlesbrough 711.          119.          118.
4 E06000003 Redcar And Cle~ 660.          110.          109.

```

```

5 E06000004 Stockton-on-Tees 832. 139. 138.
6 E06000005 Darlington 474. 79.0 78.5
7 E06000006 Halton 598. 99.7 99.0
8 E06000007 Warrington 806. 134. 133.
9 E06000008 Blackburn with Darwen 692. 115. 115.
10 E06000009 Blackpool 750. 125. 124.
# i 386 more rows
# i 2 more variables: quantile10_total_spend <dbl>, total_obs <int>

```

Warning

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

Exercise 4

1. Create a data frame with the minimum, maximum and median total spend per year for each region.
2. Produce a frequency table containing the number and percentage of local authorities in each region.
3. Convert the data object `csp_long2` back into wide format, with one row per local authority and one variable per total spend per year (**HINT:** start by selecting only the variables you need from the long data frame). Use the help file `?pivot_wider` and `vignette("pivot")` for more hints.
4. Using your new wide data frame, calculate the difference in total spending for each local authority between 2015 and 2020. How many local authorities have had an overall reduction in spending since 2015?

4 Data visualisation with ggplot2

4.1 Data visualisation with ggplot2

Data visualisation is a powerful tool with multiple 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.

4.1.1 Choosing the most appropriate visualisation

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

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

Figure 4.1: Table 6.1: Common visualisations by number and type of variables, with ggplot2 geom

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 but is by no means comprehensive.

4.1.2 The **ggplot2** package

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

For example, in the **csp_2020** dataset, we may wish to explore the relationship between the settlement funding assessment (**sfa_2020**) and council tax total (**ct_total_2020**) spending for each local authority. To visualise the relationship between two continuous numeric variables, a **scatterplot** would be most appropriate.

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

```
ggplot(data = csp_2020)
```

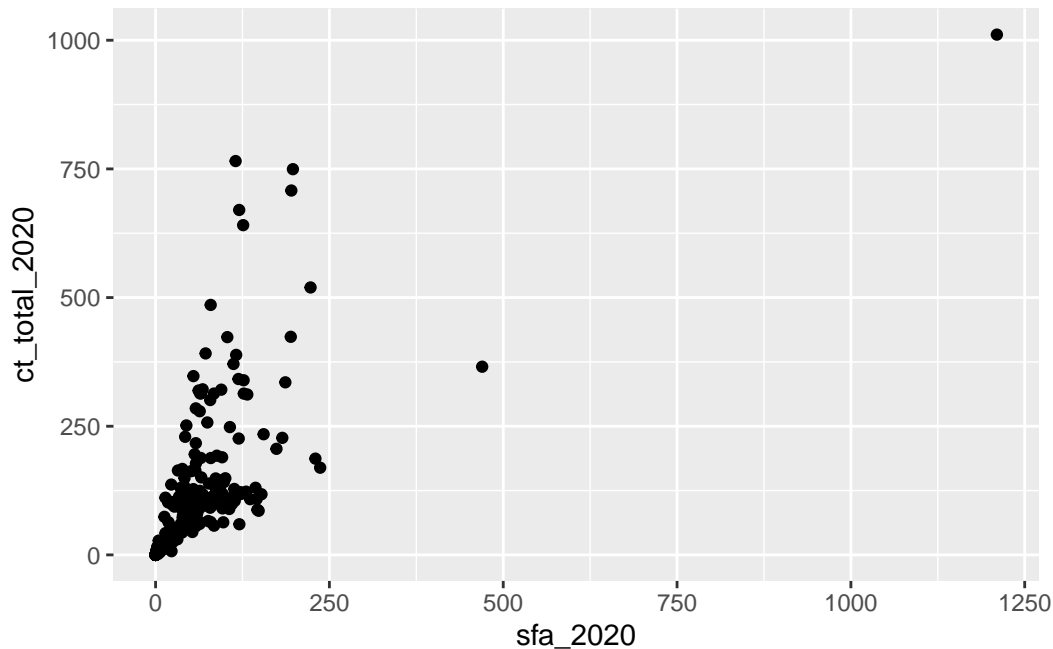
To add information to this graph, we add a **geom** layer: a visual representation of the data. There are many different geom objects built into the **ggplot2** package (begin typing **?geom** into the console to see a list). The **geom_point** function is used to create scatterplots.

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

Warning

Although **ggplot2** is a tidyverse package, it uses a different method of piping to the other packages. Use the **+** symbol to add an extra layer when working in **ggplot**.

```
# Generate the chart area and specify the data
ggplot(data = csp_2020) +
  # Add points, defined by sfa_2020 and ct_total_2020
  geom_point(mapping = aes(x = sfa_2020, y = ct_total_2020))
```

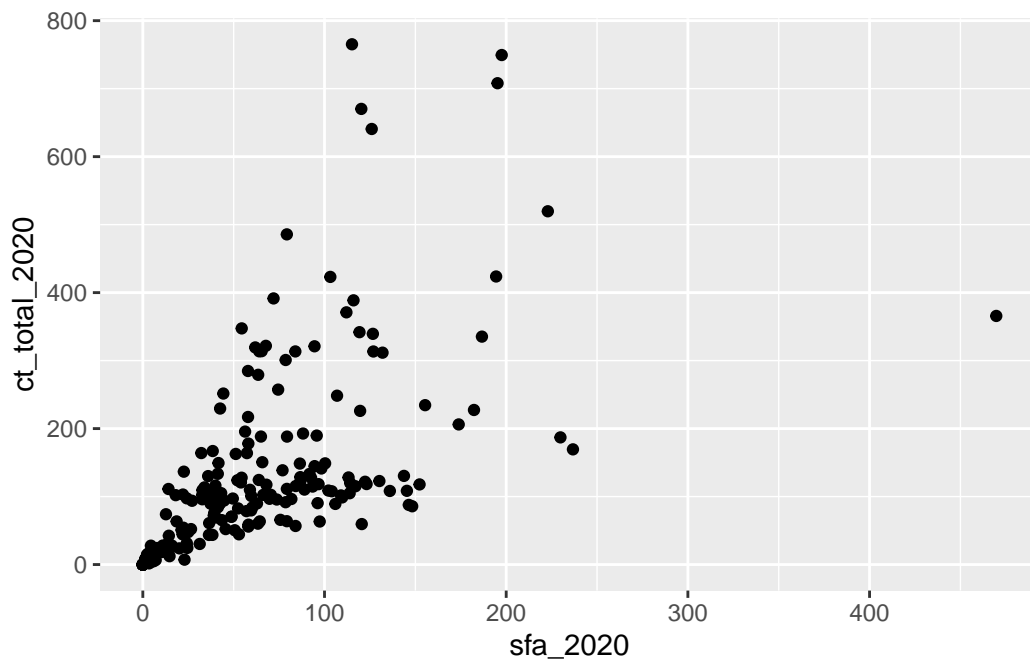


The resulting scatterplot shows a positive association between the SFA and council tax spending in English local authorities during 2020. We can identify an outlier in the top right corner of the graph. Before proceeding, we want to ensure that this observation is an outlier and not an error to be removed from the data. We can use the `filter` function to return the name of the local authority that matches these values:

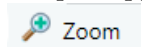
```
# Using the data csp_2020
csp_2020 %>%
  # Return just rows where sfa_2020 is over 1000, and then
  filter(sfa_2020 > 1000) %>%
  # Return the authority names
  select(authority)
## # A tibble: 1 x 1
##   authority
##   <chr>
## 1 Greater London Authority
```

This outlier is the Greater London Authority which is a combination of local authorities that are already included in the dataset. Including this observation would introduce duplicates into the analysis, and so this observation should be removed to avoid invalid results. To remove the Greater London Authority observation, we can combine the `filter` and `ggplot` functions using pipes:

```
# Take the csp_2020 data, and then
csp_2020 %>%
  # Return all rows where authority is not equal to Greater London Authority,
  # and then
  filter(authority != "Greater London Authority") %>%
  # Generate a plot
  ggplot( ) +
  # Add visual markings based on the data
  geom_point(aes(x = sfa_2020, y = ct_total_2020))
```



Graphs appear in the plot window in RStudio and can be opened in a new window using the

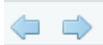


icon. Graphs in this window can also be copied and pasted into other documents using the



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:

```
# Create a new object, beginning from csp_2020, and then
sfa_ct_plot <- csp_2020 %>%
  # Return all rows where authority name is not GLA, and then
  filter(authority != "Greater London Authority") %>%
  # Create a ggplot area
  ggplot( ) +
  # Add visual markings from the data
  geom_point(aes(x = sfa_2020, y = ct_total_2020))

# Save the graph object as a png file
ggsave(sfa_ct_plot, filename = "sfa_ct_plot.png")
```

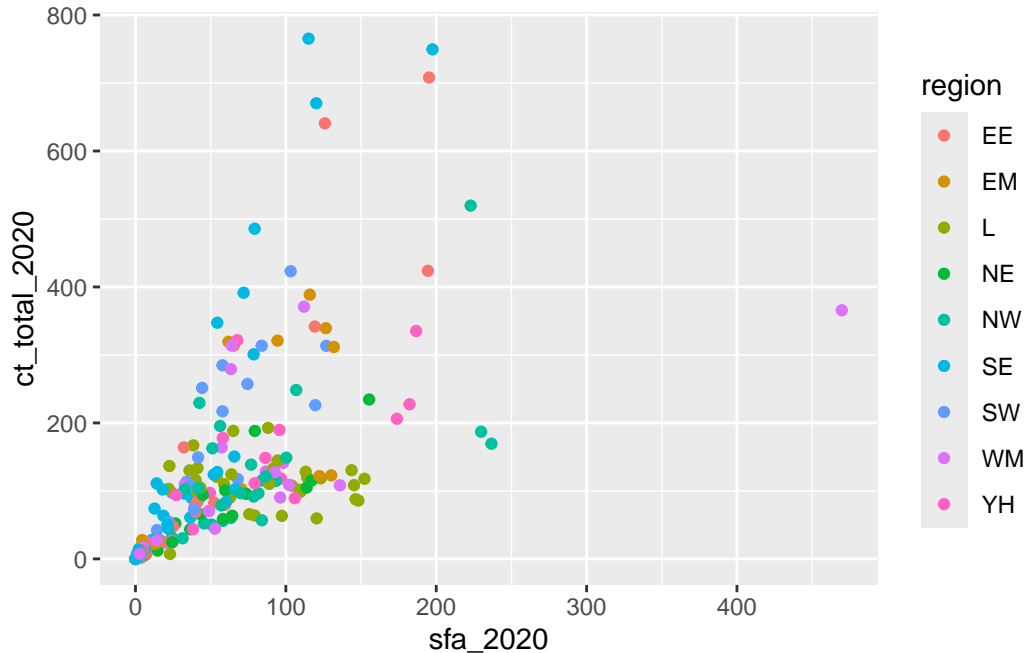
Exercise 5

1. Create a new data object containing the 2020 CSP data without the Greater London Authority observation. Name this data frame `csp_nolon_2020`.
2. Using the `csp_nolon_2020` data, create a data visualisation to check the distribution (or shape) of the SFA variable.
3. Based on the visualisation above, create a summary table for the SFA variable containing the minimum and maximum, and appropriate measures of the centre/average and spread.

4.1.3 Customising visualisations

Additional variables can be included into a visualisation within the mapping argument of a `geom` function. For example, we could explore the relationship between SFA and council tax across regions by colouring points based on the region:

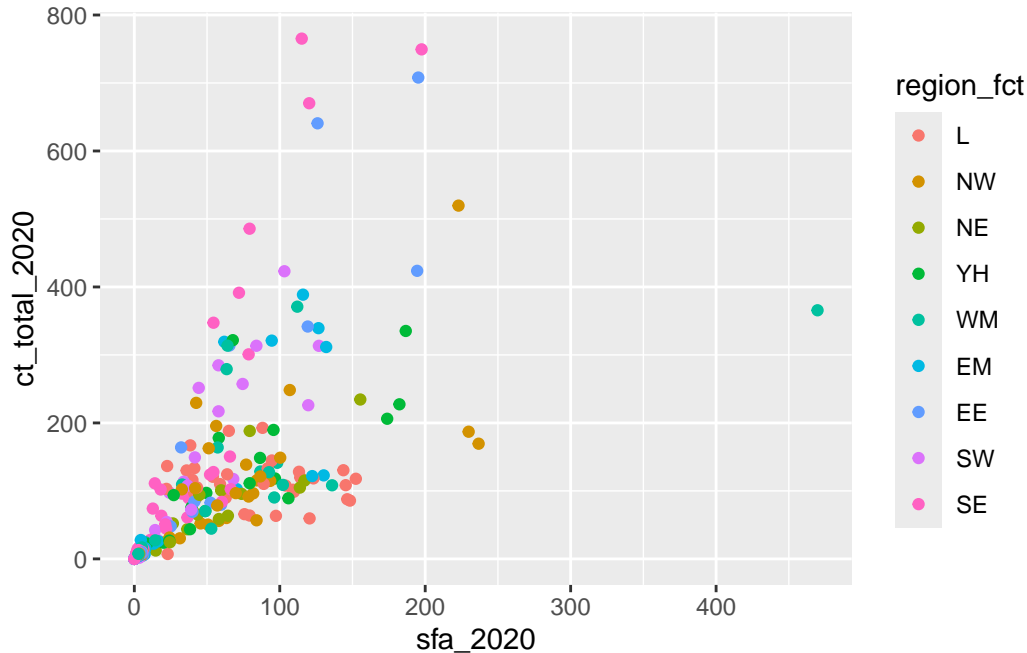
```
ggplot(data = csp_nolon_2020) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = region))
```

By default, R uses alphabetical ordering for character variables. To change this order, the variable must be converted into a **factor**. A factor is how R recognises categorical variables. For example, to order the region legend so that the London region appears first, followed by other regions from north to south, we would use the `mutate` function, combined with the `factor` function to create a new, ordered variable. The argument `levels` allows us to specify the order of categories in a factor:

```
csp_nolon_2020_new <- csp_nolon_2020 %>%
  mutate(region_fct = factor(region,
                             levels = c("L", "NW", "NE", "YH", "WM",
                                           "EM", "EE", "SW", "SE")))

ggplot(data = csp_nolon_2020_new) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = region_fct))
```



Arguments that can be adjusted within geoms include:

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

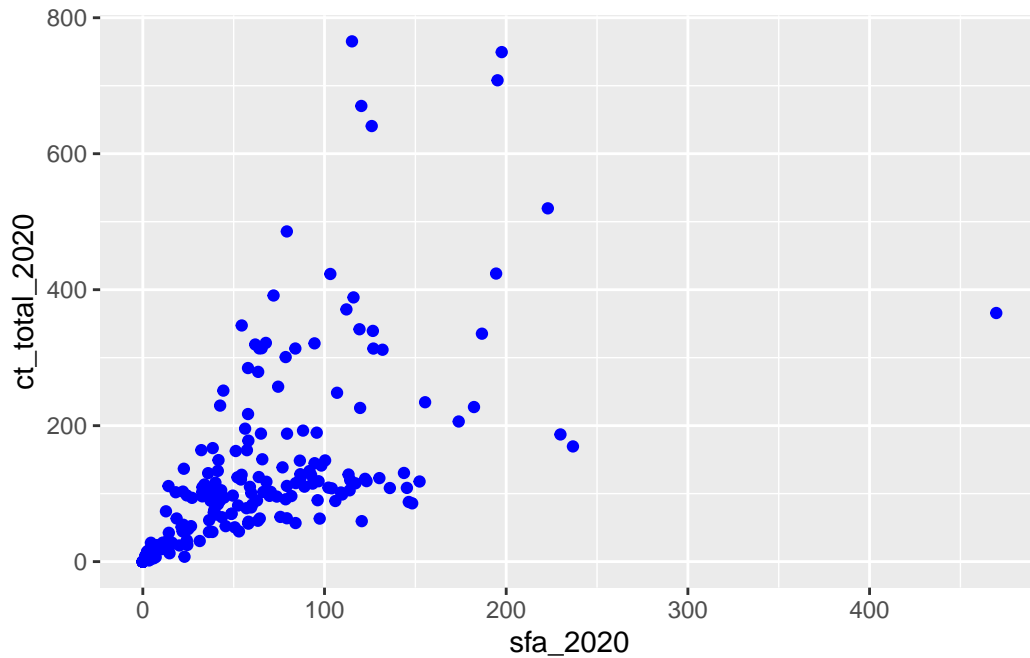
⚠ Warning

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.

Aesthetic properties of the geom object may also be set manually, outside of the `aes` function, using the same argument but with a shared value rather than a variable. For example:

```
ggplot(csp_nolon_2020_new) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020),
    # Adding the colour outside of the aes wrapper as it is not
```

```
# from the data
colour = "blue")
```



Exercise 6

1. What is the problem with the following code? Fix the code to change the shape of all the points.

```
ggplot(csp_nolon_2020) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020, shape = "*"))
```

2. Add a line of best fit to the scatterplot showing the relationship between SFA and council tax total (hint: use `?geom_smooth`).
3. Add a line of best fit for each region (hint: make each line a different colour).

4.1.4 Scale functions

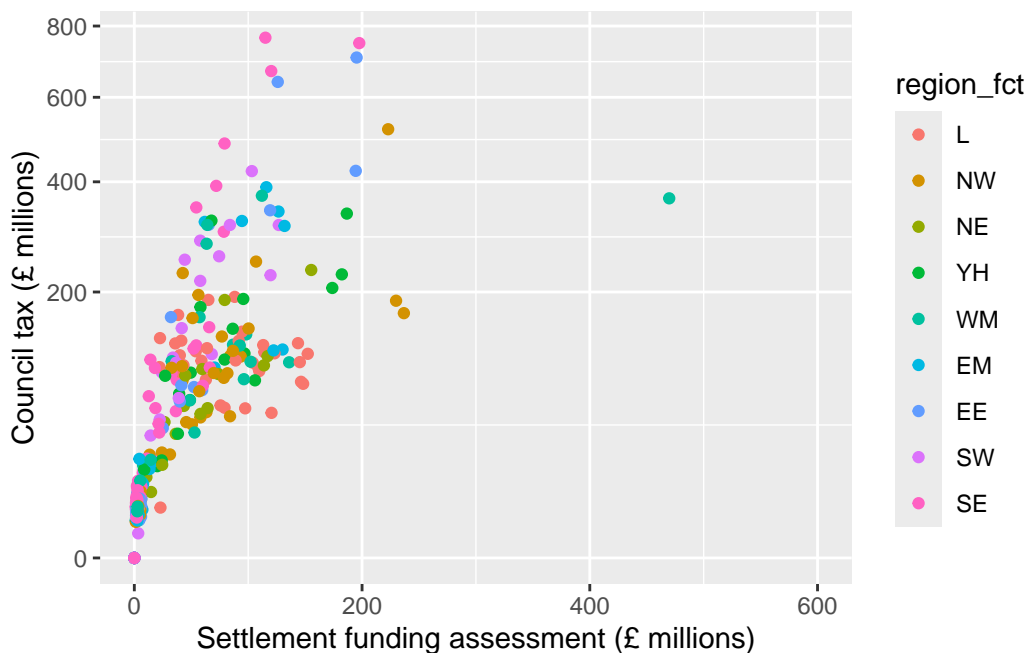
4.1.4.1 Customising axes

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'`.

For example, `scale_x_continuous` customises the x axis when the variable is continuous, and `scale_x_discrete` can be used where the variable is discrete or categorical. Arguments to customise the x or y axes include:

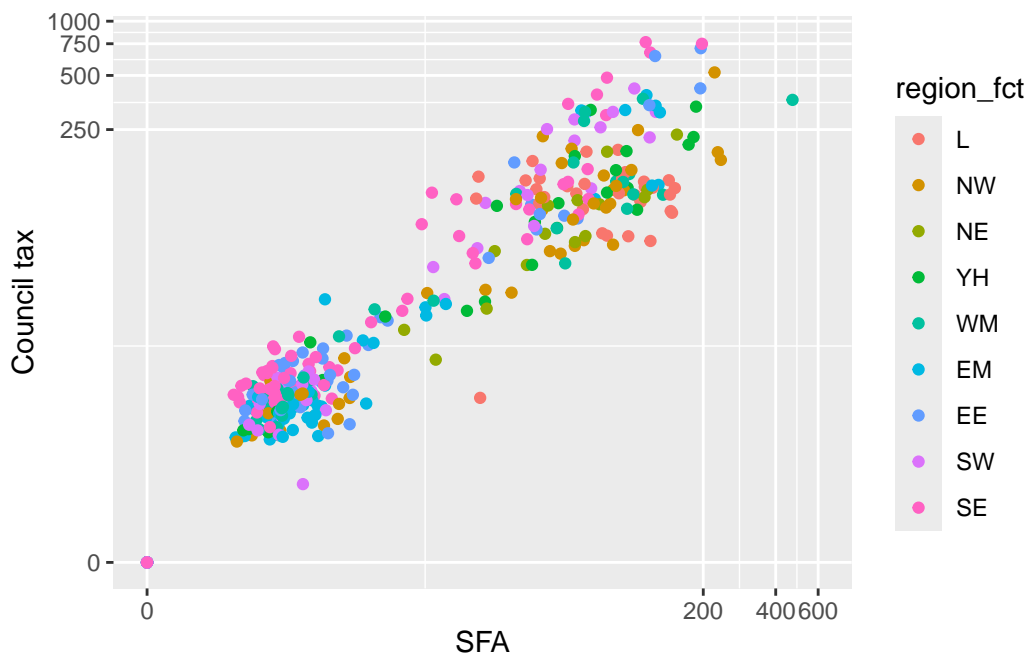
- `name` = to change the axis title
- `limits` = `c(...)` sets the axis limits
- `breaks` = `c(...)` defines tick marks
- `labels` = `c(...)` attaches labels to break values
- `trans` = transforms the scale that the axis is shown on.

```
ggplot(csp_nolon_2020_new) +  
  # Scatterplot with SFA on x, CT on y, and colour by region  
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = region_fct)) +  
  # Add title to x axis  
  scale_x_continuous(name = "Settlement funding assessment (£ millions)",  
                     # Set x axis limits from 0 to 600  
                     limits = c(0, 600),  
                     # Set tick marks every 200  
                     breaks = c(0, 200, 400, 600)) +  
  # Add title to y axis  
  scale_y_continuous(name = "Council tax (£ millions)",  
                     # Show the y axis on a square root scale  
                     trans = "sqrt")
```



A common transformation that can be useful to explore the relationship between variables which have clusters of smaller values is the logarithm (or `log`) function. Applying a `log` function to a scale increases the difference between smaller values (stretching out these clusters), while reducing the difference between the smaller values and largest ones. Log functions can only be applied to positive, non-zero numbers. Where a sample may contain zeroes, the transformation `log1p` can be applied instead which adds 1 to each value before applying the log transformation ($\log(n + 1)$):

```
ggplot(csp_nolon_2020_new) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = region_fct)) +
  scale_x_continuous(name = "SFA", limits = c(0, 600),
                     breaks = c(0, 200, 400, 600),
                     trans = "log1p") +
  scale_y_continuous(name = "Council tax",
                     trans = "log1p")
```



We can now clearly see the strong positive association between SFA and council tax spending in local authorities with lower values of this without losing any information.

4.1.4.2 Customising colour scales

There are a wide range of options for customising the colour aesthetics of geoms. These include pre-defined colour palettes, such as `scale_colour_viridis_c` for continuous variables,

or `scale_colour_viridis_d` for discrete or categorical variables. Viridis colour palettes are designed to be colourblind friendly and print well in grey scale. There are also many R packages containing colour palettes for different scenarios.

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

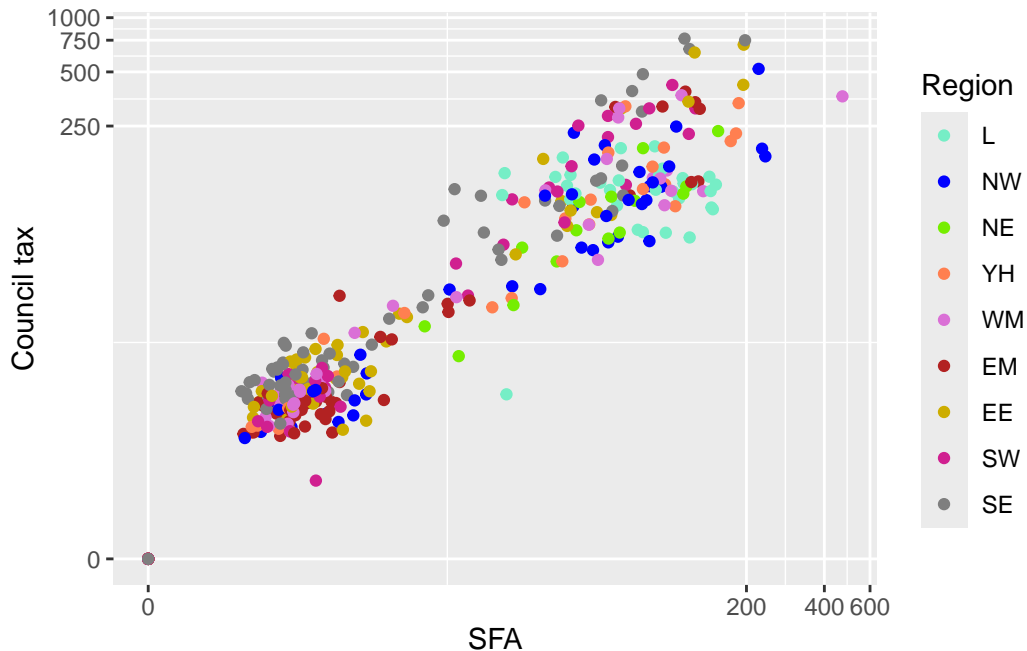
💡 Style tip

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).

Where a colour palette will be used across multiple plots, defining this list of colours as a vector and then entering this into `scale_colour_manual` will reduce repetition:

```
region_palette <- c("aquamarine2", "blue", "chartreuse2", "coral", "orchid",  
                   "firebrick", "gold3", "violetred", "grey50")  
  
ggplot(csp_nolon_2020_new) +  
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = region_fct)) +  
  scale_x_continuous(name = "SFA", trans = "log1p") +  
  scale_y_continuous(name = "Council tax", trans = "log1p") +  
  scale_colour_manual(name = "Region", values = region_palette)
```



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

4.1.5 Other labelling functions

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

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

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

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

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

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

adds a rectangle to the graph.

4.1.6 Theme functions

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

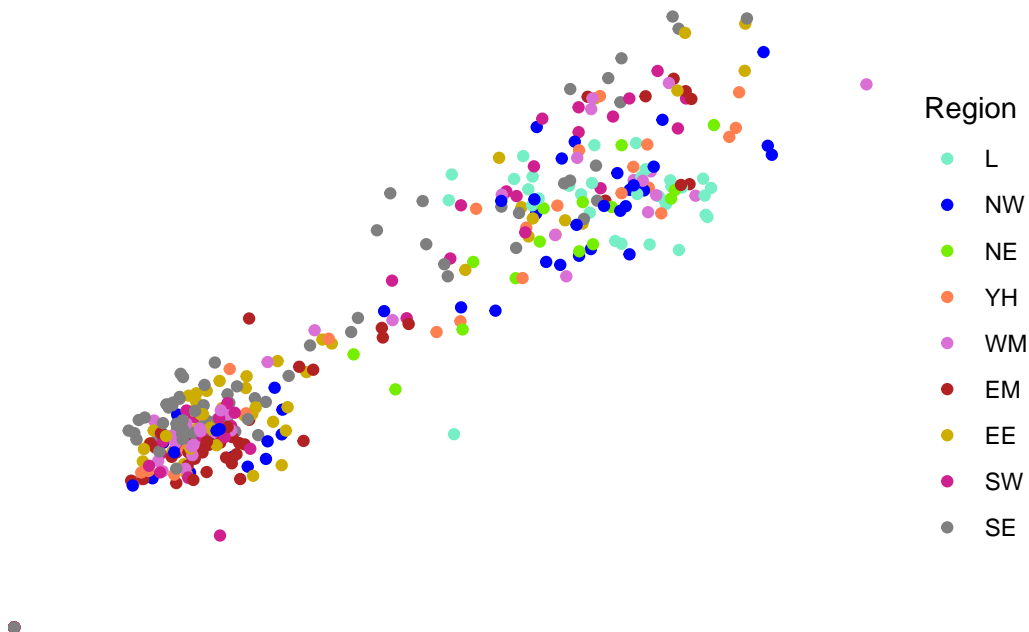
i Note

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

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

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

```
ggplot(csp_nolon_2020_new) +  
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = region_fct)) +  
  scale_x_continuous(name = "SFA", trans = "log1p") +  
  scale_y_continuous(name = "Council tax", trans = "log1p") +  
  scale_colour_manual(name = "Region", values = region_palette) +  
  theme_void( )
```

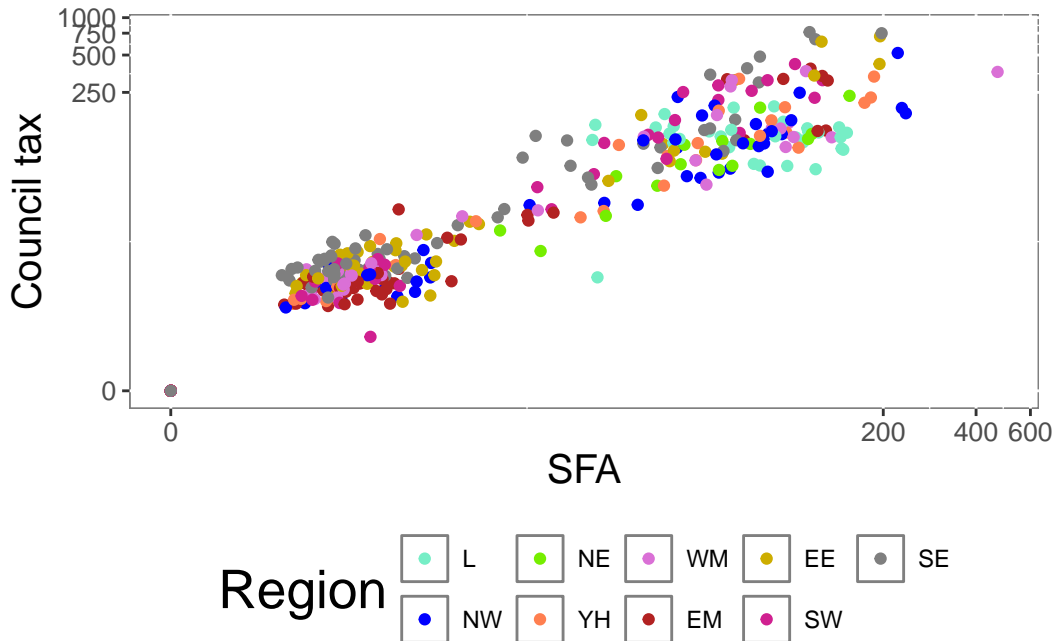



One benefit of using themes is that all visualisations will be consistent in terms of colour scheme, font size and gridlines. Although there are pre-built themes, we are able to create our own and save them as functions. These can then be used in place of R's themes. For example:

```
# Create a theme function
theme_intro_course <- function( ) {
  # Move the legend to the bottom
  theme(legend.position = "bottom",
        # Make the axis labels font size 10
        axis.text = element_text(size = 10),
        # Make the axis titles font size 15
        axis.title = element_text(size = 15),
        # Make the graph title font size 20
        title = element_text(size = 20),
        # Make the plot area white with a grey outline
        panel.background = element_rect(fill = "white", colour = "grey50"))
}
```

The function `theme_intro_course` can be added to the end of any visualisation and will move the legend to the bottom of the graph, change the axis text to size 10, the axis titles to size 15, the plot title to size 20, and the graph background to white with a grey outline:

```
ggplot(csp_nolon_2020_new) +
  geom_point(aes(x = sfa_2020, y = ct_total_2020, colour = region_fct)) +
  scale_x_continuous(name = "SFA", trans = "log1p") +
  scale_y_continuous(name = "Council tax", trans = "log1p") +
  scale_colour_manual(name = "Region", values = region_palette) +
  theme_intro_course( )
```



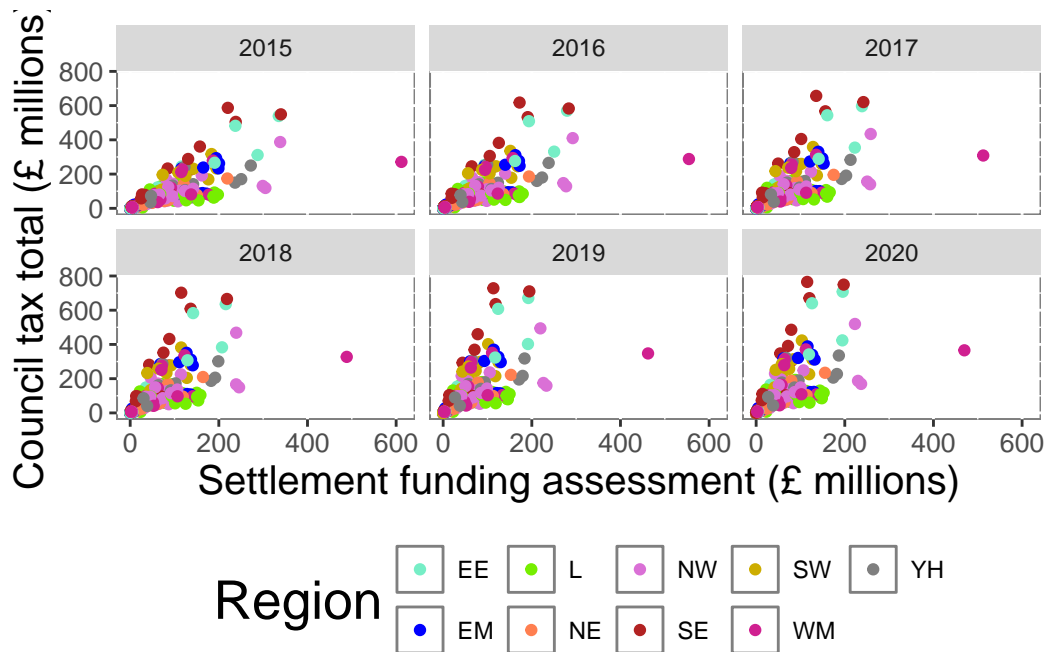
Creating a custom theme is useful to ensure all visualisations are formatted consistently.

4.1.7 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, if we wish to show the relationship between SFA, council tax total and regions over the entire time period, we may wish to create a scatterplot per year. Faceting allows us to do this in one piece of code rather than repeating it per year. Faceting will also ensure that plots are on the same scale and therefore easier to compare. The function `facet_wrap` creates these faceted plots:

```
# Take the long formatted dataset
csp_long2 %>%
  # Remove the Greater London Authority row
  filter(authority != "Greater London Authority") %>%
  ggplot( ) +
  # Plot the SFA against CT total and colour by region
  geom_point(aes(x = sfa, y = ct_total, colour = region)) +
  # Use the region colour palette
  scale_colour_manual(name = "Region", values = region_palette) +
  # Change the axis titles
  labs(x = "Settlement funding assessment (£ millions)",
       y = "Council tax total (£ millions)", colour = "Region") +
  # Separate data into a plot per region
  facet_wrap(~ year) +
  # Use the intro course theme
  theme_intro_course()
```



Exercise 7

Use an appropriate data visualisation to show how the total spend in each local authority has changed over the years between 2015 and 2020. Choose a visualisation that shows these trends over time and allows us to compare them between regions.

5 Reproducible research with RMarkdown

5.1 Introduction to RMarkdown

RMarkdown is a tool that is used to author high-quality documents, making it easy to communicate results efficiently. One of the main appeals of RMarkdown is that it is easy to integrate R code and output seamlessly into a document, encouraging openness and reproducibility in research.


There are a number of ways we can use RMarkdown to enhance the research process, such as:

- Generating reports to show the latest findings in a project, combining research output with interpretations. Reports can be automated within RMarkdown, to ensure outputs show the latest data.
- Keeping track of projects as an alternative to a notebook. Documents can include R code and visualisations alongside thoughts and comments on findings so far.
- Creating a collaborative document that can be shared with colleagues. The inclusion of R code in documents provides an audit trail, making it easy to carry out quality assurance and resolve discrepancies in results.

Before we begin working with RMarkdown in RStudio, we must first download and install the `rmarkdown` package as we would any other package:

```
install.packages("rmarkdown")  
  
library(rmarkdown)
```

5.1.1 Creating an RMarkdown files

RMarkdown files (`.Rmd`) are created and saved separately to the script files we have been using up to now on the course. To create a new RMarkdown file, either use the drop-down menu, following the *File -> New File -> R Markdown...* options, or using the  icon and selecting *R Markdown....*

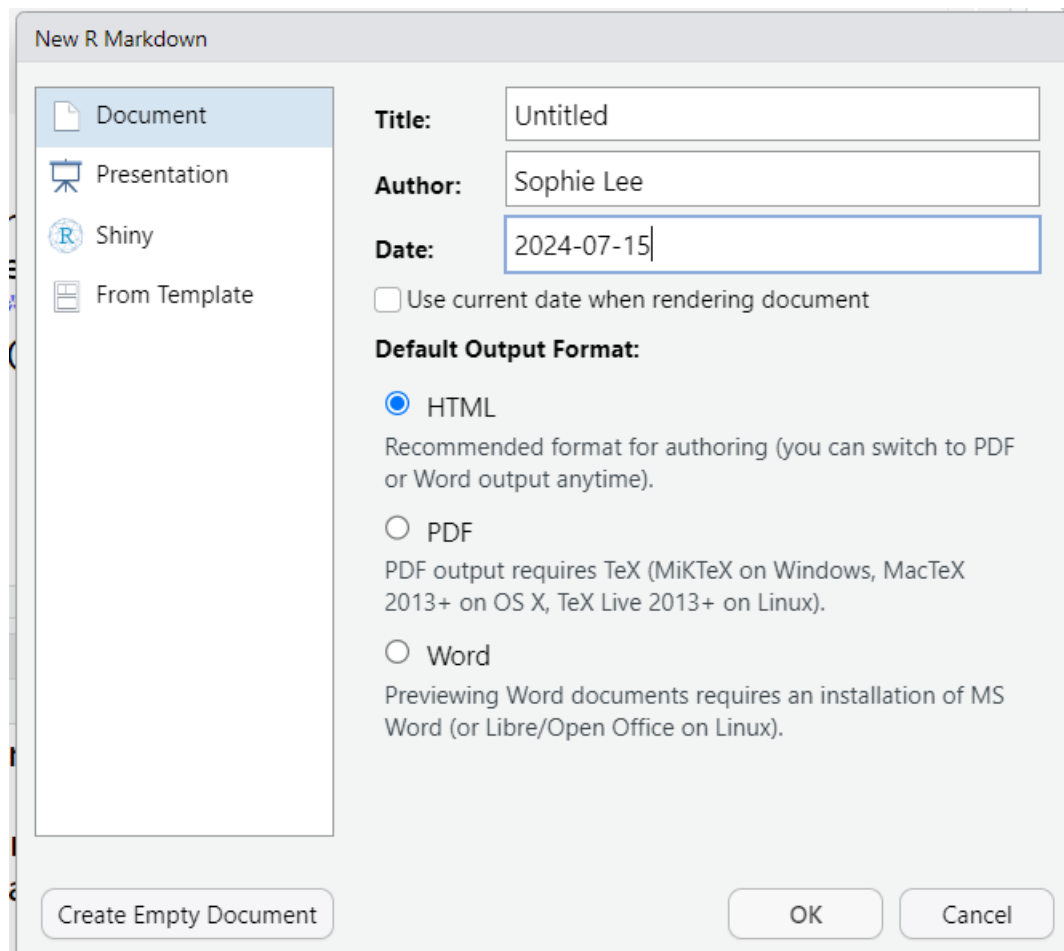


Figure 5.1: Rmarkdown new file

When creating a new RMarkdown file, we are given the option to set the title, author and date of the new document. We are also given options to select the type of document, presentation, or Shiny app we would like to create. This does not give a comprehensive list of documents available within RMarkdown and can be changed later. We will discuss output document types in more detail later in the session.

Clicking ‘OK’ on this window will produce an RMarkdown file (`.Rmd`) with some example code. If we do not want this, there is an option to ‘Create Empty Document’ on the bottom left of the window.

5.1.2 Rmarkdown content

RMarkdown files contain three main types of content:

- A YAML header (this sets the global options for the document)
- Text, or syntax (this includes headings and comments)
- Code chunks containing R code

5.1.2.1 The YAML header

The first part of an RMarkdown script, surrounded by ‘`---`’ is known as the **YAML header**. This sets global options for the document that will be produced by the script. YAML headers can include the title, author and date of a document, the output document type, table of contents options, and can include code to edit the appearance of text and figures.

For this course, we will just use the YAML to define the **title**, **author**, **date**, and **output** of our document:

```
---
title: "Introduction to R with Tidyverse"
author: My name
date: 2024-07-15
output: html_document
---
```

There are many output document types that can be produced using RMarkdown. Some of the most common include:

- **html_document**: HTML document, **.html**
- **pdf_document**: PDF document, **.pdf**, created using a LaTeX template
- **word_document**: Microsoft word document (**.docx**)
- **odt_document**: OpenDocument text document (**.odt**), similar to Microsoft Word/Google Docs but compatible with free word processors)
- **github_document**: Github document (**.md**, markdown files that are compatible with Github and are converted to HTML when viewed there)
- **powerpoint_presentation**: Powerpoint presentation (**.pptx**)

RMarkdown can also be combined with other R packages to create books (via **bookdown**), websites (via **blogdown**) and interactive dashboards (via **flexdashboard**).

5.1.2.2 RMarkdown syntax

RMarkdown text, or syntax, will generally make up the majority of a RMarkdown file. This can include headers and subheadings, equations, and any other text or comments in the document. Text is formatted using **markdown syntax**. A detailed list of syntax commands are given in the RMarkdown [cheatsheet](#). Common syntax commands that may be used in an RMarkdown document include:

- `*italic*`
 - `**bold**`
 - Add ``code`` into text
 - `#` Header 1
 - `##` Header 2
 - ...
 - `#####` Header 6
 - `[This is a link](link url)`
 - `![caption](image.png)`
- Unordered list
 - List with indent
1. Ordered list
 - With indent
 2. Second item

Equation: $r^2 = (x - a)^2 + (y - b)^2$


RMarkdown equations are built using the same language as LaTeX. [See here](#) for a list of mathematical symbols that can be used in these equations.

5.1.2.3 Code chunks

Code chunks allow us to embed R code and outputs into our documents. This is one of the main draw of RMarkdown as it removes the need to copy and paste or import results from R into another document.

Code chunks are pieces of code that begin ````{r}` and end `````. For example,

```
```{r}
1 + 1
```
```

Code chunks can be created by manually typing these wrappers, by clicking the  icon and selecting 'R', or using the keyboard shortcut `ctrl + alt + i` on Windows, and `Command + Option + i` on Mac.

Code chunks can be given titles to make an RMarkdown script easier to navigate (these will appear under the script window where lists of subheadings appear in script files). These are added inside the opening of the code chunk: ````{r chunk title}`.

There are a number of chunk options to customise which code/output to display in the document. These are included in the opening of a chunk, after the title, separated by commas `,`. Some of the most common, include:

- `echo = TRUE/FALSE`: whether to display code in the output document
- `eval = TRUE/FALSE`: whether to run the code in the chunk or not
- `include = TRUE/FALSE`: whether to include anything from the chunk (code or output) in the document
- `error/warning/message = TRUE/FALSE`: whether to display error/warning/other messages in the document

Top tip: It may be useful to add a setup code chunk at the beginning of an RMarkdown file that loads any packages and datasets that are required for the rest of the document. These can also include universal options for future code chunks to avoid repeating the code, using the `knitr::opts_chunk$set` function.

For example:

```
```{r setup, include = FALSE}
Set global options for code chunks

Do not show any R code or messages unless specified
knitr::opts_chunk$set(echo = FALSE, message = FALSE)

Load in the tidyverse package
library(tidyverse)
```
```


5.1.3 Compiling RMarkdown documents

Compiling RMarkdown actually requires multiple steps and programmes. Luckily for us, this process takes place in the background so we don't need to be aware of these steps happening!

Generating an output file from RMarkdown is know **knitting** a document. This process sends the `.Rmd` file to another R package `knitr` (which is installed alongside `rmarkdown`), which executes all the code chunks in the document and creates a markdown `.md` file including the code and output. This markdown file is then processed by another programme **pandoc** which converts markdown code into the finished document.



Figure 5.2: RMarkdown to document process

To **knit** an RMarkdown file in RStudio is very simple. Either click the  icon above the RMarkdown script, or use the keyboard shortcut `ctrl + shift + k` on Windows or `Command + shift + k` on Mac. This initiates the process above and will return an output document (if there are no errors!) in the requested format to the working directory.

5.1.4 Data visualisation in RMarkdown

Output such as graphs and tables can be embedded in code chunks, the code used to create them will be the same as it would be in any other R script.

Note

Often, when providing output in RMarkdown, we often do not want to show the code that was used to create this. Make sure to add `echo = FALSE` to the opening of the code chunk.

5.1.4.1 Graphs in RMarkdown

`ggplot` can be used to create graphs that are embedded within code chunks and included in an output document. For example, we could use the data from previous sections to show the relationship between Settlement Funding Assessment (SFA) and council tax total in English local authorities in 2020, colour code by regions in a scatterplot:

```

```{r scatterplot sfa_2020 and ct_total_2020 by region, message = FALSE}
Load and tidy the 2020 data
read_csv("data/CSP_2020.csv") %>%
 # Remove the Greater London Authority row
 filter(authority != "Greater London Authority") %>%
 # Convert region variable to factor
 mutate(region_fct = factor(region,
 levels = c("L", "NW", "NE", "YH", "WM",
 "EM", "EE", "SW", "SE"))) %>%

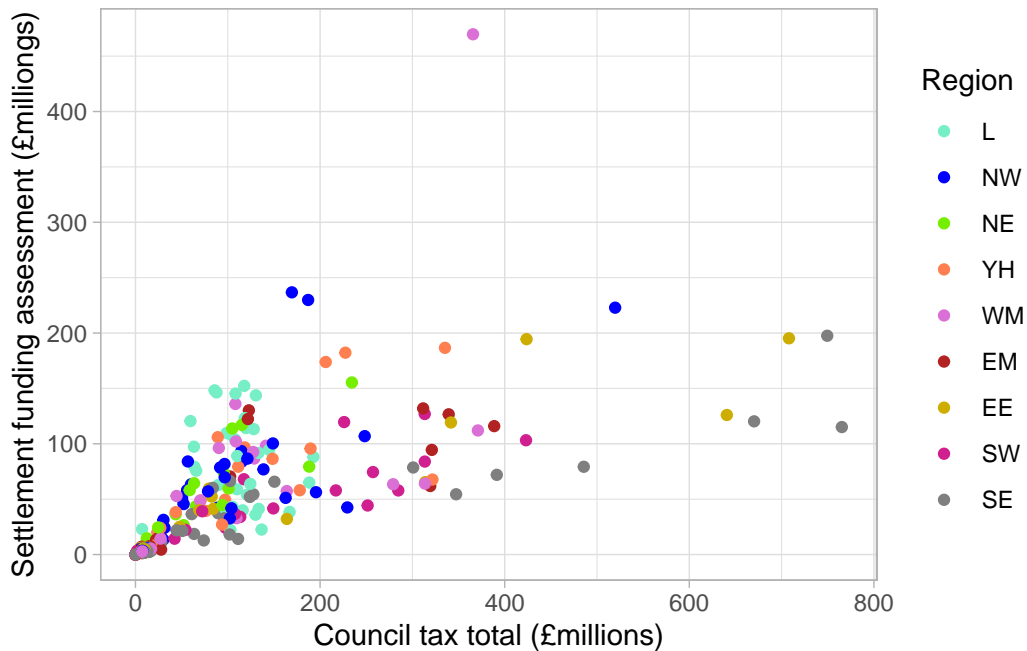
```

```

Create a ggplot
ggplot() +
Scatterplot definition
geom_point(aes(x = ct_total_2020, y = sfa_2020, colour = region_fct)) +
Add colour palette for region
scale_colour_manual(values = c("aquamarine2", "blue", "chartreuse2",
 "coral", "orchid", "firebrick",
 "gold3", "violetred", "grey50")) +

Change axis/label titles
labs(x = "Council tax total (£millions)",
 y = "Settlement funding assessment (£milliongs)",
 colour = "Region") +
theme_light()
` ``

```



#### 5.1.4.2 Tables in RMarkdown

There are a number of ways to include tables within RMarkdown which can either be entered manually, or generated using an R package. The choice of approach to creating tables depends on the format and size of the data, the amount of flexibility you would like to customise the output, the type of output document you are creating, and personal preference of how it should look.

In this course, we will look at how tables can be created using RMarkdown syntax (without the need for additional packages), and using the `kable` function within the `knitr` package.

### Manually creating tables

Tables can be created in RMarkdown syntax, using the `|` symbol to separate columns, and dashes `-` to separate column headings from the body of the table. These are created outside of code chunks within the text. For example,

```
Header 1 | Header 2 | Header 3 |
|-----|-----|-----|
| This | Is | A |
| Very | Simple | Table |
```

produces the following output:

Header 1	Header 2	Header 3
This	Is	A
Very	Simple	Table

Colons can be added to the header/body separator row of the table to control the justification of the text in each column. For example,

```
| Left | Right | Center | Default |
|:----|-----:|:-----:|:-----|
| This | Is | Another | Simple |
| Table | But | It is | Justified |
```

produces the following output:

Left	Right	Center	Default
This	Is	Another	Simple
Table	But	It is	Justified

### Creating tables from data frames

The `knitr` package that compiles RMarkdown files contains the `kable` function that can be used to create simple data tables. The `kable` function requires data to be stored as a matrix, data frame, or tibble object (although these can be easily created using the `matrix`, `data.frame` or `tibble` functions). Accessing the help file `?knitr::kable` gives a list of arguments that can be used to customise these tables.

### Note

As these tables are created using R functions, they must be generated within a code chunk.

For example, we can create a simple data table using **kable** showing the first 6 rows of the **mtcars** dataset (a dataset pre-loaded into base R):

```
```{r mtcars table, echo = FALSE}
knitr::kable(head(mtcars))
```
```

which produces the following output:

|                | mpg  | cyl | disp | hp  | drat | wt    | qsec  | vs | am | gear | carb |
|----------------|------|-----|------|-----|------|-------|-------|----|----|------|------|
| Mazda RX4      | 21.0 | 6   | 160  | 110 | 3.90 | 2.620 | 16.46 | 0  | 1  | 4    | 4    |
| Mazda RX4 Wag  | 21.0 | 6   | 160  | 110 | 3.90 | 2.875 | 17.02 | 0  | 1  | 4    | 4    |
| Datsun 710     | 22.8 | 4   | 108  | 93  | 3.85 | 2.320 | 18.61 | 1  | 1  | 4    | 1    |
| Hornet 4 Drive | 21.4 | 6   | 258  | 110 | 3.08 | 3.215 | 19.44 | 1  | 0  | 3    | 1    |
| Hornet         | 18.7 | 8   | 360  | 175 | 3.15 | 3.440 | 17.02 | 0  | 0  | 3    | 2    |
| Sportabout     |      |     |      |     |      |       |       |    |    |      |      |
| Valiant        | 18.1 | 6   | 225  | 105 | 2.76 | 3.460 | 20.22 | 1  | 0  | 3    | 1    |

Where data are not already saved as an object, we need to create them first before generating a table. For example, the table we created manually earlier can be recreated using the **kable** function, by first creating a data frame with the information, and then piping it through to the function:

```
```{r kable table}
# Create a data frame with the table information
data.frame(col1 = c("This", "Very"),
           col2 = c("Is", "Simple"),
           col3 = c("A", "Table")) %>%
  knitr::kable(., col.names = c("Header 1", "Header 2", "Header 3"))
```
```

| Header 1 | Header 2 | Header 3 |
|----------|----------|----------|
| This     | Is       | A        |
| Very     | Simple   | Table    |

## Other ways to create tables

Although the `kable` function and RMarkdown syntax tables do not require additional R packages to be installed, they are fairly simple and do not give many options to customise the tables. For a more flexible alternative, I would recommend looking at the `flextable` package, which gives a much wider range of customisable features. The `flextable` user manual can be accessed for free from [this website](#).

## Exercise 8

Create an RMarkdown file that creates a html report describing the trends in core spending power in English local authorities between 2015 and 2020. Your report should include:

- A summary table of the total spending per year per region
- A suitable visualisation showing how the total annual spending has changed over this period, compared between regions
- A short interpretation of the table and visualisation

### Note

You are not expected to be an expert in this data! Interpret these outputs as you would any other numeric variable measured over time.

# Data description

## What is ‘CSP’?

The data we will be using throughout this course relates to the Core Spending Power (CSP) of English local authorities between 2015 and 2020. This is a measure of the resources available to local authorities in England to fund service delivery. The CSP is broken down into several components, presented as variables in the data. These components include:

- Settlement Funding Assessment (**sfa**)
- Compensation for under-indexing the business rates multiplier (**under\_index**)
- Income from council tax (**ct\_total**)
- New Homes Bonus (**nhb**)
- Rural Services Delivery Grant (**rsdg**)

Spending power is given in millions of pounds (£). The data were provided by the UK government’s Department for Levelling Up, Housing and Communities. Full guidance on the data can be found on the [Department’s website](#). A brief description of the variables included in the data are given below.

## Descriptions of variables

### Identifier variables

Each dataset contains a unique identifier code variable, **ons\_code**. This is a code given by the Government’s Office for National Statistics (ONS), and is used to join different datasets. There is also an **authority** variable which contains the local authority name (to see where each local authority lies on a map, you can visit the [Government’s geoportal website](#)).

### Regions of England

In addition to each local authority’s unique code and name, we are given the region that they lie within. England is separated into 9 regions (shown on [this map](#)) which are given as acronyms in the data. These are:

- L = London
- NW = North West
- NE = North East
- YH = Yorkshire and the Humber
- WM = West Midlands
- EM = East Midlands
- EE = East England
- SW = South West
- SE = South East

## **Settlement Funding Assessment (SFA)**

The Settlement Funding Assessment (**sfa** in the data) is the baseline funding level of local authorities, and includes the Revenue Support Grant (a central government grant given to local authorities).

## **Under-indexing business rate multipliers**

The **under\_index** variable is given to compensate local authorities that under-indexed business rate multipliers in previous years (i.e. those that used a measure of inflation that was lower than that should have been used).

## **Council tax**

Council tax (**ct\_total**) is the income made by each local authority from council tax. In England, the amount of council tax charged to residents is set by each local authority to make up additional revenue needed to cover planned spending.

## **New Homes Bonus**

The **nhb** variables is the funding received as part of the New Homes Bonus, a government incentive to encourage local authorities to promote new housing development.

## **Rural Services Delivery Grant**


The **rsdg** variable is funding received as part of the Rural Services Delivery Grant, provided to rural councils to recognise additional costs in these areas.

# Exercise solutions

## Exercise 1

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

## Solutions

1. To open a new R script, click the  icon and select 'R script'.
2. Save this file by following **File -> Save as...** from the drop-down menu, selecting the folder you are working from in this course, and giving the file an appropriate name.

## 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 calculate the following calculations. Add the result to the same line of the script file in a way that ensures there are no errors in the code.
  - a.  $64^2$
  - b.  $3432 \div 8$
  - c.  $96 \times 72$

When you have finished this exercise, select the entire script file (using **Ctrl + a** on windows or **Command + a** on Mac) and run it to ensure there are no errors in the code.

## Solutions

1. Add a **#** symbol to the script file before printing your name and the date,
2. After running the calculation, copy and paste the result after a **#** symbol to ensure there are no errors:



```
64 ^ 2 # 4096
[1] 4096
3432 / 8 # 429
[1] 429
96 * 72 # 6912
[1] 6912
```

## Exercise 3

1. How many local authorities were included in the London region?
2. Give three different ways that it would be possible to select all spend variables (sfa\_2020, nhb\_2020, etc.) from the CSP\_2020 dataset.
3. Create a new tibble, `em_2020`, that just includes local authorities from the East Midlands (EM) region.
  - a. How many local authorities in the East Midlands had an SFA of between £5 and 10 million?
  - b. Create a new variable with the total overall spend in 2020 for local authorities in the East Midlands.

## Solutions

1. First `filter` the data to return only rows which belong to the London region, then count the number of rows in this subgroup:

```
csp_2020 %>%
 filter(region == "L") %>%
 count()
A tibble: 1 x 1
n
<int>
1 34
```

2. There are many different ways to select variables from a dataset. For a list of selection helpers, check the helpfile `?tidyr_tidy_select`. Some examples include:

```
Using the : symbol to return consecutive columns

By variable name:
select(csp_2020, sfa_2020:rsdg_2020)

Or by column number:
select(csp_2020, 4:9)

Returning all variables with names ending "_2020"
select(csp_2020, ends_with("_2020"))

Return all numeric variables
select(csp_2020, where(is.numeric))

Return all variables that are not character
select(csp_2020, where(!is.character))
```

3. To create a new tibble, use **filter** to select the subgroup where region is “EM”, and attach as an object using the **<-** symbol

```
em_2020 <- filter(csp_2020, region == "EM")
```

- a. Use **filter** to select the subgroup and then **count** the number of rows:

```
em_2020 %>%
 filter(between(sfa_2020, 5, 10)) %>%
 count()
A tibble: 1 x 1
n
<int>
1 3
```

- b. Use the **mutate** function to create a new variable from the existing ones. **Hint:** If you are not sure of the variable names in the data, use the **names** function and copy them from the console:

```
names(em_2020)
[1] "ons_code" "authority" "region" "sfa_2020"
[5] "under_index_2020" "ct_total_2020" "nhb_2020" "nhb_return_2020"
[9] "rsdg_2020"
```

```
em_2020 <- em_2020 %>%
 mutate(total_spend = sfa_2020 + under_index_2020 + ct_total_2020 +
 nhb_2020 + nhb_return_2020 + rsdg_2020)
```

## Exercise 4

1. Create a data frame with the minimum, maximum and median total spend per year for each region.
2. Produce a frequency table containing the number and percentage of local authorities in each region.
3. Convert the data object `csp_long2` back into wide format, with one row per local authority and one variable per total spend per year (**HINT:** start by selecting only the variables you need from the long data frame). Use the help file `?pivot_wider` and `vignette("pivot")` for more hints.
4. Using your new wide data frame, calculate the difference in total spending for each local authority between 2015 and 2020. How many local authorities have had an overall reduction in spending since 2015?

## Solutions

1. Use the `summarise` function after grouping by the `year` and `region` variables:

```
csp_long2 %>%
 group_by(region, year) %>%
 summarise(min_spend = min(total_spend),
 max_spend = max(total_spend),
 median_spend = median(total_spend)) %>%
 ungroup()
A tibble: 54 x 5
region year min_spend max_spend median_spend
<chr> <dbl> <dbl> <dbl> <dbl>
1 EE 2015 0 883. 15.9
2 EE 2016 0 860. 16.2
3 EE 2017 0 845. 15.0
4 EE 2018 0 860. 14.4
5 EE 2019 0 874. 14.7
6 EE 2020 0 915. 15.2
7 EM 2015 0 492. 12.4
8 EM 2016 0 479. 12.0
```

```
9 EM 2017 0 475. 11.1
10 EM 2018 0 483. 11.0
i 44 more rows
```

2. To calculate the percentage of local authorities in each region, we need the total number in each region and the overall number of local authorities:

```
Use the csp_2020 data as only require one row per local authority
csp_2020 %>%
 # Begin by calculating number of local authorities per region
 group_by(region) %>%
 # Count number of rows in each group
 summarise(n_la_region = n()) %>%
 ungroup() %>%
 # Create a new variable with the total number of local authorities (the sum)
 mutate(n_la_overall = sum(n_la_region),
 # Calculate the percentage (and round to make easier to read)
 perc_la_region = round((n_la_region / n_la_overall) * 100, 2)) %>%
 # Remove the total local authority column
 select(-n_la_overall)
A tibble: 9 x 3
region n_la_region perc_la_region
<chr> <int> <dbl>
1 EE 57 14.4
2 EM 51 12.9
3 L 34 8.59
4 NE 15 3.79
5 NW 46 11.6
6 SE 81 20.4
7 SW 47 11.9
8 WM 38 9.6
9 YH 27 6.82
```

3. Use the `pivot_wider` function, use the year to set the new variable names suffix (`names_from =`), add a prefix to avoid variable names beginning with a number (`names_prefix =`), and take the `values_from` the current `total_spend` column:

```
csp_total_wide <- csp_long2 %>%
 # Select variables to keep
 select(ons_code:year, total_spend) %>%
 pivot_wider(names_from = year,
 names_prefix = "total_spend_",
 values_from = total_spend)
```

4. Begin by using `mutate` to create a variable with the difference between total spend 2015 - 2020. Use `filter` to return rows where there is a reduction in spend, count the number of rows:

```
csp_total_wide %>%
 mutate(total_diff = total_spend_2020 - total_spend_2015) %>%
 filter(total_diff < 0) %>%
 count()
A tibble: 1 x 1
n
<int>
1 234
```

## Exercise 5

1. Create a new data object containing the 2020 CSP data without the Greater London Authority observation. Name this data frame `csp_nolon_2020`.
2. Using the `csp_nolon_2020` data, create a data visualisation to check the distribution (or shape) of the SFA variable.
3. Based on the visualisation above, create a summary table for the SFA variable containing the minimum and maximum, and appropriate measures of the centre/average and spread.

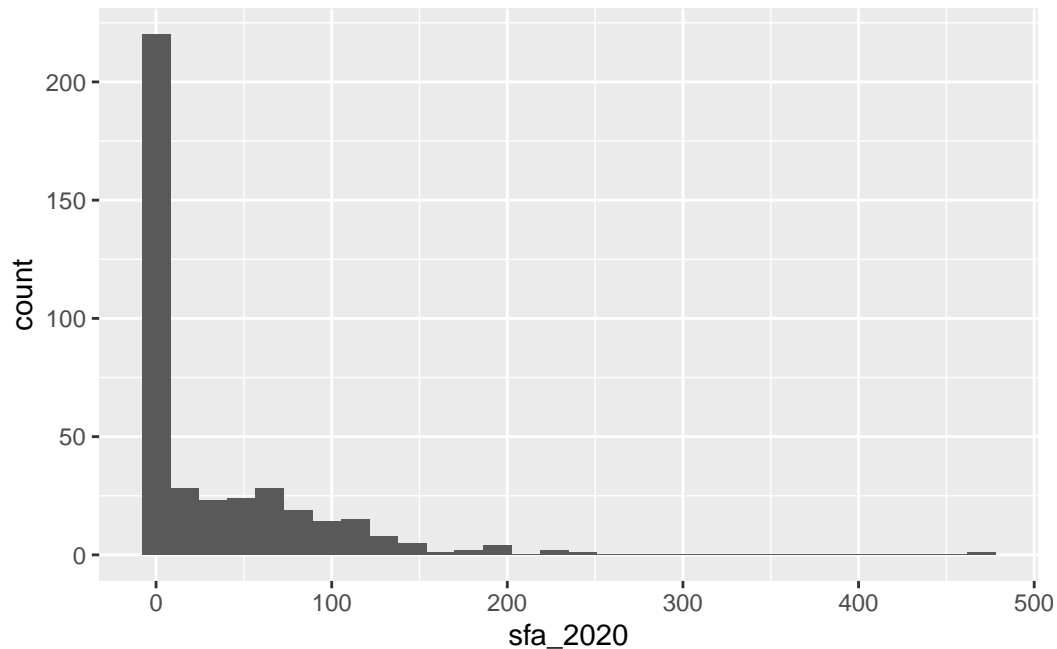
## Solutions

1. Create a new object using the `<-` symbol, use `filter` to remove the duplicate row:

```
csp_nolon_2020 <- filter(csp_2020, authority != "Greater London Authority")
```

2. Histograms are used to check the distribution of numeric variables:

```
ggplot(data = csp_nolon_2020) +
 geom_histogram(aes(x = sfa_2020))
```



3. The histogram shows that the `sfa_2020` variable is very skewed, therefore the `median` and `IQR` are the most appropriate measures of centre and spread:

```
summarise(csp_nolon_2020,
 min_sfa = min(sfa_2020),
 max_sfa = max(sfa_2020),
 median_sfa = median(sfa_2020),
 iqr_sfa = IQR(sfa_2020))
A tibble: 1 x 4
min_sfa max_sfa median_sfa iqr_sfa
<dbl> <dbl> <dbl> <dbl>
1 0 470. 4.62 54.7
```

## Exercise 6

1. What is the problem with the following code? Fix the code to change the shape of all the points.

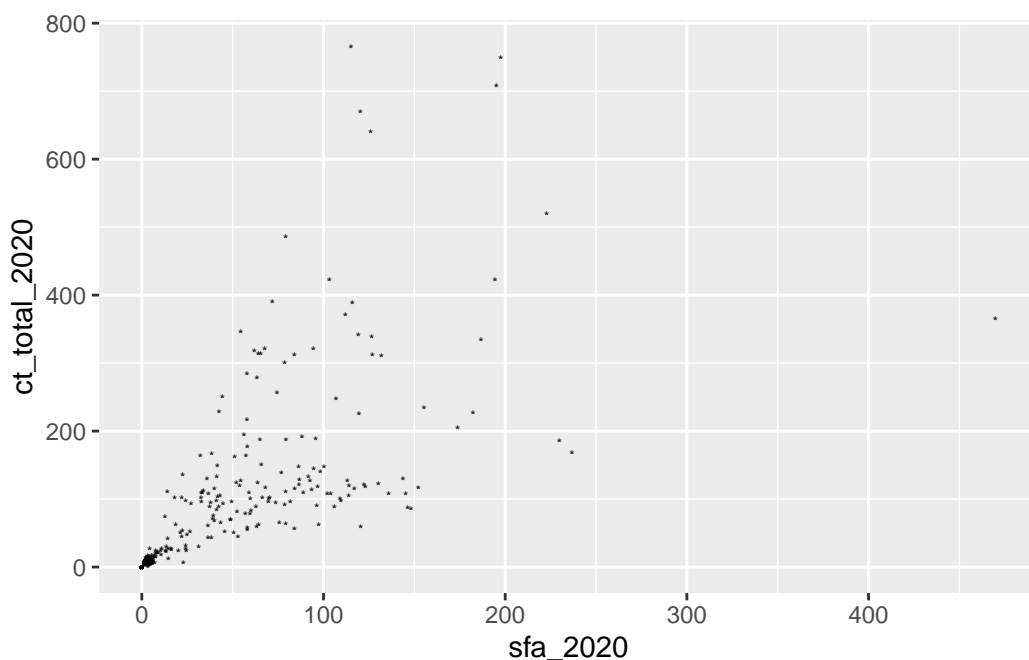
```
ggplot(csp_nolon_2020) +
 geom_point(aes(x = sfa_2020, y = ct_total_2020, shape = "*"))
```

2. Add a line of best fit to the scatterplot showing the relationship between SFA and council tax total (hint: use `geom_smooth`).
3. Add a line of best fit for each region (hint: make each line a different colour).

## Solutions

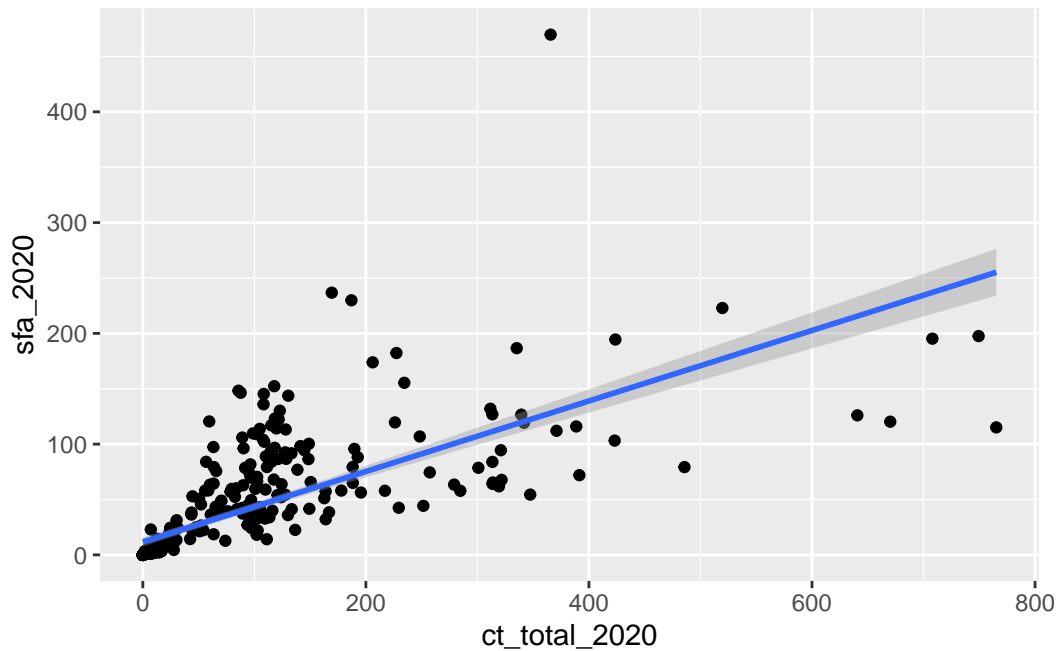
1. Only aesthetics determined by variables in the data should lie inside the `aes` function, the `shape` argument should be outside of this:

```
ggplot(csp_nolon_2020) +
 geom_point(aes(x = sfa_2020, y = ct_total_2020), shape = "*")
```



2. The function `geom_smooth` adds a line of best fit, make sure to set `method = lm` to fit a linear trend:

```
ggplot(data = csp_nolon_2020, aes(x = ct_total_2020, y = sfa_2020)) +
 geom_point() +
 geom_smooth(method = "lm")
```

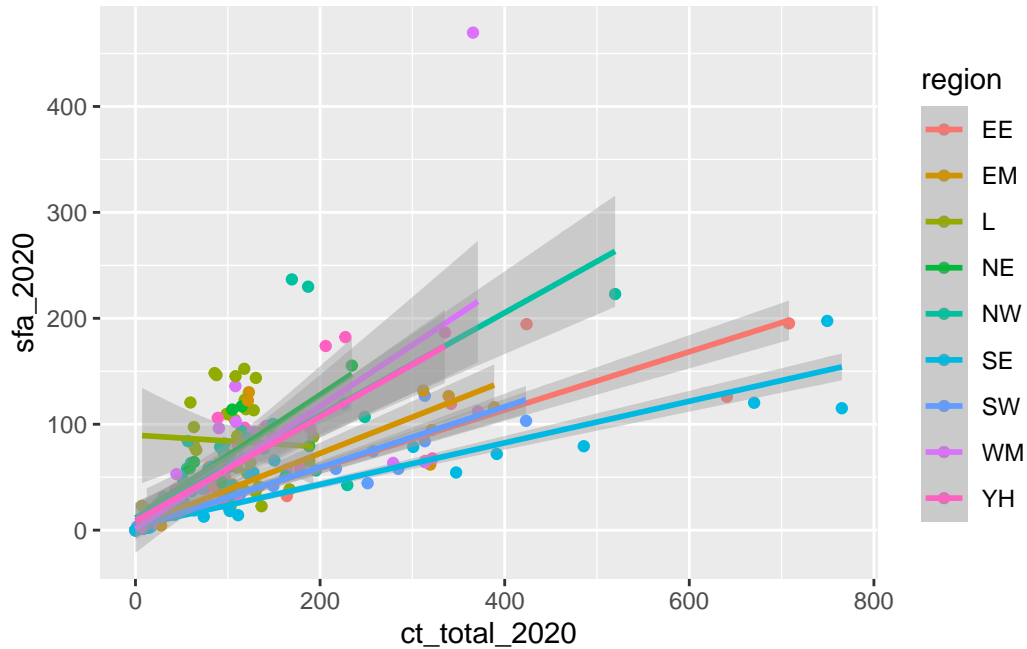


**Hint:** To reduce repetitive coding, setting `aes` in the `ggplot` function applies these to the entire object.

3. A line of best fit for each group simply requires adding this to the `aes` function as a colour:

```
ggplot(data = csp_nolon_2020,
 aes(x = ct_total_2020, y = sfa_2020, colour = region)) +
 geom_point() +
 geom_smooth(method = "lm")
```





## Exercise 7

Use an appropriate data visualisation to show how the total spend in each local authority has changed over the years between 2015 and 2020. Choose a visualisation that shows these trends over time and allows us to compare them between regions.

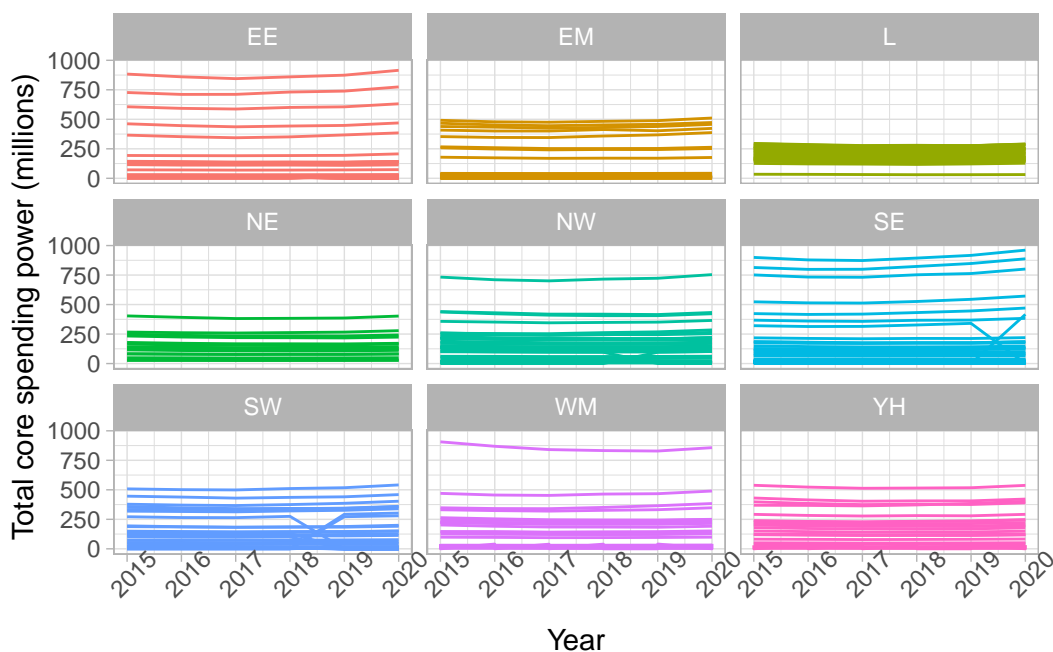
## Solution

The most appropriate plot to show a change in variable over time is a line graph (with year on the x-axis and spend on the y-axis). To compare these between regions, we could set the colour of these lines, but as there are so many local authorities, this would overload the graph and make it hard to compare. As an alternative, we can facet this graph by region to show the line graphs on the same scale on the same output.

Be sure to set appropriate axis labels, font sizes, etc.

```
Remove the Greater London Authority duplicate
csp_long2 %>%
 filter(authority != "Greater London Authority") %>%
 ggplot() +
```

```
Need to add a group to know what each line represents
geom_line(aes(x = year, y = total_spend, group = ons_code,
 # OPTIONAL: colour by region to make it prettier!
 colour = region)) +
facet_wrap(~ region) +
labs(x = "Year", y = "Total core spending power (millions)") +
Add theme_light to make the background a nicer colour
theme_light() +
Rotate the x-axis labels to avoid overlap
theme(axis.text.x.bottom = element_text(angle = 45),
 # Remove the legend (not needed, we have labels on the facets)
 legend.position = "none")
```



## Exercise 8

Create an RMarkdown file that creates a html report describing the trends in core spending power in English local authorities between 2015 and 2020. Your report should include:

- A summary table of the total spending per year per region
- A suitable visualisation showing how the total annual spending has changed over this period, compared between regions

- A short interpretation of the table and visualisation

**Note:** You are not expected to be an expert in this data! Interpret these outputs as you would any other numeric variable measured over time.

## Solutions

There are many different correct solutions to this exercise. All RMarkdown files should begin with a YAML header similar to the one below:

```

title: "Core spending power in English local authorities, 2015 - 2020"
author: Sophie Lee
output: html_document

```

Next, you may have a code chunk that sets up the global chunk options, loads any packages you needed, and loads the data that we will be using for the report. For example:

```
```{r setup, include = FALSE}
# Set global chunk options to not show R code or messages
knitr::opts_chunk$set(echo = FALSE, message = FALSE)

# Load the tidyverse package
library(tidyverse)

# Load the long dataset
csp_long2 <- read_csv("data/CSP_long_201520.csv")
```
```

You may have begun with an introduction using RMarkdown syntax:

```
Introduction
The following report will investigate the trends in core spending power
across England between 2015 and 2020. All values are give in millions
of pounds.

The core spending power was made up of the following provisions:

- Settlement funding assessment (SFA)
- Compensation for under-indexing the business rates multipliers
```

- council tax
- New homes bonus
- New homes bonus returned funding
- Rural Services Delivery Grant (RSDG)

Followed by a summary table, created using `summarise` and displayed using `kable`:

```
Total core spending power by region
Below is a summary table containing the total core spending power per year
per region, given in millions of £:

```{r csp total summary table}
csp_long2 %>%
  group_by(region, year) %>%
  summarise(min_spend = min(total_spend),
            max_spend = max(total_spend),
            median_spend = median(total_spend),
            iqr_spend = IQR(total_spend)) %>%
  ungroup() %>%
  knitr::kable(.,
               col.names = c("Region", "Year", "Minimum",
                             "Maximum", "Median", "IQR"))
```
```

Then an additional code chunk producing a faceted line chart, similar to the one in Exercise 7:

```
```{r}
csp_long2 %>%
  filter(authority != "Greater London Authority") %>%
  ggplot() +
  geom_line(aes(x = year, y = total_spend, group = ons_code,
                colour = region)) +
  facet_wrap( ~ region) +
  labs(x = "Year", y = "Total core spending power (millions)") +
  theme_light() +
  theme(axis.text.x.bottom = element_text(angle = 45),
        legend.position = "none")
```
```